|  |  |  |  |
| --- | --- | --- | --- |
| Fond-Rec_eFond-Rec_e  Fond-Rec_e | **International Telecommunication Union** | | |
|  |  | | |
| **ITU-T** | **Z.100** | |
| TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU | | **Annex F2**  (11/2018) |
|  | SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS  Formal description techniques (FDT) – Specification and Description Language (SDL) | | |
|  | Specification and Description Language – Overview of SDL‑2010  **Annex F2: SDL‑2010 formal definition: Static semantics** | | |
|  | Recommendation ITU‑T Z.100  – Annex F2 | | |



ITU-T Z-SERIES RECOMMENDATIONS

**LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS**

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

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

|  |
| --- |
| Recommendation ITU-T Z.100  Specification and Description Language – Overview of SDL‑2010  Annex F2  SDL‑2010 formal definition: Static semantics |

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

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| History   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Edition | Recommendation | Approval | Study Group | Unique ID[[1]](#footnote-2)\* | | 1.0 | ITU-T Z.100 | 1984-10-19 |  | [11.1002/1000/2222](http://handle.itu.int/11.1002/1000/2222) | | 1.1 | ITU-T Z.100 Annex A | 1984-10-19 |  | [11.1002/1000/6664](http://handle.itu.int/11.1002/1000/6664) | | 1.2 | ITU-T Z.100 Annex B | 1984-10-19 |  | [11.1002/1000/6665](http://handle.itu.int/11.1002/1000/6665) | | 1.3 | ITU-T Z.100 Annex C1 | 1984-10-19 |  | [11.1002/1000/6666](http://handle.itu.int/11.1002/1000/6666) | | 1.4 | ITU-T Z.100 Annex C2 | 1984-10-19 |  | [11.1002/1000/6667](http://handle.itu.int/11.1002/1000/6667) | | 1.5 | ITU-T Z.100 Annex D | 1984-10-19 |  | [11.1002/1000/6668](http://handle.itu.int/11.1002/1000/6668) | | 2.0 | ITU-T Z.100 | 1987-09-30 | X | [11.1002/1000/10954](http://handle.itu.int/11.1002/1000/10954) | | 2.1 | ITU-T Z.100 Annex A | 1988-11-25 |  | [11.1002/1000/6669](http://handle.itu.int/11.1002/1000/6669) | | 2.2 | ITU-T Z.100 Annex B | 1988-11-25 |  | [11.1002/1000/6670](http://handle.itu.int/11.1002/1000/6670) | | 2.3 | ITU-T Z.100 Annex C1 | 1988-11-25 |  | [11.1002/1000/6671](http://handle.itu.int/11.1002/1000/6671) | | 2.4 | ITU-T Z.100 Annex C2 | 1988-11-25 |  | [11.1002/1000/6672](http://handle.itu.int/11.1002/1000/6672) | | 2.5 | ITU-T Z.100 Annex D | 1988-11-25 | X | [11.1002/1000/3646](http://handle.itu.int/11.1002/1000/3646) | | 2.6 | ITU-T Z.100 Annex E | 1988-11-25 |  | [11.1002/1000/6673](http://handle.itu.int/11.1002/1000/6673) | | 2.7 | ITU-T Z.100 Annex F1 | 1988-11-25 | X | [11.1002/1000/3647](http://handle.itu.int/11.1002/1000/3647) | | 2.8 | ITU-T Z.100 Annex F2 | 1988-11-25 | X | [11.1002/1000/3648](http://handle.itu.int/11.1002/1000/3648) | | 2.9 | ITU-T Z.100 Annex F3 | 1988-11-25 | X | [11.1002/1000/3649](http://handle.itu.int/11.1002/1000/3649) | | 3.0 | ITU-T Z.100 | 1988-11-25 |  | [11.1002/1000/3153](http://handle.itu.int/11.1002/1000/3153) | | 3.1 | ITU-T Z.100 Annex C | 1993-03-12 | X | [11.1002/1000/3155](http://handle.itu.int/11.1002/1000/3155) | | 3.2 | ITU-T Z.100 Annex D | 1993-03-12 | X | [11.1002/1000/3156](http://handle.itu.int/11.1002/1000/3156) | | 3.3 | ITU-T Z.100 Annex F1 | 1993-03-12 | X | [11.1002/1000/3157](http://handle.itu.int/11.1002/1000/3157) | | 3.4 | ITU-T Z.100 Annex F2 | 1993-03-12 | X | [11.1002/1000/3158](http://handle.itu.int/11.1002/1000/3158) | | 3.5 | ITU-T Z.100 Annex F3 | 1993-03-12 | X | [11.1002/1000/3159](http://handle.itu.int/11.1002/1000/3159) | | 3.6 | ITU-T Z.100 App. I | 1993-03-12 | X | [11.1002/1000/3160](http://handle.itu.int/11.1002/1000/3160) | | 3.7 | ITU-T Z.100 App. II | 1993-03-12 | X | [11.1002/1000/3161](http://handle.itu.int/11.1002/1000/3161) | | 4.0 | ITU-T Z.100 | 1993-03-12 | X | [11.1002/1000/3154](http://handle.itu.int/11.1002/1000/3154) | | 4.1 | ITU-T Z.100 (1993) Add. 1 | 1996-10-18 | 10 | [11.1002/1000/3917](http://handle.itu.int/11.1002/1000/3917) | | 5.0 | ITU-T Z.100 | 1999-11-19 | 10 | [11.1002/1000/4764](http://handle.itu.int/11.1002/1000/4764) | | 5.1 | ITU-T Z.100 (1999) Cor. 1 | 2001-10-29 | 17 | [11.1002/1000/5567](http://handle.itu.int/11.1002/1000/5567) | | 6.0 | ITU-T Z.100 | 2002-08-06 | 17 | [11.1002/1000/6029](http://handle.itu.int/11.1002/1000/6029) | | 6.1 | ITU-T Z.100 (2002) Amd. 1 | 2003-10-29 | 17 | [11.1002/1000/7091](http://handle.itu.int/11.1002/1000/7091) | | 6.2 | ITU-T Z.100 (2002) Cor. 1 | 2004-08-29 | 17 | [11.1002/1000/356](http://handle.itu.int/11.1002/1000/356) | | 7.0 | ITU-T Z.100 | 2007-11-13 | 17 | [11.1002/1000/9262](http://handle.itu.int/11.1002/1000/9262) | | 8.0 | ITU-T Z.100 | 2011-12-22 | 17 | [11.1002/1000/11387](http://handle.itu.int/11.1002/1000/11387) | | 8.1 | ITU-T Z.100 Annex F1 | 2000-11-24 | 10 | [11.1002/1000/5239](http://handle.itu.int/11.1002/1000/5239) | | 8.2 | ITU-T Z.100 Annex F2 | 2000-11-24 | 10 | [11.1002/1000/5576](http://handle.itu.int/11.1002/1000/5576) | | 8.3 | ITU-T Z.100 Annex F3 | 2000-11-24 | 10 | [11.1002/1000/5577](http://handle.itu.int/11.1002/1000/5577) | | 8.4 | ITU-T Z.100 Annex F1 | 2015-01-13 | 17 | [11.1002/1000/12354](http://handle.itu.int/11.1002/1000/12354) | | 8.5 | ITU-T Z.100 Annex F2 | 2015-01-13 | 17 | [11.1002/1000/12355](http://handle.itu.int/11.1002/1000/12355) | | 8.6 | ITU-T Z.100 Annex F3 | 2015-01-13 | 17 | [11.1002/1000/12356](http://handle.itu.int/11.1002/1000/12356) | | 9.0 | ITU-T Z.100 | 2016-04-29 | 17 | [11.1002/1000/12846](http://handle.itu.int/11.1002/1000/12846) | | 9.1 | ITU-T Z.100 Annex F1 | 2016-10-29 | 17 | [11.1002/1000/13040](http://handle.itu.int/11.1002/1000/13040) | | 9.2 | ITU-T Z.100 Annex F2 | 2016-10-29 | 17 | [11.1002/1000/13041](http://handle.itu.int/11.1002/1000/13041) | | 9.3 | ITU-T Z.100 Annex F3 | 2016-10-29 | 17 | [11.1002/1000/13042](http://handle.itu.int/11.1002/1000/13042) | |

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

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

© ITU 2019

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

**Table of Contents**

Page

Annex F2 – SDL‑2010 formal definition: Static semantics 1

F2.1 General information for the static semantics 1

F2.2 Static semantics 2

F2.3 Transformation of SDL‑2010 shorthands 303

Recommendation ITU-T Z.100

Specification and Description language – Overview of SDL‑2010

Annex F2  
  
SDL‑2010 formal definition: Static semantics

## F2.1 General information for the static semantics

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

### F2.1.1 Definitions used from Annex F1

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

The keywords case, choose, controlled, else, elseif, endcase, endchoose, endif, endlet, if, initially, let, monitored, of, otherwise, provided, then.

The domains X, Boolean, Nat, Token, DefinitionAS1 and DefinitionAS0.

The functions chr, empty, false, head, isAncestorAS0, isAncestorAS1, isSameNode0, isSameNode1, last, length, num, parentAS0, parentAS0ofKind, parentAS1, parentAS1ofKind, replaceInSyntaxTree0, replaceOnceInSyntaxTree0, substring, tail, take, toSet, true and undefined.

The operation symbols \*, +, -set, -seq, =, ≠, ∧, ∨, ⇒, ⇔, ¬, ∃, ∀, >, ≥, <, ≤, +, -, \*, /, in, ×, ⁀, ∪, ∩, \, ∈, ∉, ⊆, ⊂, | |, **U**, ∅, mk-, s-, s1-, s2-.

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

The domain Token is used in the definition of the AS0 constructors for <name>, <quoted operation name>, <character string>, <hex string>, <bit string> and the AS1 constructor Name. In the formal definition it is assumed the unique Token element values can be represented by character strings such as "Predefined", so that the <name> for the predefined package is mk-<name>("Predefined") and the Name for the predefined package is mk-Name("Predefined"). In most cases, the Token element value is the same for the <name> and the corresponding Name. The construct <name>(x) is a <name> where <name>.s-Token is represented by the sequence of characters for the actual name x: letters or digits or underlines for <name> in clause 6.1 of [ITU‑T Z.101]; decimal digits for <integer name> in clause 6.1 of [ITU‑T Z.101]; decimal digits or <full stop> or e or E or <hyphen> or <plus sign> for <real name> in clause 6.1 of [ITU‑T Z.101]; and for a <string name> (<character string>, <hex string> and <bit string> in clause 6.1 of [ITU‑T Z.101]) the sequence of characters specified including enclosing and enclosed apostrophes. One case where the Token value is not the same for the <name> and the corresponding Name is for operator names (including literal and quoted operation names) where the Token value for the Name depends on the signature of the operation and data type in which it is defined (see clause F2.2.9.5 Operations below).

### F2.1.3 Status of Annex F2 (this annex)

There have been a significant number of changes since the (10/2016) edition, but there remains some issues to be resolved for full alignment with the text of SDL‑2010 in the main body of [ITU‑T Z.100], [ITU‑T Z.101], [ITU‑T Z.102], [ITU‑T Z.103], [ITU‑T Z.104], [ITU‑T Z.105], [ITU‑T Z.106], and [ITU‑T Z.107]. These issues are identified by the phrase "Further study needed …" in the text.

## F2.2 Static semantics

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

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

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

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

∀d, d*1* ∈ EntityDefinition1: d.entityKind1= d1.entityKind1 ∧ d ≠ d1⇒ d.identifier1 ≠ d1.identifier1

where:

*d* and *d1* represent any two abstract syntax tree nodes belonging to the set EntityDefinition1.

d.entityKind1 and d1.entityKind1 get the entity kinds of the corresponding syntax constructs.

d.identifier1 andd1.identifier1 get the full identifiers of the corresponding syntax constructs.

### F2.2.1 General definitions

#### F2.2.1.1 Division of text

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

Abstract syntax

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

Conditions on abstract syntax

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

Concrete syntax

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

Conditions on concrete syntax

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

Transformations

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

Mapping to abstract syntax

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

Auxiliary functions

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

#### F2.2.1.2 Abstraction of the concrete grammar (AS0)

For the sake of the definition of the static semantics rules, a special format of the concrete grammar is used. This special format is called abstract syntax level 0 (AS0). It is an abstraction of the concrete grammar of the text-only phrase representation (SDL‑PR), where all the unnecessary grammar items such as separators and terminal keywords are omitted. Any concrete graphical representation (SDL‑GR) graphical constructs in a description are converted to the equivalent SDL‑PR: in the conversion annotation (comments, notes and other items such as <create line area> and <gate property area>) and underlines followed by spaces are removed; each text extension is made part of the text it extends; and each multiple page diagram is reduced to one page. Formal rewrite rules for this conversion of SDL‑GR graphical constructs to SDL‑PR is an issue for further study, but most of the syntax rules of SDL-GR are textual rules that are also part of SDL‑PR and therefore require no conversion.

AS0 is in principle generated by a simple parsing algorithm from the concrete grammar of SDL-PR, and the SDL-PR syntax rule names are preserved in AS0. As this algorithm is an item for further study not specified here, the assertion that AS0 is a valid abstraction of SDL-PR is true has been checked by comparing the SDL-PR concrete syntax and AS0.

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

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

<data type name>

the following sections will use

<data type<name>

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

#### F2.2.1.3 Static conditions on the concrete syntax (AS0)

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 "=2=>", i.e., they are checked before any transformations except macro expansion and conversion to AS0.

#### F2.2.1.4 Transformation rules

Transformations and redefinition of functions on AS0 are represented by rewrite rules. The syntax for rewrite rules is:

<rewrite rule> ::=

<pattern> [ <provision> ] "=" [ <integer> ] "=>" <expression> { and <dependent transformation> }\*

<dependent transformation> ::=

<pattern> [ <provision> ] { "=>" } <expression>

<provision> ::=

provided <Boolean<expression>

The pattern (as well as the expression) is written using the syntax for ASM transition rules defined in Part 1 of Annex F.

The pattern can be a sequence, such as " < a = <textual typebased agent definition> >" in which case the expression is a replacement sequence. The non-terminal constructor names shall all match a non-terminal in the AS0 syntax. A variable is not allowed to appear more than once on the left hand side. Free variable names that appear on the right hand side shall also appear on the left hand side. Furthermore, the pattern and expression patterns shall be correctly typed and be of the same type.

The pattern can be a function, such as "implicitName" in which case the expression is a replacement function.

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

choose v:DefinitionAS0

case v

| Pattern then

let e = *CreateExpr*(Expression) in

replaceInSyntaxTree0(v, e, v.parentAS0)

endlet

endcase

endchoose

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 replaceInSyntaxTree0 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.

A transformation rule can be enclosed in an ASM let constructor.

A rule **let** *e* = expression **in**Pattern =i=> Replacement **endlet** is equivalent to an ASM rule of the form

**let** *e* = expression in

choose v:DefinitionAS0 in

case v

| Pattern then

let e = *CreateExpr*(*Replacement*) in

replaceInSyntaxTree0(v, e, v.parentAS0)

endlet

endcase

endchoose

endlet

Dependent transformation rules have a similar semantics. They are interpreted together with their main rule for each matching dependent pattern while the provisions of both the main rule and the dependent transformation are met.

A rule **let** *e* = expression **in**Pattern =i=> Replacement **and** Pattern1 => Replacement1 **endlet** is equivalent to an ASM rule of the form

**let** *e* = expression

**do** **in-parallel**

choose v:DefinitionAS0 in

case v

| Pattern then

let e = *CreateExpr*(*Replacement*) in

replaceInSyntaxTree0(v, e, v.parentAS0)

endlet

endcase

endchoose,

choose v:DefinitionAS0 in

case v

| Pattern1 then

let e = *CreateExpr*(*Replacement1*) in

replaceInSyntaxTree0(v, e, v.parentAS0)

endlet

endcase

endchoose

**enddo**

endlet

The integer in a rewrite rule means the transformation step this rule belongs to. The steps are described in clause F2.3. In the case of the integer being omitted, the rewrite rule can be applied in any transformation step.

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

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

monitored newName: <name>→ <name>

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

NOTE – *newName* is usually used without an argument, which has the same meaning as *newName*(*undefined*).

#### F2.2.1.5 Mapping rules

The mapping rules introduce a function.

Mapping: DefinitionAS0 → DefinitionAS1

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

Mapping(a: DefinitionAS0): DefinitionAS1 =def

case a of

This way the mapping function is defined step by step in the appropriate places in the **Mapping** sections. Each alternative of the mapping will thus be preceded by a bar ("|"), because it is one alternative of the Mapping function description. The bar ("|") is followed by a case selector terminated by the keyword then followed by the mapping for that case. Sometimes the Mapping function is called with a DefinitionAS0 parameter that is undefined, in which case the result is also undefined.

Consider the following as AS0 syntax:

<definition x> :: <y> <zlist>

Typically this would have a Mapping starting with the case selector, such as:

| <definition x> (yvalue, zl) then

If this Mapping were a list of the component mappings, this would be:

| <definition x> (yvalue, zl) then  
 < Mapping(yvalue), Mapping(zl) >

Where the zlist value is not used in the mapping, this would be

| <definition x>(yvalue, \*) then  
 < Mapping(yvalue) >

Or alternatively (with the same meaning)

| dx = <definition x> then  
 < Mapping(dx.s-<y>) >

If the result of mapping has only one component, the list brackets "<" and ">" are optional, so that the mapping of *dx* above could also be defined by:

| dx = <definition x> then  
 Mapping(dx.s-<y>)

Where a syntax item is a list, for example in:

<zlist> :: <z>\*

If the result of mapping needs to be a list, this could be (without outermost list brackets):

| <zlist> (zitems) then  
 < Mapping(*zitems*[*i*]) | i ∈ 1.. *zitems*.*length* >

If the result of the mapping needs to be a set, this could be:

| <zlist> (zitems) then  
 { Mapping(*zitems*[*i*]) | i ∈ 1.. *zitems*.*length* }

There is no general rule for the mapping of a list component (such as zitems) of a selector (such as <zlist>), so that the mapping has to be given explicitly in each case.

#### F2.2.1.6 Predefinition

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

##### F2.2.1.6.1 General functions

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

bigSeq(s: (X\*)\*): X\* =def

if s = empty then empty else s.head ⁀ bigSeq(s.tail) endif

##### F2.2.1.6.2 Domain definitions for AS1

AgentKind1: the set of agent kinds in AS1.

AgentKind1=def {system, block, process}

EntityDefinition1: the union of all the entity definitions in AS1.

EntityDefinition1=def Package-definition

∪ Agent-definition

∪ Agent-type-definition

∪ Procedure-definition

∪ Composite-state-type-definition

∪ Channel-definition

∪ Gate-definition

∪ Signal-definition

∪ Timer-definition

∪ Variable-definition

∪ Data-type-definition

∪ State-node

∪ Syntype-definition

∪ Literal-signature

∪ Operation-signature

EntityKind1: the set of all the entity kinds in AS1.

EntityKind1 =def { agent, agent type, channel, exception, gate, literal, operation, package,

procedure, signal, sort, state, state type, timer, variable }

ScopeUnit1: the union of all the scope units in AS1.

ScopeUnit1=def Package-definition

∪ Agent-definition

∪ Agent-type-definition

∪ Procedure-definition

∪ Signal-definition

∪ Composite-state-type-definition

∪ Data-type-definition

∪ State-node

∪ Compound-node

Signal1=def {id∈Identifier: id.idKind1∈{signal, timer}}

TypeDefinition1=defAgent-type-definition∪Procedure-definition∪Composite-state-type-definition∪

Data-type-definition

Value1=def Literal-signature

##### F2.2.1.6.3 Domain definitions for AS0

Binding0 =def <name> × EntityDefinition0

BindingList0 =def Binding0\*

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

EntityDefinition0 =def

<block definition> ∪

<channel definition> ∪

<composite state definition> ∪

<literal signature> ∪

<operation definition> ∪

<operation signature> ∪

<package definition> ∪

<parameters of sort> ∪

<process definition> ∪

<remote variables of sort> ∪

<signal list definition> ∪

<synonym definition> ∪

<syntype definition> ∪

<system definition> ∪

<textual gate definition> ∪

<textual typebased agent definition> ∪

<textual typebased state partition def> ∪

<timer definition item> ∪

<variables of sort> ∪

FormalContextParameter0 ∪

PredefinedDefinition0 ∪

TypeDefinition0

EntityKind0 =def {agent, agent type, block, block type, channel, exception, gate,

interface, literal, method, operator, package, procedure,

process, process type, remote procedure, remote variable,

signal, signallist, sort, state, state type, synonym, system,

system type, timer, type, variable}

The domain Expression0 is the union of constructor rule alternatives of <expression0>

Expression0 =def

<binary expression> ∪

<operand5> ∪

<equality expression> ∪

<create body> ∪

<procedure call body> ∪

<remote procedure call body> ∪

<decoding expression> ∪

<encoding expression>

FormalContextParameter0 =def

<agent context parameter> ∪

<agent type context parameter> ∪

<compositestate type context parameter> ∪

<gate context parameter> ∪

<interface context parameter name> ∪

<procedure context parameter> ∪

<remote procedure context parameter> ∪

<remotevariable contextparameter names> ∪

<signal context parameter name> ∪

<sort context parameter> ∪

<synonym context parameter name> ∪

<timer context parameter name> ∪

<variable context parameter names>

IntToken0=def (SingleDigit0 ∪ { it + d | it ∈ IntToken0 : d ∈ SingleDigit0 }) ⊂ Token

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

PredefinedDefinition0=defPredefinedLiteral0∪PredefinedOperation0∪PredefinedSort0

PredefinedLiteral0=def represents all the literals defined within the package Predefined.

PredefinedOperation0=def represents all the operations defined within the package Predefined.

PredefinedSort0=def represents all the sorts defined within the package Predefined.

Reference0 =def <package reference> ∪ <block reference> ∪ <process reference> ∪

<composite state reference> ∪ <operation reference> ∪ TypeReference0

ScopeUnit0 =def

<agent definition> ∪

<composite state definition> ∪

<compound statement> ∪

<operation definition> ∪

<package definition> ∪

<signal context parameter name> ∪

<sort context parameter> ∪

TypeDefinition0

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

SingleDigit0=def {"0", "1", "2", "3", "4", "5", "6", "7", "8", "9"} ⊂ Token

StatePart0 =def

<connect part> ∪

<continuous signal> ∪

<input part> ∪

<priority input> ∪

<save part> ∪

<spontaneous transition> ∪

<state timer part>

The domain Symbol0 is used in the function isDefinedSym0 to determine if the given symbol is defined for the range condition of a specified syntype.

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

TypeDefinition0 =def

<block type definition> ∪

<composite state type definition> ∪

<data type definition> ∪

<interface definition> ∪

<procedure definition> ∪

<process type definition> ∪

<signal definition> ∪

<system type definition>

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

<procedure reference>

##### F2.2.1.6.4 Function definitions on AS1

The function agentInstanceId1 returns the fully qualified Agent-identifier of an agent instance set from an Agent-instance list.

agentInstanceId1(ail:Agent-instance\*, pl: Path-item\*): Agent-identifier =def

let sys=take({s ∈ Sdl-specification}) in

if ail = empty then

mk-Identifier(mk-Qualifier(empty), sys.s-Agent-definition.s-Agent-name)

elseif pl = empty then

agentInstanceId1(ail, mk-Agent-type-qualifier(sys.s-Agent-definition.s-Agent-name))

elseif ail.length = 1

mk-Identifier(mk-Qualifier(pl), ail[1].s-Agent-name)

else

agentInstanceId1(ail.tail, pl ⁀ mk-Agent-type-qualifier(ail.head.s-Agent-name))

endif

endlet

The functionagentInstanceSort1 returns the pid sort of the interface defined in the context of the agent that has the same name as the agent. Note that no conversion is needed between the Agent-identifier returned by agentInstanceId1 for the Sort-reference-identifier because both these are equivalent to Identifier.

agentInstanceSort1(aiv:Agent-instance-pid-value): Sort-reference-identifier =def

agentInstanceId1(aiv.s-Agent-instance-seq, empty)

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

agentKind1(d: Agent-definition∪Agent-type-definition):AgentKind1=def

if d∈Agent-type-definition then d.s-Agent-kind

else

let td=getEntityDefinition1(d.s-Agent-type-identifier, agent type) in

td.s-Agent-kind

endlet

endif

The functiondecodeResult1 returns the static sort of decoding an expression according to the identified encoding rules. Each of the three elements of Encoding-rules is equivalent to Identifier; therefore s3-Identifier is used to select the Decode-procedure-identifier of er.

decodeResult1(de:Decoding-expression): Sort-reference-identifier =def

let ep = de.s-Encoding-path in

let er =

if ep.s*-*Encoding-rules = undefined

then getEntityDefinition1(ep.s-Gate-identifier, gate).s*-*Encoding-rules

else ep.s*-*Encoding-rules

endif

in

getEntityDefinition1(er.s3-Identifier, procedure).s*-*Result.s*-*Sort-reference-identifier

endlet

endlet

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

derivedDataType1(d: Syntype-definition∪Data-type-definition): Data-type-definition =def

if d∈Data-type-definition then d

else getEntityDefinition1(d.s-Parent-sort-identifier, sort).derivedDataType1

The functionencodeResult1 returns the static sort of encoding an expression according to the identified encoding rules. Each of the three elements of Encoding-rules is equivalent to Identifier, therefore s1-Identifier is used to select the Rules-identifier of er and s2-Identifier is used to select the Encode-procedure-identifier of er.

encodeResult1(ee:Encoding-expression): Sort-reference-identifier =def

let ep = ee.s-Encoding-path in

let er =

if ep.s*-*Encoding-rules = undefined

then getEntityDefinition1(ep.s-Gate-identifier, gate).s*-*Encoding-rules

else ep.s*-*Encoding-rules

endif

in

let nm = getEntityDefinition1(er.s1-Identifier, literal).s*-*Identifier.s*-*Name.s*-*Token in

case nm of

| "text" then predefinedId1("Charstring")

| "BER" then predefinedId1("Octetstring")

| "CER" then predefinedId1("Octetstring")

| "DER" then predefinedId1("Octetstring")

| "APER" then predefinedId1("Octetstring")

| "UPER" then predefinedId1("Bitstring")

| "CAPER" then predefinedId1("Octetstring")

| "CUPER" then predefinedId1("Bitstring")

| "BXER" then predefinedId1("Charstring")

| "CXER" then predefinedId1("Charstring")

| "EXER" then predefinedId1("Charstring;

| "OER" then predefinedId1("Octetstring")

otherwise

getEntityDefinition1(er.s2-Identifier, procedure).s*-*Result.s*-*Sort-reference-identifier

endcase

endlet

endlet

endlet

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

entityKind1(d: EntityDefinition1): EntityKind1 =def

case d of

| Package-definition then package

| Agent-definition then agent

| Agent-type-definition then agent type

| Procedure-definition then procedure

| State-node then state

| Composite-state-type-definition then state type

| Channel-definition then channel

| Gate-definition then gate

| Signal-definition then signal

| Timer-definition then timer

| Variable-definition then variable

| Data-type-definition then sort

| Syntype-definition then sort

| Interface-definition then sort

| Literal-signature then literal

| Operation-signature then operation

otherwise undefined

endcase

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

entityName1(d: EntityDefinition1): Name =def

case d of

| Package-definition then d.s-Package-name

| Agent-definition then d.s-Agent-name

| Agent-type-definition then d.s-Agent-type-name

| Procedure-definition then d.s-Procedure-name

| State-node then d.s-State-name

| Composite-state-type-definition then d.s-State-type-name

| Channel-definition then d.s-Channel-name

| Gate-definition then d.s-Gate-name

| Signal-definition then d.s-Signal-name

| Timer-definition then d.s-Timer-name

| Variable-definition then d.s-Variable-name

| Value-data-type-definition then d.s-Sort

| Syntype-definition then d.s-Syntype-name

| Interface-definition then d.s-Sort

| Literal-signature then d.s-Literal-name

| Operation-signature then d.s-Operation-name

otherwise undefined

endcase

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

fullQualifier1(d: EntityDefinition1): Qualifier =def

**let** su= parentAS1ofKind(d, ScopeUnit1) **in**

**if** su = undefined **then** empty

**elseif** d.entityKind1∈{operation, literal}∧su ∈ Data-type-definition then su.fullQualifier1

**else** su.fullQualifier1⁀mk-Qualifier(su.entityKind1, su.entityName1)

**endif**

**endlet**

The function getEntityDefinition1 gets the entity definition for an identifier and entity kind.

getEntityDefinition1(id: Identifier, k: EntityKind1): EntityDefinition1=def

take({d∈EntityDefinition1: d.identifier1 = id ∧ d.entityKind1 = k∧

(d.entityKind1 = operation ⇒

isActualAndFormalParameterMatched1(id.actualParameterListOfOpId1,

d.formalParameterSortList1))})

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

identifier1(d: EntityDefinition1):Identifier =def

mk-Identifier(d. fullQualifier1, d.entityName1)

The function idKind1 is used to return the entity kind for an identifier in AS1 based on syntactic context.

idKind1(id:Identifier):EntityKind1=def

case id.parentAS1 of

| Create-request-node∪Signal-destination then agent

| Agent-type-definition∪Agent-definition then agent type

| Procedure-definition∪Call-node∪Value-returning-call-node then procedure

| Gate-definition∪Output-node∪Save-signalset then signal

| Channel-path then

if id in id.parentAS1.s-Identifier-seq then signal

else gate

endif

| Data-type-definition∪Parameter∪Result∪Signal-definition∪Timer-definition∪

Formal-argument∪Variable-definition∪Any-expression then sort

| Set-node∪Reset-node∪Timer-active-expression then timer

| Originating-gate∪Destination-gate then gate

| Composite-state-type-definition∪State-machine∪State-node∪State-partition then state type

| Literal then literal

| Open-range∪Operation-application then operation

| Input-node then

if id in id.parentAS1.s-Variable-identifier-seq then variable

else signal

endif

| Variable-access ∪ Assignment then variable

| Direct-via then

if getEntityDefinition1(id, channel) ≠ undefined then channel

else gate

endif

endcase

The function isCompatibleTo1 determines if one Sort-reference-identifieris compatible to the other.

isCompatibleTo1(id1: Sort-reference-identifier, id2: Sort-reference-identifier): Boolean=def

let d1 = getEntityDefinition1(id1, sort) in

let d2 = getEntityDefinition1(id2, sort) in

d1=d2 ∨ isSuperType1(d2, d1)

endlet

endlet

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

isDirectSuperType1(d: EntityDefinition1, d1: EntityDefinition1): Boolean=def

case d1 of

| Agent-type-definition then

d ∈Agent-type-definition ∧ d = getEntityDefinition1(d1.s-Agent-type-identifier, agent type)

| Procedure-definition then

d∈Procedure-definition ∧ d = getEntityDefinition1(d1.s-Procedure-identifier, procedure)

| Composite-state-type-definition then

d∈Composite-state-type-definition ∧

d = getEntityDefinition1(d1.s-Composite-state-type-identifier, state type)

| Value-data-type-definition then

d∈Value-data-type-definition ∧ d = getEntityDefinition1(d1.s-Data-type-identifier, sort)

| Interface-definition then

d∈Interface-definition ∧

(∃dataId∈Data-type-identifier: dataId .parentAS1= d1 ∧ d = getEntityDefinition1(dataId, sort))

| Syntype-definition then

isDirectSuperType1(d,d1.derivedDataType1)

otherwise false

endcase

The function isInOpenRange1 determines if the given value is in the given open range.

isInOpenRange1(v1: Value1, r: Open-range): Boolean =def

let operator = r.s-Operation-identifier.s-Name in

let vrc = r.s-Constant-expression.value1 in

let v= v1.value1 in

case operator of

| "=" then v= vrc

| "/=" then ¬ (v= vrc)

| "<=" then v≤ vrc

| ">=" then v≥ vrc

| ">" then v> vrc

| "<" then v< vrc

endcase

endlet

endlet

endlet

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

isSuperType1(d1: EntityDefinition1, d2: EntityDefinition1): Boolean =def

isDirectSuperType1(d1 d2)∨∃ d3∈EntityDefinition1: isSuperType1 (d1 d3)∧isSuperType1 (d3, d2)

The function predefinedId1 produces the Identifier of package Predefined with a specified Name given as a character string (to represent a Token), for example predefinedId1("Boolean").

predefinedId1(nm: Token): Identifier =def

qualifiedId1(mk-Qualifier(<mk-Package-qualifier(mk-Name("Predefined"))>), nm)

The function qualifiedId1 produces the Identifier with the specified Qualifier and a specified Name given as a character string (to represent a Token).

qualifiedId1(ql: Qualifier, nm: Token): Identifier =def

mk-Identifier(ql, mk-Name(nm))

range1 returns the set of elements that satisfy the condition*.*

range1(r: Range-condition): Value1-**set** =def

{ v∈Value1:

∃ci ∈ r.**s**-Condition-item:

((ci ∈ Open-range) ∧ isInOpenRange1(v, ci)) ∨

((ci∈Closed-range) ∧

(∃s1, s2∈ ci.s-Open-range: (s1 ≠ s2) ∧ isInOpenRange1(v, s1)∧ isInOpenRange1(v, s2)))}

The functionstaticSort1 returns the static sort of *e*.

staticSort1(e:Expression ): Sort-reference-identifier =def

case e of

| Active-agents-expression then predefinedId1("Natural")

| Agent-instance-pid-value then agentInstanceSort1(e)

| Any-expression then e.s-Sort-reference-identifier

| Conditional-expression then staticSort1(e.s-Consequence-expression)

| Decoding-expression then decodeResult1(e)

| Encoding-expression then encodeResult1(e)

| Equality-expression then predefinedId1("Boolean")

| Literal then getEntityDefinition1(e.s-Literal-identifier, literal).s-Result

| Now-expression then predefinedId1("Time")

| Offspring-expression then predefinedId1("Pid")

| Operation-application then getEntityDefinition1(e.s-Operation-identifier, operation).s-Result

| Parent-expression then predefinedId1("Pid")

| Range-check-expression then predefinedId1("Boolean")

| Sender-expression then predefinedId1("Pid")

| Self-expression then

let at = parentAS1ofKind(e, Agent-type-definition) in

qualifiedId1(at.s-Qualifier, at.s-Agent-type-name.s-Token)

// Interface

endlet

| State-expression then predefinedId1("Charstring")

| Timer-active-expression then predefinedId1("Boolean")

| Timer-remaining-duration then predefinedId1("Duration")

| Type-check-expression then predefinedId1("Boolean")

| Type-coercion then e.s-Sort-reference-identifier

| Value-returning-call-node then getEntityDefinition1(e.s-Procedure-identifier, procedure).s*-*Result

| Variable-identifier then getEntityDefinition1(e, variable).s-Sort-reference-identifier

// Variable-access

endcase

##### F2.2.1.6.5 Function definitions on AS0

The function binaryOpName0gets the name of an operation used in a <binary expression>.

binaryOpName0(be: <binary expression> ): <name> =def

mk-<name>(

case be.s-<infix operation name>.s-implicit of

| <implies sign> then "=>"

| or then "or"

| xor then "xor"

| and then "and"

| <greater than sign> then ">"

| <greater than or equals sign> then ">="

| <less than sign> then "<"

| <less than or equals sign> then "<="

| in then "in"

| <plus sign> then "+"

| <hyphen> then "-"

| <concatenation sign> then "//"

| <asterisk> then "\*"

| <solidus> then "/"

| mod then "mod"

| rem then "rem"

endcase

)

The function contextQualifier0gets the actual <qualifier> specified in an entity definition or a reference. In most cases for an entity definition the <qualifier> in the heading is optional, because the qualifier of the entity identifier is determined by the nearest enclosing scope unit in which the entity definition occurs. The <qualifier> is needed for any <referenced definition> (except outermost packages) if the entity kind with the entity name is ambiguous. It is only allowed to have a <qualifier> in the heading of an entity definition that is a <referenced definition>.

contextQualifier0(d: EntityDefinition0 ∪ Reference0 ): <qualifier> =def

case d of

| <block definition> then d.s-<block heading>.s-<qualifier>

| <block type definition> then d.s-<block type heading>.s-<qualifier>

| <block type reference> then d.s-<identifier>.s-<qualifier>

| <composite state definition> then d.s-<composite state heading>.s-<qualifier>

| <composite state type definition> then

if d.s-implicit ∈ <composite state type graph>

then d.s-implicit.s-<composite state type heading>.s-<qualifier>

else d.s-implicit.s-<state aggregation type heading>.s-<qualifier>

endif

| <composite state type reference> then d.s-<identifier>.s-<qualifier>

| <internal procedure definition> then d.s-<procedure heading>.s-<qualifier>

| <operation definition> then d.s-<operation heading>.s-<qualifier>

| <package definition> then d.s-<package heading>.s-<qualifier>

| <package reference> then d.s-<identifier>.s-<qualifier>

| <procedure reference> then d.s-<procedure reference heading>.s-<qualifier>

| <process definition> then d.s-<process heading>.s-<qualifier>

| <process type definition> then d.s-<process type heading>.s-<qualifier>

| <process type reference> then d.s-<identifier>.s-<qualifier>

| <system type definition> then d.s-<system type heading>.s-<qualifier>

| <system type reference> then d.s-<identifier>.s-<qualifier>

otherwise undefined

endcase

The function entityKind0gets the kind of a given entity definition.

entityKind0(ed: EntityDefinition0): EntityKind0 =def

case ed **of**

| <agent context parameter> then ed.s-<agent kind>

| <agent type context parameter> then

if ed.s-<agent kind> = block then block type **else** process typeendif

| <block definition> then block

| <block type definition> then block type

| <channel definition> then channel

| <composite state type definition> then state type

| <composite state definition> then state

| <compositestate type context parameter> then state type

| <data type definition> then type

| <external synonym definition item> then synonym

| <gate context parameter> then gate

| <interface context parameter name> then interface

| <interface definition> then interface

| <internal synonym definition item> then synonym

| <literal signature> then literal

| <operation definition> then ed.s-<operation heading>.s-<operation kind>

| <operation signature> then if ed.parentAS0 ∈<operator list> then operator **else** methodendif

| <package definition> then package

| <procedure context parameter> then procedure

| <procedure definition> then procedure

| <process definition> then process

| <process type definition> then process type

| <remote procedure context parameter> then remoteprocedure

| <remote procedure definition> then remoteprocedure

|<remote variables of sort> then remote variable

|<remotevariable contextparameter names> then remote variable

| <signal context parameter name> then signal

| <signal definition> then signal

| <signal list definition> then signallist

| <sort context parameter> then type

| <synonym context parameter name> then synonym

| <syntype definition> then type

| <system definition> then system

| <system type definition> then system type

| <textual gate definition> then gate

| <textual typebased block definition> then block

| <textual typebased process definition> then process

| <textual typebased state partition def> then state

| <textual typebased system definition> then system

| <timer context parameter name> then timer

| <timer definition item> then timer

|<variable context parameter names> then variable

|<variables of sort> then variable

otherwise undefined

endcase

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

entityName0(ed: EntityDefinition0): <name> =def

case ed of

| <block definition> then ed.s-<block heading>.s-<name>

| <block type definition> then ed.s-<block type heading>.s-<name>

| <composite state type definition> then ed.s-<composite state type heading>.s-<name>

| <composite state definition> then ed.s-<composite state heading>.s-<name>

| <data type definition> then ed.s-<data type heading>.s-<name>

| <external procedure definition> then ed.s-<procedure heading>.s-<name>

| <interface definition> then ed.s-<interface heading>.s-<name>

| <internal procedure definition> then ed.s-<procedure heading>.s-<name>

| <literal signature> then

if ed∈<literal name> then ed

elseif ed∈<named number> then ed.s-<literal name>

else undefined

endif

| <operation definition> then ed.s-<operation heading>.s-<name>

| <operation signature> then

if ed.s-<operation name> ∈ <operation<name> then ed.s-<operation name>

else mk-<name>(ed.s-<quoted operation name>)

endif

| <package definition> then ed.s-<package heading>.s-<name>

| <process definition> then ed.s-<process heading>.s-<name>

| <process type definition> then ed.s-<process type heading>.s-<name>

| <signal definition> then ed.s-<name>

| <signal list definition> then ed.s-<name>

| <system definition> then ed.s-<system heading>.s-<name>

| <system type definition> then ed.s-<system type heading>.s-<name>

| <syntype definition> then

let s=ed.s-implicit in

if s∈<syntype definition syntype> then s.s-<syntype<name>

else s.s-<data type heading>.s-<data type<name>

endif

endlet

| <textual gate definition> then ed.s-<name>

| <textual typebased block definition> then ed.s-<name>

| <textual typebased process definition> then ed.s-<name>

| <textual typebased state partition def> then ed.s-<name>

| <textual typebased system definition> then ed.s-<name>

| <timer definition item> then ed.s-<name>

| <variables of sort> then ed.s-<name>.head

| FormalContextParameter0 then ed.s-<name>

endcase

The function getEntities0 is used to get the entity definitions of the enclosing scope unit.

getEntities0(n: DefinitionAS0): EntityDefinition0\* =def

if n = undefined then undefined else

case n of

| <system type definition> ∪ <block type definition> ∪ <process type definition>

then n.s-<agent structure>.s-<entity in agent>-seq

| <agent definition> then n.s-<agent structure>.s-<entity in agent>-seq

| <composite state definition>(<composite state graph>(\*,\*,<composite state structure>(\*, entities,\*)))

then entities

| <composite state definition>(<state aggregation>(\*,\*,<aggregation structure>(entities,\*)))

then entities

| <composite state type definition>(

<composite state type graph>(\*,\*,<composite state structure>(\*, entities,\*),\*))

then entities

| <composite state type definition>(

<state aggregation type>(\*,\*,<aggregation structure>(entities,\*),\*))

then entities

| <data type definition>(\*, \*, \*,\*, <data type definition body>(entities, \*, \*,\*)) then entities

| <internal procedure definition>(\*, \*, entities, \*) then entities

| <package definition>(\*, \*, entities) then entities

otherwise n.parentAS0.getEntities0

endcase

endif

The function isConstantExpression0 is true if and only if the given expression does not contain an <active primary> or a <value returning procedure call>.

isConstantExpression0 (expr: Expression0): Boolean =def

¬(∃ subexpr ∈ <variable access> ∪ <value returning procedure call> : isAncestorAS0(expr,subexpr)

The function kind0gets the name of a given entity definition or reference.

kind0(d: EntityDefinition0∪ Reference0 ): EntityKind0 =def

case ed **of**

| Reference0 then d.referenceKind0

| EntityDefinition0 then d.entityKind0

endcase

The function name0gets the name of a given entity or reference.

name0(d: EntityDefinition0∪ Reference0 ): <name> =def

case d of

| Reference0 then d.referenceName0

| EntityDefinition0 then d.entityName0

otherwise undefined

endcase

The function natToIntToken produces a Token corresponding to the given Natural number.

natToIntToken(n: Nat): IntToken0 =def

case n of

| 0 then "0"

| 1 then "1"

| 2 then "2"

| 3 then "3"

| 4 then "4"

| 5 then "5"

| 6 then "6"

| 7 then "7"

| 8 then "8"

| 9 then "9"

otherwise

take({t ∈ Token :

tens ∈ Nat ∧ tens > 0

∧ units ∈ Nat ∧ units < 10

∧ n = tens \* 10 + units

∧ t = if tens=0 then "" else natToIntToken(tens) endif ⁀ natToIntToken(units)

})

endcase

The function intTokenToNat produces a Nat corresponding to the given IntToken0.

intTokenToNat(t: IntToken0): Nat =def

take({n ∈ NAT : natToIntToken(n) = t})

The function predefinedId0 produces the <identifier> of package Predefined with a specified name given as a character string, for example predefinedId0("Charstring").

predefinedId0(cs: Token ): <identifier> =def

mk-<identifier>( < mk-<path item>(package, mk-<name>("Predefined"))>, mk-<name>(cs))

The function predefinedItem0 produces an <identifier> for an item defined in a data type of package Predefined with the specified data type name given as a character string, and the specified item name given as a character string, for example predefinedItem0("Boolean","true").

predefinedItem0(q: Token, n: Token): <qualifier> =def

mk-<identifier>( predefinedQual0(q), mk-<name>(n))

The function predefinedQual0 produces a <qualifier> for the context of data type defined in package Predefined with a specified data type name given as a character string, for example predefinedQual0("Integer").

predefinedQual0(cs: Token ): <qualifier> =def

< mk-<path item>(package, mk-<name>("Predefined")), mk-<path item>(sort, mk-<name>(cs)) >

The function predefinedSort0 produces a predefined <sort> of package Predefined with a specified name given as a character string, for example predefinedSort0 ("Integer").

predefinedSort0(cs: Token ): <sort> =def

mk-<sort>(mk-<type expressin>(predefinedId0(cs)))

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

referenceKind0(ref: Reference0 ): EntityKind0=def

case ref of

| <block reference> then block

| <block type reference> then block type

| <composite state reference> then state

| <composite state type reference> then state type

| <operation reference> then ref.s-<operation kind>

| <package reference> then package

| <procedure reference> then procedure

| <process reference> then process

| <process type reference> then process type

| <system type reference> then system type

otherwise undefined

endcase

The function referenceName0 gets the name of a given reference.

referenceName0(ref: Reference0 ): <name> =def

case ref of

| TypeReference0 then ref.s-<identifier>.s-<name>

| <operation reference> then ref.s-<operation heading>.s-<operation name>

otherwise ref.s-<name>

endcase

The function surroundingQualifier0 gets the qualifier to use for entities defined in the surrounding scope unit of a node in AS0.

NOTE – The function surroundingQualifier0 has the same body as the function fullQualifier0 but accepts any DefinitionAS0 item as a parameter, whereas fullQualifier0 requires an EntityDefinition0.

surroundingQualifier0(d: DefinitionAS0): <qualifier> =def

letsu = d.surroundingScopeUnit0 in

ifsu = undefined then empty

else su.fullQualifier0⁀ mk-<path item>(su.entityKind0, su.entityName0)

endlet

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

surroundingScopeUnit0(d: DefinitionAS0): ScopeUnit0 =def

ifd ∈ <referenced definition> ∧ d.parentAS0 ∈ <sdl specification> then

parentAS0ofKind(d.referencedBy0, ScopeUnit0)

else

parentAS0ofKind(d, ScopeUnit0)

endif

##### F2.2.1.6.6 Lexis

The following lexical items are used here:

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

<asterisk> :: ()

<concatenation sign> :: ()

<equals sign> :: ()

<greater than or equals sign> :: ()

<greater than sign> :: ()

<hyphen> :: ()

<implies sign> :: ()

<infix operation name> ::

<implies sign> | **or** | **xor** | **and**

| <greater than sign> | <greater than or equals sign>

| <less than sign> | <less than or equals sign> | **in**

| <plus sign> | <hyphen> | <concatenation sign> | <asterisk> | <solidus> | **mod** | **rem**

<less than or equals sign> :: ()

<less than sign> :: ()

<not equals sign> :: ()

<plus sign> :: ()

<solidus> :: ()

### F2.2.2 Visibility rules, names and identifiers

#### F2.2.2.1 Name

Abstract syntax

Name :: Token

Agent-name = Name

Agent-type-name = Name

Channel-name = Name

Compound-node-name = Name

Connector-name = Name

Data-type-name = Name

Exception-name = Name

Gate-name = Name

Interface-name = Name

Literal-name = Name

Operation-name = Name

Package-name = Name

Procedure-name = Name

Signal-name = Name

State-entry-point-name = Name

State-exit-point-name = Name

State-name = Name

State-type-name = Name

Syntype-name = Name

Timer-name = Name

Variable-name = Name

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

Concrete syntax

<name> :: Token

<quoted operation name> :: Token

<character string> :: Token

<hex string> :: Token

<bit string> :: Token

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

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

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

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

Mapping to abstract syntax

| <name>(x) then mk-Name(x)

| <quoted operation name>(x) then mk-Name(x)

| <hex string>(x) then mk-Name(x)

| <bit string>(x) then mk-Name(x)

| <character string>(x) then

let cscontext = x.parentAS0 in

case cscontext of

| <transition option> ∪ <task> ∪ <decision>

then mk-Informal-text()

| <textual answer part>

then

let q = cscontext.parentAS0.parentAS0.s-<question> in

if q.s-<operand5>.s-<primary>.s-<literal identifier> ∈ <character string>

then mk-Informal-text() // after interpretation how the system behaves is undefined

else

if (isSubSort0(predefinedId0("Charstring")), q.staticSort0)) ∨

(length(x) =1) ∧ (isSubSort0(predefinedId0("Char")) q.staticSort0)) ∨

(isSubSort0(predefinedId0("NumericString")), q.staticSort0)) ∨

(isSubSort0(predefinedId0("PrintableString")), q.staticSort0)) ∨

(isSubSort0(predefinedId0("TeletexString")), q.staticSort0)) ∨

(isSubSort0(predefinedId0("VideotexString")), q.staticSort0)) ∨

(isSubSort0(predefinedId0("VisibleString")), q.staticSort0))

then mk-Name(x)

else mk-Informal-text() // after interpretation how the system behaves is undefined

endif

endif

endlet

otherwise mk-Name(x)

endcase

endlet

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

#### F2.2.2.2 Identifier

Abstract syntax

Identifier :: Qualifier Name

Qualifier = Path-item\*

Agent-identifier = Identifier

State-identifier = Identifier

Agent-type-identifier = Identifier

Procedure-identifier = Identifier

Signal-identifier = Identifier

Data-type-identifier = Identifier

Sort-reference-identifier = Sort-identifier

| Syntype-identifier

Sort-identifier = Identifier

Syntype-identifier = Identifier

Timer-identifier = Identifier

Gate-identifier = Identifier

Exception-identifier = Identifier

Composite-state-type-identifier = Identifier

Literal-identifier = Identifier

Operation-identifier = Identifier

Variable-identifier = Identifier

Conditions on abstract syntax

∀d, d1 ∈ EntityDefinition1: d.entityKind1 = d1.entityKind1 ∧ d ≠ d1⇒ d.identifier1 ≠ d1.identifier1

Each entity of a kind shall have an Identifierdifferent from any other entity of the same kind.

NOTE - Consequently, no two definitions in the same scope unit and belonging to the same entity kind are allowed to have the same <name>, except operations defined in the same <data type definition> that differ in at least one argument <sort> or the result <sort> (in which case they have different Operation-name values).

Concrete syntax

<identifier> :: <qualifier> <name>

<qualifier> = <path item>\*

Transformations

i=<identifier> provided i ≠ fullIdentifier0(i) =14=> fullIdentifier0(i)

It is allowed to omit some of the leftmost <path item>s, or the whole <qualifier> of an <identifier> if it is possible to uniquely expand the remaining <path item>s to a full <qualifier>. See clause 6.6 of [ITU‑T Z.101].

Mapping to abstract syntax

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

Auxiliary functions

For any given identifier, return its full identifier.

fullIdentifier0(i:<identifier>): <identifier> =def i.refersto0.identifier0

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

refersto0(i:<identifier>): EntityDefinition0 =def getEntityDefinition0(i, idKind0(i))

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

identifier0(def: EntityDefinition0): <identifier> =def

mk-<identifier>(def.fullQualifier0, def.entityName0)

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

fullQualifier0(d: EntityDefinition0): <qualifier> =def

letsu = d.surroundingScopeUnit0 in

ifsu = undefined then empty

else su.fullQualifier0⁀ mk-<path item>(su.entityKind0, su.entityName0)

endlet

The function fullQualifierWithin0 is used to get the full qualifier for items defined directly within the entity definition d.

fullQualifierWithin0(d: EntityDefinition0): <qualifier> =def

d.fullQualifier0 ⁀ mk-<path item>(d.entityKind0, d.entityName0)

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

getEntityDefinition0(id:<identifier>, ek: EntityKind0): EntityDefinition0 =def

if ek∈{operator, literal, method} then

resolveByContext0(id)

else

letsu = getStartingScopeUnit0(id, id.surroundingScopeUnit0) in

if su = undefined then undefined

else resolveByContainer0(su, id, ek)

endif

endlet

endif

The function resolveByContainer0 binds an <identifier> to a definition through resolution by container.

resolveByContainer0(su: ScopeUnit0, id:<identifier>, ek:EntityKind0): EntityDefinition0 =def

let d1=bindInLocalDefinition0(su, id, ek) in

if d1≠ undefined then d1

else let d2=bindInBaseType0(su, id, ek) in

if d2≠ undefined then d2

else let d3=bindInUsedPackage0(su.s-<package use clause>-seq, id, ek) in

if d3≠ undefined then d3

else let d4=bindInLocalInterface0(su.localInterfaceDefinitionSet0, id, ek) in

if d4≠ undefined then d4

else let su1=su.surroundingScopeUnit0 in

if su1 ≠ undefined

then resolveByContainer0(su1, id, ek)

elseif ek= system ∧ (∃ sd ∈ <system definition>: sd.s-<name> = id.s-<name> ) then

take({sd ∈ <system definition>: sd.s-<name> = id.s-<name>})

elseif ek= package ∧ id.s-<qualifier> = undefined

( ∃ pd ∈ <package definition>: pd.s-<name> = id.s-<name>

∧ pd.s-<qualifier> = undefined)

then

take({pd ∈ <package definition>: pd.s-<name> = id.s-<name>

∧ pd.s-<qualifier> = undefined})

else undefined

endif endlet

endif endlet

endif endlet

endif endlet

endif

endlet

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

bindInLocalDefinition0(su: ScopeUnit0, id: <identifier>, ek: EntityKind0): EntityDefinition0=def

let d = take({d∈ EntityDefinition0:

d.surroundingScopeUnit0 = su ∧ isSameEntityName0(id.s-<name>, d)∧

isConsistentKindTo0(d.entityKind0, ek)∧isVisibleIn0(d, id.surroundingScopeUnit0)}) in

**if** d ≠ undefined then d

else let rd = take({rd ∈Reference0 : rd.surroundingScopeUnit0 = su ∧

rd. referencedDefinition0.entityName0= id.s-<name>∧

isConsistentKindTo0(rd.referencedDefinition0.entityKind0, ek) ∧

isVisibleIn0(rd, id.surroundingScopeUnit0))}) in

**if** rd ≠ undefined then rd.referencedDefinition0

else undefined

endif endlet

endif

endlet

The function bindInBaseType0 finds the entity definition corresponding to the given <identifier> in the base type of the scope unit if there is a specialization and the <identifier> is not within the specialization.

bindInBaseType0(su: ScopeUnit0, id: <identifier>, ek: EntityKind0): EntityDefinition0=def

let spec = su.specialization0 in

if (spec = undefined) ∨ isAncestorAS0(spec, id) then undefined

else resolveByContainer0(spec.s-<type expression>.baseType0, id, ek)

endif

endlet

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

bindInUsedPackage0(ucl:<package use clause>\*, id: <identifier>, ek: EntityKind0):

EntityDefinition0=def

if ucl = empty then undefined

elseif ucl.head = id.parentAS0 then

bindInUsedPackage0(ucl.tail, id, ek)

else

let d = bindInLocalDefinition0(ucl.head.usedPackage0, id, ek) in

if d ≠ undefined then d

else bindInUsedPackage0(ucl.tail, id,ek)

endif

endlet

endif

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

bindInLocalInterface0(is:<interface definition>-set, id: <identifier>, ek: EntityKind0):

EntityDefinition0 =def

if is = ∅ then undefined

else let d = is.take in

let ed = bindInLocalDefinition0(d, id, ek) in

if ed ≠ undefined then ed

else bindInLocalInterface0(is \ {d})

endif

endlet

endif

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

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

(n = d.entityName0)

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

getStartingScopeUnit0(id: <identifier>, su: ScopeUnit0): ScopeUnit0=def

if su= undefined then undefined

elseif isQualifierMatched0(id.s-<qualifier>, su) then su

else let su1= getStartingSuInUsedPackage0(id, su.usedPackageDefinitionList0) in

if su1≠ undefined then su1

else let su2= getStartingSuInInterface0(id, su.localInterfaceDefinitionSet0) in

if su2≠ undefined then su2

else getStartingScopeUnit0(id, su.surroundingScopeUnit0)

endif

endlet

endif endlet

endif

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

getStartingSuInUsedPackage0(id: <identifier>, pdl:<package definition>\*): ScopeUnit0=def

if pdl =empty then undefined

elseif isQualifierMatched0(id.s-<qualifier>, pdl.head) then pdl.head

else getStartingSuInUsedPackage0(id, pdl.tail)

endif

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

getStartingSuInInterface0(id: <identifier>, ifds:<interface definition>-set): ScopeUnit0=def

if ifds = ∅ then undefined

else let d = ifds.take in

if isQualifierMatched0(id.s-<qualifier>, d) then d

else getStartingSuInInterface0(id, ifds \ {d})

endif endlet

endif

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

isDefinedIn0(ed: EntityDefinition0, su: ScopeUnit0): Boolean=def

if (su = undefined ∨ ed =undefined)then false

else

su = ed.surroundingScopeUnit0 ∨

isVisibleInInterface0(ed, su) ∨

isVisibleInDataType0(ed, su) ∨

isVisibleThroughBaseType0(ed, su)

endif

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

isVisibleIn0(ed: EntityDefinition0, su: ScopeUnit0): Boolean=def

isDefinedIn0(ed, su) ∨

isVisibleThroughUsedPackage0(ed, su) ∨

isVisibleIn0(ed, su.surroundingScopeUnit0)

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

isVisibleInInterface0(ed: EntityDefinition0, su: ScopeUnit0): Boolean=def

( ed.surroundingScopeUnit0∈ su.localInterfaceDefinitionSet0 )

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

isVisibleInDataType0(ed: EntityDefinition0, su: ScopeUnit0): Boolean=def

(ed.surroundingScopeUnit0∈ su.localDataTypeDefinitionSet0) ∧

(ed∈<literal signature> ∪ <operation signature> ⇒ ed.isPublic0)

The function isVisibleThroughBaseType0 determines if the entity is visible through the base type of the scope unit. The entity is visible if the type is a direct sub type of the base type in which the entity is visible, and the entity is either a literal/operation (that is not private/renamed) or is a formal context parameter of the base type not bound to an actual context parameter.

isVisibleThroughBaseType0(ed: EntityDefinition0, su: ScopeUnit0): Boolean=def

let spec = su.specialization0 in

if spec = undefined then false

else (∃btd∈TypeDefinition0 : isDirectSubType0(su, btd) ∧ isVisibleIn0(ed, btd) ∧

(ed ∈ <literal signature>∪<operation signature> ⇒

¬isPrivate0(ed)∧¬ isRenamedBySpec0(ed, spec))∨

ed∈ FormalContextParameter0⇒¬isBoundToActualContextPara0(ed, spec))

endif

endlet

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

isVisibleThroughUsedPackage0(ed: EntityDefinition0, su: ScopeUnit0): Boolean =def

let ucs = su.s-<package use clause>-seq.toSet in

if ucs = ∅ then false

else (∃uc∈ucs: ed.surroundingScopeUnit0= uc.usedPackage0∧

isVisibleThroughUseClause0(ed, uc))

endif

endlet

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

isBoundToActualContextPara0 (fcp: FormalContextParameter0, spec: <specialization>):

Boolean =def

(∃acp∈<actual context parameter>:

acp.parentAS0.parentAS0 = spec∧isContextParameterCorresponded0(fcp, acp))

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

isVisibleThroughUseClause0(ed: EntityDefinition0, uc:<package use clause>): Boolean =def

let pd=uc.usedPackage0 in

let pp = pd.s-<package heading>.s-<package public> in

(pp = undefined∨

∃ds∈ <definition selection>: ds.parentAS0= pp∧

isMentionedInDefSelection0(ed, ds, pd)) ∧

(uc.s-<definition selection>-seq = empty ∨

∃ds∈<definition selection>: ds.parentAS0 = uc ∧

isMentionedInDefSelection0(ed, ds, pd))

endlet

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

isMentionedInDefSelection0

(ed: EntityDefinition0, ds:<definition selection>, pd:<package definition>):

Boolean =def

(ds.s-<name> = ed. entityName0 ∧

(ds.s-<selected entity kind> ≠undefined⇒ ds.s-<selected entity kind> = ed. entityKind0))∨

(ed.entityKind0= signal∧ ds.s-<selected entity kind> = signallist ∧

(∃sld∈<signal list definition>: sld.surroundingScopeUnit0= pd∧

sld. entityName0= ds.s-<name> ∧

(∃sigId∈ sld.signalSet0: getEntityDefinition0(sigId, signal) = ed)))

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

isConsistentKindTo0(t1: EntityKind0, t2: EntityKind0): Boolean =def

t1 = t2 ∨

(t2= agent) ∧ ((t1 = system)∨(t1 = block)∨(t1 = process)) ∨

(t2= agent type) ∧ ((t1 = system type)∨(t1 = block type)∨(t1 = process type)) ∨

(t2 = sort)∧(( t1=interface)∨( t1= type))

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

isQualifierMatched0(q1: <qualifier>, su: ScopeUnit0): Boolean =def

if q1 = undefined then true

elseif su∈<compound statement> then false

else

let q2 = su.fullQualifier0 ⁀< mk-<path item>(su.entityKind0, su.entityName0) > in

(q1.length≤ q2. length∧

(∀i∈1.. q1.length:∀j∈Nat: j = q2.length - q1. length+ i ⇒

(q1[i].s-<name> = q2[j].s-<name> ∧

(q1[i].s-<scope unit kind> ≠ undefined ⇒

q1[i].s-<scope unit kind> = q2[j].s-<scope unit kind>))))

endlet

endif

The function resolveByContext0 is used to bind all <name>s of entity kind operator, method and literal in an <expression> in an <assignment> or <decision> or <expression> to their corresponding entity definitions. The function resolveByContext0 is called only from getEntityDefinition0 with the identifier of an operator, method and literal.

resolveByContext0(id:<identifier>):EntityDefinition0=def

let bl = take(getBindingListSet0(id.contextOfIdentifier0)) in

getDefinitionInBindingList0 (id.s-<name>, bl)

endlet

The function contextOfIdentifier0 gets the context for resolving the identifier.

contextOfIdentifier0(id:<identifier>):Context0=def

if (∃exp∈<expression>: isAncestorAS0(exp, id))

then contextOfExp0(parentAS0ofKind(id, <expression>))

else undefined

endif

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

getBindingListSet0(c: Context0): BindingList0-set =def

let nameList = c.nameList0 in

let possibleBindingListSet = nameList.possibleBindingListSet0 in

let possibleResultSet = {pbl∈ possibleBindingListSet: isSatisfyStaticCondition0(pbl, c)} in

let resultSet = {r∈ possibleResultSet: ∀r1∈ possibleResultSet: r ≠ r1 ⇒

mismatchNumber0(r, c)≤ mismatchNumber0(r1, c)} in

if |resultSet| = 1 then resultSet

else ∅

endif

endlet

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

nameList0(c: Context0): <name>\*=def

case c of

|<assignment> then c.s-<variable>.nameListInVariable0⁀c.s-<expression>.nameListInExpression0

|<decision> then c.nameListInDecision0

|<expression> then c.nameListInExpression0

otherwise empty

endcase

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

nameListInExpression0(exp: <expression>):<name>\*=def

case exp of

| <binary expression> then

let mk-<binary expression>(e1, op, e2)= exp in

exp.s1-<expression>.nameListInExpression0⁀

binaryOpName0(exp) ⁀

exp.s2-<expression>.nameListInExpression0

endlet

| <create expression> then

exp.actualParameterList0.nameListInActualParameterList0

| <equality expression> then

exp.s1-<expression>.nameListInExpression0 ⁀

case exp.s-implicit of

| <equals sign> then < mk-<name>("=") >

| <not equals sign> then < mk-<name>("/=") >

endcase ⁀

exp.s2-<expression>.nameListInExpression0

| <range check expression> then exp.s-<expression>.nameListInExpression0

| <operand5> then

case exp.s-implicit of

| <hyphen> then < mk-<name>("-") >

| not then < mk-<name>("not") >

otherwise empty

endcase ⁀ exp.s-<primary>.nameListInPrimary0

| <value returning procedure call> then

exp.actualParameterList0.nameListInActualParameterList0

endcase

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

nameListInPrimary0(p: <primary>):<name>\*=def

case p of

| <conditional expression> then

p.s-<expression>.nameListInExpression0⁀

p.s-<consequence expression>.nameListInExpression0⁀

p.s-<alternative expression>.nameListInExpression0

| <expression> then p.nameListInExpression0

| <extended primary> then p.nameListInExtendedPrimary0

| <literal> then <p.s-<literal identifier>.s-<literal name>>

| <operator application> then p.nameListInOperationApplication0

otherwise: empty

endcase

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

nameListInOperationApplication0(oa: <operator application>):<name>\*=def

case oa of

| <method application> then

oa.s-<primary>.nameListInPrimary0⁀<oa.s-<operation identifier>.s-<operation name>>⁀

oa.actualParameterList0.nameListInActualParameterList0

| <operator application> then

<oa.s-<operation identifier>>⁀oa.actualParameterList0.nameListInActualParameterList0

endcase

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

nameListInExtendedPrimary0(ep:<extended primary>):<name>\*= def

case ep of

| <composite primary> then ep.s-<actual parameter>-seq.nameListInActualParameterList0

| <field primary> then ep.s-<primary>.nameListInPrimary0

| <indexed primary> then

ep.s-<primary>.nameListInPrimary0⁀

ep.s-<actual parameter>-seq.nameListInActualParameterList0

endcase

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

nameListInActualParameterList0(el: <expression>\*): <name>\*=def

if el= empty then empty

else

el.head.nameListInExpression0⁀el.tail.nameListInActualParameterList0

endif

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

nameListInVariable0(v:<variable>):<name>\*=def

if v∈<indexed variable> then

v.s-<variable>.nameListInVariable0⁀

v.s-<actual parameter>-seq.nameListInActualParameterList0

elseif v∈<field variable> then

v.s-<variable>.nameListInVariable0

else empty

endif

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

nameListInDecision0 (d: <decision>): <name>\*= def

d.s-<question>.nameListInExpression0⁀ d.rangeConditionList0.nameListInRangeConditions0

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

nameListInRangeConditions0 (rcl: <range condition>\*): <name>\*= def

if rcl = empty then empty

else rcl.head.nameListInRangeList0⁀rcl.tail.nameListInRangeConditions0

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

nameListInRangeList0 (rl: <range>\*): <name>\*= def

if rl = empty then empty

else rl.head.nameListInRange0⁀rl.tail.nameListInRangeList0

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

nameListInRange0(r: <range>): <name>\*= def

case r of

| <closed range> then

let r = mk-<closed range> (c1, c2) in

c1.nameListInExpression0⁀c2.nameListInExpression0

endlet

| <open range> then

let r = mk-<open range>(c1, n, c2) in

c1.nameListInExpression0⁀<n>⁀c2.nameListInExpression0

endlet

endcase

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

possibleBindingListSet0(n: <name>\*):BindingList0-set =def

{b∈BindingList0: b.length = n.length∧

∀i∈1..b.length:b[i].s-<name>= n[i]∧ b[i].s-EntityDefinition0∈n[i].possibleDefinitionSet0}

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

isSatisfyStaticCondition0(bl:BindingList0, c:Context0): Boolean=def

case c of

| <assignment> then isSatisfyAssignmentCondition0(bl, c)

| <decision> then isSatisfyDecisionCondition0(bl, c)

| <expression> then isSatisfyExpressionCondition0(bl, c)

otherwise false

endcase

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

isSatisfyAssignmentCondition0(bl:BindingList0, ass: <assignment>): Boolean=def

let varSort= getVariableSort0(ass.s-<variable>) in

let expSort= getStaticSort0(ass.s-<expression>, bl) in

(isSortCompatible0(varSort, expSort) ∨ isSortCompatible0(expSort, varSort))∧

(ass.s-<variable>∈<indexed variable>⇒

isSatisfyIndexVariableCondition0(bl, ass.s-<variable>)

endlet

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

isSatisfyIndexVariableCondition0(bl:BindingList0, var:<indexed variable>): Boolean=def

let apl=var.s-<actual parameter>-seq in

isSortCompatible0(getStaticSort0(apl[1],bl), getIndexSort0(getVariableSort0(var)))∧

(var.s-<variable>∈<indexed variable>⇒

isSatisfyIndexVariableCondition0(bl, var.s-<variable>))

endlet

Get the static sort of a <variable>.

getVariableSort0(var:<variable>):<sort>=def

case var of

| <identifier> then getEntityDefinition0(var, variable).parentAS1.s-<sort>

| <indexed variable> then getItemSort0(getVariableSort0(var.s-<variable>))

| <field variable> then getFieldSort0(getVariableSort0(var.s-<variable>), var.s-<field name>)

endcase

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

getItemSort0(s: <sort>):<sort>=def

let d=getEntityDefinition0(s, sort).derivedDataType0 in

if d.specialization0 = undefined then undefined

else

let btn= d.specialization0.s-<base type>.s-<name> in

if btn = "String" ∨ btn = "Vector"

then d.actualContextParameterList0[1]

elseif btn = "Array"

then d.actualContextParameterList0[2]

else undefined

endif

endlet

endif

endlet

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

getIndexSort0(s: <sort>):<sort>=def

let d=getEntityDefinition0(s, sort).derivedDataType0 in

if d.specialization0 = undefined then undefined

else

let btn= d.specialization0.s-<base type>.s-<name> in

if btn = "String" ∨ btn = "Vector"

then predefinedId0("Integer"))

elseif btn = "Array"

then d.actualContextParameterList0[1]

else undefined

endif

endlet

endif

endlet

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

getFieldSort0(s: <sort>, n:<name>):<sort>=def

let d=getEntityDefinition0(s, sort).derivedDataType0 in

let cons= d.s-<data type definition body>.s-<data type constructor> in

if cons ∈ <structure definition> then

take({fos.s-<sort> : fos∈<fields of sort>∧

(∃name∈<name>: name.parentAS0=fos∧name=n)})

else undefined

endif

endlet

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

isSatisfyDecisionCondition0(bl:BindingList0, d: <decision>): Boolean=def

let q=d.s-<question>, rcs=d.rangeConditionList0.toSet in

isSatisfyExpressionCondition0(bl, q)∧

(∀rc1∈rcs:∀ce1∈<constant expression>:isAncestorAS1(rc1, ce1)⇒

isSatisfyExpressionCondition0(bl, ce1)∧

isSortCompatible0(getStaticSort0(ce1, bl), getStaticSort0(q, bl)∧

(∀rc2∈rcs: ∀ce2∈<constant expression>:isAncestorAS1(rc2, ce2)⇒

(isSortCompatible0(getStaticSort0(ce1, bl), getStaticSort0(ce2, bl))∨

isSortCompatible0(getStaticSort0(ce2, bl), getStaticSort0(ce1, bl))))

endlet

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

isSatisfyExpressionCondition0(bl:BindingList0, exp: <expression>): Boolean=def

case exp of

| <binary expression> then

let opDef = getDefinitionInBindingList0(binaryOpName0(exp), bl) in

let fpl = opDef.operationParameterSortList0 in

fpl.length = 2∧

isSortCompatible0(getStaticSort0(exp.s-<expression>, bl), fpl[1])∧

isSortCompatible0(getStaticSort0(exp.s2-<expression>, bl), fpl[2])∧

isSatisfyExpressionCondition0 (bl, exp.s-<expression>)∧

isSatisfyExpressionCondition0 (bl, exp.s2-<expression>)

endlet

endlet

| <create expression> then isSatisfyCreateCondition0(bl, exp)

| <equality expression> then isSatisfyEqualityExpCondition0(bl, exp)

| <operand5> then

let pr = exp.s-<primary> in

case exp.s-implicit of

| <hyphen> then

let fpl = getDefinitionInBindingList0(mk-<name>("-"),bl).operationParameterSortList0 in

fpl.length = 1∧

isSortCompatible0(getStaticSort0(pr, bl), fpl[1])∧

isSatisfyPrimaryCondition0(bl, pr)

endlet

| not then

let fpl = getDefinitionInBindingList0(mk-<name>("not"),bl).operationParameterSortList0 in

fpl.length = 1∧

isSortCompatible0(getStaticSort0(pr, bl), fpl[1])∧

isSatisfyPrimaryCondition0(bl, pr)

endlet

otherwise isSatisfyPrimaryCondition0(bl, pr)

endcase

endlet // pr

| <range check expression> then isSatisfyRangeCheckCondition0(bl, exp)

| <value returning procedure call> then

if exp∈<procedure call body> then isSatisfyProcedureCallBodyCondition0(bl, exp)

else isSatisfyRemoteProcCallBodyCondition0(bl, exp)

endif

endcase

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

isSatisfyCreateCondition0(bl: BindingList0, ce: <create expression>):Boolean =def

let def = ce.getCreatedAgentDefinition0 in

if def =undefined then false

else

let fpl = def.agentFormalParameterList0 in

let apl = ce.actualParameterList0 in

fpl.length=apl.length ∧

(∀i∈1..fpl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpl[i].parentAS1.s-<sort>)∧

isSatisfyExpressionCondition0(bl, apl[i]))

endlet

endlet

endif endlet

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

getCreateExpSort0 (ce: <create expression>): <sort>=def

let def = ce.getCreatedAgentDefinition0 in

if def =undefined then Pid

else def.identifier0

endif endlet

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

isSatisfyProcedureCallBodyCondition0(bl: BindingList0, body: <procedure call body>):Boolean=def

let apl = body.actualParameterList0 in

let fpsl=*body.*calledProcedure0.formalParameterSortList0 in

fpsl.length=apl.length ∧

(∀i∈1..fpsl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpsl[i)∧

isSatisfyExpressionCondition0(bl, apl[i]))

endlet

endlet

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

getProcCallBodySort0 (body: <procedure call body>): <sort>=def

case body.s-implicit of

| id = <identifier> then getEntityDefinition0(id, procedure).procedureResult0

| te = <type expression> then te.baseType0.procedureResult0

endcase

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

procedureResult0(pd: <procedure definition>):<sort>=def

if pd.s-<procedure heading>.s-<procedure result>≠undefined then

pd.s-<procedure heading>.s-<procedure result>

elseif pd.specialization0 ≠ undefined then

pd.baseType0.procedureResult0

else undefined

endif

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

getRemoteProcCallBodySort0 (body: <remote procedure call body>, bl: BindingList0): <sort>=def

getEntityDefinition0(body.s-<remote procedure<identifier>,remote procedure).procedureResult0

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

isSatisfyRemoteProcCallBodyCondition0(bl: BindingList0, body: <remote procedure call body>):

Boolean =def

let rpd = getEntityDefinition0(body.s-<remote procedure<identifier>,remote procedure) in

let fpsl= rpd.formalParameterSortList0 in

let apl = body.actualParameterList0 in

fpsl.length=apl.length ∧

(∀i∈1..fpsl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpsl[i])∧

isSatisfyExpressionCondition0(bl, apl[i]))

endlet

endlet

endlet

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

operationResultSort0(os:<operation signature>):<sort>=def

if os∈PredefinedOperation0 then os.getPredefinedOpResult0

else os.s-<result>.s-<sort>

endif

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

isSatisfyRangeCheckCondition0(bl: BindingList0, rce: <range check expression>): Boolean=def

if rce.s-implicit ∈ <range check constrained sort> then

isSatisfyExpressionCondition0(bl, rce.s-<expression>)∧

isSameSort0(getStaticSort0(rce.s-<expression>, bl), rcesort.s-<sort<identifier>)

elseif rce.s-implicit ∈ <identifier> ∧ getEntityDefinition0(rce.s-implicit,sort) ∈ <syntype definition>

then // syntype

isSatisfyRangeCheckCondition0(bl,

mk-<range check expression>(rce.s-<expression>,

mk-<range check constrained sort>( rce.s-implicit,

getEntityDefinition0(rce.s-implicit,sort).s-implicit.s-<constraint>)))

elseif rce.s-implicit ∈ <identifier> ∧ isPidSort0(mk-<sort>( (rce.s-implicit))

then // pid sort

isPidSort0(getStaticSort0(rce.s-<expression>, bl)) // expression has to be pid sort

else true // <datatype<type expression> any expression valid. Type check in F3

endif

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

isSatisfyEqualityExpCondition0(bl: BindingList0, eq: <equality expression>): Boolean=def

isSatisfyExpressionCondition0(bl, eq.s-<expression>)∧

isSatisfyExpressionCondition0(bl, eq.s2-<expression>)∧

(isSortCompatible0(getStaticSort0(eq.s-<expression>, bl), getStaticSort0(eq.s2-<expression>, bl))∨

isSortCompatible0(getStaticSort0(eq.s2-<expression>, bl),

getStaticSort0(eq.s-<expression>, bl)))

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

isSatisfyPrimaryCondition0(bl: BindingList0, pr: <primary>):Boolean=def

case pr of

| <operator application> then isSatisfyOpAppCondition0(bl, pr)

| <method application> then isSatisfyMethodAppCondition0(bl, pr)

| <expression> then isSatisfyExpressionCondition0(bl, pr)

| <conditional expression> then

isSatisfyExpressionCondition0(bl, pr.s-<expression>)∧

isSatisfyExpressionCondition0(bl, pr.s-<consequence expression>)∧

isSatisfyExpressionCondition0(bl, pr.s-<alternative expression>)

| <extended primary> then isSatisfyExtendedPrimaryCond0(bl, pr)

otherwise true

endcase

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

isSatisfyExtendedPrimaryCond0(bl: BindingList0, epr: <extended primary>):Boolean=def

case epr of

| <indexed primary> then

isSatisfyPrimaryCondition0(epr.s-<primary>, bl)∧

(∀i∈1..epr.s-<actual parameter>-seq.length:

isSatisfyExpressionCondition0(bl,epr.s-<actual parameter>-seq[i]))∧

isSortCompatible0(getStaticSort0(epr.s-<actual parameter>-seq[1], bl),

getIndexSort0(getPrimarySort0(epr.s-<primary>, bl)))

| <field primary> then

(epr.s-<primary> ≠ undefined)⇒

(isSatisfyPrimaryCondition0(epr.s-<primary>, bl)∧

getFieldSort0(getPrimarySort0(epr.s-<primary>, bl), epr.s-<field name>) ≠ undefined)

| <composite primary> then

let sl = <getStaticSort0(para, bl)| para in epr.s-<actual parameter>-seq> in

epr.s-<actual parameter>-seq≠empty⇒

(getCompositeSort0(sl)≠ undefined ∧

(∀i∈1..epr.s-<actual parameter>-seq.length: epr.s-<actual parameter>-seq[i]= undefined∨

isSatisfyExpressionCondition0(bl,epr.s-<actual parameter>-seq[i])))

endlet

endcase

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

getCompositeSort0(sl:[<sort>]\*):<sort>= def

let def = take({ d∈<data type definition>∪<syntype definition>:

∃cons∈<structure definition>:

let fsl = cons.fieldSortList0 in

fsl.length=sl.length∧

(∀i∈1..fsl.length: sl[i]=undefined∨isSortCompatible0(sl[i], fsl[i]))

endlet // fsl

}) in

def.derivedDataType0.identifier0

endlet // def

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

getPrimarySort0(pr: <primary>, bl: BindingList0): <sort>= def

case pr of

| <operation application> then getOpAppSort0(pr, bl)

| <literal> then

getDefinitionInBindingList0 (pr, bl).surroundingScopeUnit0.identifier0

| <expression> then getStaticSort0(pr, bl)

| <conditional expression> then getStaticSort0(pr.s-<consequence expression>, bl)

| <indexed primary> then getItemSort0(getPrimarySort0(pr.s-<primary>, bl))

| <field primary> then getFieldSort0(getPrimarySort0(pr.s-<primary>, bl), pr.s-<field name>)

| <composite primary> then

getCompositeSort0(<getStaticSort0(para, bl)| para in pr.s-<actual parameter>-seq>)

| <variable access> then getVarAccessSort0(pr)

| <imperative expression> then getImperativeExpSort0(pr)

otherwise getSynonymSort0(pr) // <synonym>

endcase

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

getVarAccessSort0(va: <variable access>): <sort>= def

if va∈ <identifier> then

getEntityDefinition0(va, variable).s-<sort>

else

let od = parentAS0ofKind(va, <operation definition>) in

od.operationFormalparameterList0[1].parentAS1.s-<sort>

endlet

endif

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

getImperativeExpSort0(ie: <imperative expression>): <sort>= def

case ie of

| <now expression> then predefinedId0("Time")

| <pid expression> then

if ie ≠ self then

predefinedId0("Pid"))

else

let def = parentAS0ofKind(ie, <agent definition>∪<agent type definition>) in

if def = undefined then predefinedId0("Pid"))

else def.identifier0

endif

endlet

endif

| <any expression> then ie.s-<sort>

| <state expression> then predefinedId0("Charstring"))

endcase

The function getSynonymSort0 gets the static sort of a <synonym>. For a valid model the <constant expression> has only one possible static sort, which is returned. If there is more than one possible sort, undefined is returned.

getSynonymSort0(s: <synonym>): <sort>= def

let sd = getEntityDefinition0(s, synonym) in //sd is an external or internal synonym definition item

if sd.s-<sort> ≠ undefined then sd.s-<sort> //sd is an external item or internal item with sort

else sd.s-<constant expression>.staticSort0 // sort of <constant expression> of internal synonym definition

endif endlet

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

isSatisfyOpAppCondition0(bl:BindingList0, oa:<operation application>): Boolean=def

let opDef = getDefinitionInBindingList0(oa.s-<operation identifier>.s-<name>, bl) in

let fpl = opDef.operationParameterSortList0 in

let apl =oa.actualParameterList0 in

fpl.length = apl.length ∧

(∀i∈1..fpl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpl[i])∧

isSatisfyExpressionCondition0(bl, apl[i])

endlet

The function getOpAppSort0 gets the static sort of an <operation application>.

getOpAppSort0(oa:<operation application>, bl: BindingList0): <sort>= def

getDefinitionInBindingList0(oa.s-<operation identifier>.s-<name>, bl).operationResultSort0

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

isSatisfyMethodAppCondition0(bl:BindingList0, ma: <method application>): Boolean=def

let opDef = getDefinitionInBindingList0(ma.s-<operation identifier>.s-<name>, bl) in

let fpl = opDef.operationParameterSortList0 in

let apl =ma.actualParameterList0 in

fpl.length = apl.length ∧isSatisfyPrimaryCondition0(bl, ma.s-<primary>)∧

(∀i∈1..fpl.length: isSortCompatible0(getStaticSort0(apl[i], bl), fpl[i])∧

isSatisfyExpressionCondition0(bl, apl[i])

endlet

The function operationParameterSortList0 gets the operation formal parameter sort list.

operationParameterSortList0(os:<operation signature>):<sort>\*=def

if os. getPredefinedOpParas0 ≠ undefined then

os.getPredefinedOpParas0

else

<para.s-<sort> : para in os.operationSignatureParameterList0>

endif

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

getDefinitionInBindingList0 (n: <name>, bl: BindingList0): EntityDefinition0=def

take ({d∈EntityDefinition0: ∃i∈1..bl.length: bl[i].s-<name>=n∧ bl[i].s-EntityDefinition0=d})

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

possibleDefinitionSet0(n: <name>):EntityDefinition0-set =def

{d∈EntityDefinition0: ((d.entityName0=n)∨(isRenamedBy0(d.entityName0, n)))∧

( (d.entityKind0=n.idKind0∧isVisibleIn0(d, n.surroundingScopeUnit0))∨

(d∈PredefinedLiteral0∧ n.idKind0=literal)∨

(d∈PredefinedOperation0∧ n.idKind0∈{operator, method}))}

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

isRenamedBy0(n1, n2: <name>):Boolean=def

(∃rp∈<rename pair>:(rp.s-<operation name> = n2∧ rp.s2-<operation name>=n1)∨

(rp.s-<literal name> = n2∧ rp.s2-<literal name>=n1))∨

(∃n3∈<name>: isRenamedBy0(n1, n3)∧ isRenamedBy0(n3, n2))

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

actualParameterList0(d:

<create expression>∪<procedure call body>∪<remote procedure call body>

∪<value returning procedure call>∪<operation application>): <expression>\*=def

d.s-<actual parameter>-seq

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

getCreatedAgentDefinition0 (ce: <create expression>): <agent definition>∪<agent type definition>=def

let id = ce.s-implicit in

if id∈<identifier> then getEntityDefinition0(id, id.idKind0)

elseif id.surroundingScopeUnit0∈<agent definition>∪<agent type definition> then

id.surroundingScopeUnit0

else undefined

endif

The function getStaticSort0 gets the static sort of an expression.

getStaticSort0(exp: <expression>, bl:BindingList0): <sort>=def

case exp of

| <create expression> then getCreateExpSort0(exp)

| <procedure call body> then getProcCallBodySort0(exp)

| <remote procedure call body> then getRemoteProcCallBodySort0(exp, bl)

| <range check expression> then predefinedSort0("Boolean"))

| <equality expression> then predefinedSort0("Boolean"))

| <binary expression> then

getDefinitionInBindingList0(binaryOpName0(exp), bl).operationResultSort0

| <operand5> then

case exp.s-implicit of

| <hyphen> then getDefinitionInBindingList0(mk-<name>("-"), bl).operationResultSort0

| not then getDefinitionInBindingList0(mk-<name>("not"), bl).operationResultSort0

otherwise getPrimarySort0(exp.s-<primary>, bl)

endcase

endcase

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

mismatchNumber0(bl:BindingList0, c:Context0): Nat=def

case c of

|<assignment> then mismatchNumberOfAssignment0(bl, c)

|<decision> then mismatchNumberOfDecision0(bl, c)

|<expression> then mismatchNumberOfExpression0(bl, c)

endcase

mismatchNumberOfAssignment0(bl:BindingList0, ass: <assignment>): Nat=def

let varSort= getVariableSort0(ass.s-<variable>) in

let expSort= getStaticSort0(ass.s-<expression>, bl) in

case ass.s-<variable> of

| <identifier>∪<field variable> then

if ¬isSameSort0(varSort, expSort)

then 1+mismatchNumberOfExpression0(bl, ass.s-<expression>)

else mismatchNumberOfExpression0(bl, ass.s-<expression>)

endif

| <indexed variable> then

if ¬isSameSort0(varSort, expSort) then

1 + mismatchNumberOfExpression0(bl, ass.s-<expression>)+

mismatchNumberInIndexVariable0(bl, ass.s-<variable>)

else

mismatchNumberOfExpression0(bl, ass.s-<expression>)+

mismatchNumberInIndexVariable0(bl, ass.s-<variable>)

endif

endcase

mismatchNumberInIndexVariable0(bl:BindingList0, var:<indexed variable>): Nat=def

let acp=var.s-<actual parameter>-seq in

if var.s-<variable>∉<indexed variable> then

if ¬isSameSort0(getStaticSort0(acp[1],bl), getIndexSort0(getVariableSort0(var))) then 1

else 0

endif

elseif ¬isSameSort0(getStaticSort0(acp[1],bl), getIndexSort0(getVariableSort0(var))) then

1 + mismatchNumberInIndexVariable0(bl, var.s-<variable>)

else mismatchNumberInIndexVariable0(bl, var.s-<variable>)

endif

endlet

mismatchNumberOfDecision0(bl:BindingList0, d: <decision>): Nat=def

let q=d.s-<question>, rcl=d.rangeConditionList0 in

mismatchNumberOfExpression0(bl, q) + mismatchNumberOfRangeConditionList0(bl, rcl))

endlet

mismatchNumberOfRangeConditionList0(bl:BindingList0, rcl: <range condition>\*): Nat=def

if rcl.= empty then 0

else

mismatchNumberOfRangeCond0(bl, rcl.head) +

mismatchNumberOfRangeConditionList0 (bl, rcl.tail])

endif

mismatchNumberOfRangeCond0(bl:BindingList0, rl: <range>\*): Nat=def

if rl = empty then 0

else

mismatchNumberOfRange0(bl, rl.head)+ mismatchNumberOfRangeCond0 (bl, rl.tail])

endif

mismatchNumberOfRange0(bl:BindingList0, range: <range>): Nat=def

case range of

| <closed range> then

mismatchNumberOfExpression0(bl,range.s-<constant>) +

mismatchNumberOfExpression0(bl,range.s2-<constant> )

| <open range> then

if range∈<constant> then mismatchNumberOfExpression0(bl,range)

else

mismatchNumberOfExpression0(bl,range.s-<constant>)+

mismatchNumberOfExpression0(bl,range.s2-<constant> )

endif

endcase

mismatchNumberOfExpression0(bl:BindingList0, exp: <expression>): Nat=def

case exp of

| <binary expression> then

let opDef = getDefinitionInBindingList0(exp.s-implicit, bl) in

let fpsl = opDef.operationParameterSortList0 in

mismatchNumberOfParas0(bl, fpsl, exp.s-<expression>⁀ exp.s2-<expression>)+

mismatchNumberOfExpression0(bl, exp.s-<expression>)+

mismatchNumberOfExpression0(bl, exp.s2-<expression>)

endlet

| <create expression> then mismatchNumberOfCreateExp0(bl, exp)

| <equality expression> then

mismatchNumberOfExpression0(bl,exp.s-<expression>)+

mismatchNumberOfExpression0(bl,exp.s2-<expression>)

| <range check expression> then mismatchNumberOfExpression0(bl,exp.s-<expression>)

| <operand5> then

let pr = exp.s-<primary> in

if exp.s-implicit = undefined then

mismatchNumberOfPrimary0(bl, pr)

else

let opDef = getDefinitionInBindingList0(exp.s-implicit, bl) in

let fpsl = opDef.operationParameterSortList0 in

if isSameSort0(getStaticSort0(pr, bl), fpsl[1])

then mismatchNumberOfPrimary0(bl, pr)

else mismatchNumberOfPrimary0(bl, pr)+1

endif

endlet

endlet

endif

endlet

| <value returning procedure call> then

if exp∈<procedure call body> then

mismatchNumberOfProcedureCallBody0(bl, call)

else

mismatchNumberOfRemoteProcCallBody0(bl, call)

endif

endcase

mismatchNumberOfCreateExp0(bl: BindingList0, ce: <create expression>):Nat=def

let def = ce.getCreatedAgentDefinition0 in

if def =undefined then 0

else

mismatchNumberOfParas0(bl, def.formalParameterSortList0, ce.actualParameterList0) +

mismatchNumberOfActualParas0(bl, ce.actualParameterList0)

endif

endlet

mismatchNumberOfProcedureCallBody0(bl: BindingList0, body: <procedure call body>): Nat=def

case body.s-implicit of

| id=<identifier> then

let fpsl=getEntityDefinition0(id, procedure).formalParameterSortList0 in

mismatchNumberOfParas0(bl, fpsl, apl)

endlet

| te=<type expression> then

let fpsl=id.baseType0.formalParameterSortList0 in

let apl = body.actualParameterList0 in

mismatchNumberOfParas0(bl, fpsl, apl) + mismatchNumberOfActualParas0(bl, apl)

endlet

endlet

endcase

mismatchNumberOfRemoteProcCallBody0(bl: BindingList0, body: <remote procedure call body>):

Nat=def

let rpd = getEntityDefinition0(body.s- <identifier>,remote procedure) in

let fpsl= rpd.formalParameterSortList0 in

let apl = body.actualParameterList0 in

mismatchNumberOfParas0(bl, fpsl, apl) + mismatchNumberOfActualParas0(bl, apl)

endlet

endlet

endlet

mismatchNumberOfPrimary0(bl: BindingList0, pr: <primary>):Nat=def

case pr of

| <operator application> then mismatchNumberOfOpApp0(bl, pr)

| <method application> then mismatchNumberOfMethodApp0(bl, pr)

| <expression> then mismatchNumberOfExpression0(bl, pr)

| <conditional expression> then

mismatchNumberOfExpression0(bl, pr.s-<expression>) +

mismatchNumberOfExpression0(bl, pr.s-<consequence expression>) +

mismatchNumberOfExpression0(bl, pr.s-<alternative expression>)

otherwise 0

endcase

mismatchNumberOfOpApp0(bl:BindingList0, oa: <operator application>): Nat=def

let opDef = getDefinitionInBindingList0(oa.s-<operation identifier>.s-<name>, bl) in

let fpsl = opDef.operationParameterSortList0 in

let apl =oa.actualParameterList0 in

mismatchNumberOfParas0(bl, fpsl, apl)+mismatchNumberOfActualParas0(bl, apl)

endlet

endlet

endlet

mismatchNumberOfMethodApp0(bl:BindingList0, ma: <method application>):Nat=def

let opDef = getDefinitionInBindingList0(ma.s-<operation identifier>.s-<name>, bl) in

let fpsl = opDef.operationParameterSortList0 in

let apl =ma.actualParameterList0 in

mismatchNumberOfParas0(bl, fpsl, apl)+mismatchNumberOfActualParas0(bl, apl)+

mismatchNumberOfPrimary0(bl, ma.s-<primary>)

endlet

endlet

endlet

mismatchNumberOfParas0(bl: BindingList0, fpsl: <sort>\*, apl: <expression>\*):Nat=def

if fpsl = empty then 0

elseif fpsl.head=undefined∨isSameSort0(fpsl.head, apl.head)

then mismatchNumberOfParas0(bl, fpsl.tail, apl.tail)

else mismatchNumberOfParas0(bl, fpsl.tail, apl.tail)+1

endif

mismatchNumberOfActualParas0(bl: BindingList0, apl: <expression>\*): Nat=def

if apl = empty then 0

else mismatchNumberOfExpression0(bl, apl.head)+mismatchNumberOfActualParas0(bl, apl.tail)

endif

formalParameterSortList0(d:

<agent definition> ∪ <agent type definition> ∪ <composite state definition> ∪

<composite state type definition> ∪ <procedure definition> ∪

<remote procedure definition>):<sort>\*= def

case d of

| <agent definition> ∪ <agent type definition>∪

<composite state definition> ∪ <composite state type definition> then

<fp.parentAS1.s-<sort> | fp in d.agentFormalParameterList0>

| <procedure definition> then

<fp.parentAS1.s-<sort> | fp in d.procedureFormalParameterList0>

| <remote procedure definition> then

<fp.s-<sort> | fp in d.s-<procedure signature>.s-<formal parameter>>

endcase

The function staticSortSet0 returns the possible static sort set for an <expression> that is in an <assignment> or a <decision> or <expression>.

staticSortSet0(exp: <expression>):<sort>-set =def

{getStaticSort0(exp, bl) | bl∈getBindingListSet0(exp.contextOfExp0)}

The function staticSort0 gets one of the sorts from possible static sort set for an <expression> that is in an <assignment> or a <decision> or <expression>, and therefore for an expression with all names resolved returns the static sort of the expression as the possible static sort set and has only one member.

staticSort0(expr:<expression>):<sort>=def

let possiblesorts = expr.staticSortSet0 in

if | possiblesorts | = 1 then take(possiblesorts) else undefined endif

endlet

The function contextOfExp0 finds the context in AS0 (<assignment> or <decision> or <expression>) that the <expression> is in.

contextOfExp0(exp: <expression>):Context0=def

if (∃ass∈<assignment>: isAncestorAS0(ass, exp)) then

parentAS0ofKind(exp, <assignment>)

elseif (∃dec∈<decision>: isAncestorAS1(dec, exp)) then

parentAS0ofKind(exp, <decision>)

elseif (∃exp1∈<expression>: isAncestorAS0(exp1, exp)) then

parentAS0ofKind(exp, <expression>).contextOfExp0

else exp

endif

The functiongetPredefinedOpParas0gets the sort list of the formal parameters of a predefined operation. This function is not formally defined in this Recommendation (ITU-T Z.100).

getPredefinedOpParas0: PredefinedOperation0 → <sort>\*

The functiongetPredefinedOpResult0gets the result sort of a predefined operation. This function is not formally defined in this Recommendation (ITU-T Z.100).

getPredefinedOpResult0: PredefinedOperation0 → <sort>

#### F2.2.2.3 Path item

Abstract syntax

Path-item = Package-qualifier

| Agent-type-qualifier

| Agent-qualifier

| State-type-qualifier

| State-qualifier

| Data-type-qualifier

| Procedure-qualifier

*|* Interface-qualifier

| Compound-node-qualifier

Package-qualifier :: Package-name

Agent-type-qualifier :: Agent-type-name

Agent-qualifier :: Agent-name

State-type-qualifier :: State-type-name

State-qualifier :: State-name

Data-type-qualifier :: Data-type-name

Procedure-qualifier :: Procedure-name

Interface-qualifier :: Interface-name

Compound-node-qualifier :: Compound-node-name

Concrete syntax

<path item> :: <scope unit kind> <name>

<scope unit kind> =

package

| system type

| system

| block

| block type

| process

| process type

| state

| state type

| procedure

| signal

| type

| operator

| method

| interface

| composition

Mapping to abstract syntax

| <path item>(package,n) then mk-Package-qualifier(n)

| <path item>(system,n) then mk-Agent-qualifier(n)

| <path item>(system type,n) then mk-Agent-type-qualifier(n)

| <path item>(block,n) then mk-Agent-qualifier(n)

| <path item>(block type,n) then mk-Agent-type-qualifier(n)

| <path item>(process,n) then mk-Agent-qualifier(n)

| <path item>(process type,n) then mk-Agent-type-qualifier(n)

| <path item>(state,n) then mk-State-qualifier(n)

| <path item>(statetype,n) then mk-State-type-qualifier(n)

| <path item>(procedure,n) then mk-Procedure-qualifier(n)

| <path item>(operator,n) then mk-Procedure-qualifier(n)

| <path item>(method,n) then mk-Procedure-qualifier(n)

| <path item>(type,n) then mk-Data-type-qualifier(n)

| <path item>(interface,n) then mk-Interface-qualifier(n)

| <path item>(composition,n) then mk-Compound-node-qualifier(n)

### F2.2.3 Informal text

Abstract syntax

Informal-text :: ...

Concrete syntax

<informal text> :: <character string>

Mapping to abstract syntax

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

### F2.2.4 General framework

#### F2.2.4.1 SDL‑2010 specification

Abstract syntax

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

Concrete syntax

<sdl specification> ::

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

<textual system specification> ::

<agent definition>

| <package use clause>\* <textual typebased agent definition>

Transformations

<sdl specification>(sys, r)

provided sys ∈ (<process definition> ∪ <textual typebased process definition> ∪

<block definition> ∪ <textual typebased block definition>)

=5=> <sdl specification>(

<system definition>(empty,

<system heading>(sys.name0, <agent additional heading>(undefined, empty)),

<agent structure>(undefined, < sys >, <agent body>(undefined, empty))),

r)

A <system specification> being a <process definition> or a <textual typebased process definition> is derived syntax for a <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. See clause 5.3.1 *Model* of [ITU‑T Z.106].

A <system specification> being a <block definition> or a <textual typebased block definition> is derived syntax for a <system definition> having the same name as the block, containing implicit channels and containing the <block definition> or <textual typebased block definition> as the only definition. See clause 5.3.1 *Model* of [ITU‑T Z.106].

Mapping to abstract syntax

| <sdl specification>(s=<textual system specification>, refs) then

let packages =

{ p ∈ ( refs.toSet ∩ <package definition> ) : p.surroundingScopeUnit0 = undefined } in

mk-Sdl-specification(Mapping(s), { Mapping(p): p ∈ packages },)

endlet

| <sdl specification>(unit=<package definition>, refs) then

let packages = unit ∪

{ p ∈ ( refs.toSet ∩ <package definition>) : p.surroundingScopeUnit0 = undefined } in

mk-Sdl-specification({ Mapping(p): p ∈ packages })

endlet

#### F2.2.4.2 Package

Abstract syntax

Package-definition :: Package-name

Package-definition-set

Data-type-definition-set

Syntype-definition-set

Signal-definition-set

Agent-type-definition-set

Composite-state-type-definition-set

Procedure-definition-set

Concrete syntax

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

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

<entity in package> =

<agent type definition>

| <agent type reference>

| <package definition>

| <package reference>

| <signal definition list>

| <signal list definition>

| <remote variable definition>

| <data definition>

| <procedure definition>

| <procedure reference>

| <remote procedure definition>

| <composite state type definition>

| <composite state type reference>

| <select definition>

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

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

<selected entity kind> =

system type

| block type

| process type

| package

| signal

| procedure

| remote procedure

| type

| signallist

| state type

| synonym

| remote

| interface

<package public> = <definition selection>\*

Conditions on concrete syntax

See clause 7.2 *Concrete grammar* of [ITU‑T Z.101].

∀pi∈ <package use clause>.s-<identifier>:

getEntityDefinition0 (pi, package) ≠ undefined

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

∀pd∈ <package definition>:

pd.parentAS0∈<sdl specification> ⇒ pd.contextQualifier0 = undefined

There shall be a <qualifier> in <package<identifier> only if the package is logically contained in another package.

∀ds1,ds2∈<definition selection>:

(ds1. parentAS0 = ds2. parentAS0∧ ds1 ≠ ds2)⇒

(ds1.s-<selected entity kind> ≠ ds2.s-<selected entity kind> ∨ ds1.s-<name> ≠ ds2.s-<name>)

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

∀ds∈<definition selection>: ds. parentAS0 ∈<package public> ⇒

(∃!d∈EntityDefinition0∪ Reference0 :

d.surroundingScopeUnit0 = ds.surroundingScopeUnit0∧ d. entityName0 = ds.s-<name> ∧

if ds.s-<selected entity kind> = undefined

then true

else d.entityKind0 = ds.s-<selected entity kind>

endif)

For a <definition selection> in a <package public> clause, the <selected entity kind> is allowed to be omitted only if the name is used for just one defining occurrence directly in the <package>. For a <definition selection> in <package public>, if the <selected entity kind> is not omitted there shall exist an entity definition having the same entity kind and the same name in the package.

∀ uc ∈ <package use clause>:∀ds∈<definition selection>:

let pd = uc.usedPackage0 in

ds. parentAS0 = uc ∧ ds.s-<selected entity kind> = undefined⇒

((∃!ds1∈<definition selection>:

ds1.surroundingScopeUnit0= pd∧ds.s-<name>= ds1.**s**-<name>) ∨

(( pd.s-<package heading>.s-<package public> = undefined)∧

(∃!d∈ EntityDefinition0∪ Reference0 :

d.surroundingScopeUnit0= pd ∧ d. entityName0= ds.s-<name>)))

endlet

For a <definition selection> in a <package use clause>, <selected entity kind> is allowed to 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.

Transformations

let usePredef = <package use clause>(

<identifier>(empty, <name>("Predefined")), undefined) in

<package definition>(uses, heading, entities)

provided uses.head ≠ usePredef

=6=> <package definition>(< usePredef > ⁀ uses, heading, entities)

endlet

let usePredef = <package use clause>(

<identifier>(empty, <name>("Predefined") ), undefined) in

<system definition>(uses, heading, struct)

provided uses.head ≠ usePredef

=6=> <system definition>(< usePredef > ⁀ uses, heading, struct)

endlet

The *Package-definition-****set*** of an *Sdl-specification* always includes the *Package-definition* of **package** Predefined defined in [ITU-T Z.104]; consequently this package does not have to be explicitly included as a <referenced definition>. See clause 7.1 *Model* of [ITU‑T Z.101].

< <package use clause>(id, sel1) > ⁀ something ⁀ < <package use clause>(id, sel2) >

=6=>

(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> items of a definition (in the same text area or different text areas of the definition), these are replaced by one <package use clause> that selects the union of the definitions selected in the <package use clause> items. See clause 7.2 *Model* of [ITU‑T Z.101].

Mapping to abstract syntax

| <package definition>(\*, <package heading>(\*, n, \*), entities) then

let entitiesSet = { Mapping(entities[i]:i ∈ 1..entities.length} in

mk-Package-definition(n,

{ pkd ∈ entitiesSet: (pkd ∈ Package-definition) },

{ dtd ∈ entitiesSet: (dtd ∈ Data-type-definition) }∪

{ implicitInterface1(atd) | (atd ∈ entitiesSet ∧ atd ∈ Agent-type-definition) },

{ snd ∈ entitiesSet: (snd ∈ Syntype-definition) },

{ sgd ∈ entitiesSet: (sgd ∈ Signal-definition) },

{ atd ∈ entitiesSet: (atd ∈ Agent-type-definition) },

{ std ∈ entitiesSet: (std ∈ Composite-state-type-definition) },

{ pdf ∈ entitiesSet: (pdf ∈ Procedure-definition) } )

endlet // entitiesSet

Auxiliary functions

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

usedPackageDefinitionList0(su: ScopeUnit0): <package definition> \*=def

< u.usedPackage0 | u in su.s-<package use clause>-seq >

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

usedPackage0 (uc: <package use clause>): <package definition> =def

getEntityDefinition0 (uc.s-<identifier>, package)

The function *implicitInterface1* returns the implicit interface definition associated with an Agent-type-definition or an Agent-definition. See section F2.2.9.2.

implicitInterface1(d: Agent-definition ∪ Agent-type-definition ∪ State-machine): Interface-definition =def

case d of

| Agent-definition then

let sigDefs = getEntityDefinition1(d.s-Agent-type-identifier, agent type).s-Signal-definition-set in

mk-Interface-definition (d.s-Agent-name, // *Sort* name – same as the agent

mk-Literal-signature(mk-Name("null"),

mk-Result(mk-Identifier(Mapping(d.fullQualifier0), d.s-Agent-type-name), REF),

0),

*empty*, // Data-type-identifier-set

sigDefs, // Signal-definition-set

*d.baseType1.inputSignalSet1* ∪ { sd.identifier1 | sd ∈ sigDefs } // *Signal-identifier*-set

)

endlet

| Agent-type-definition then

let sigDefs = d.s-Signal-definition-set in

mk-Interface-definition (d.s-Agent-type-name,

mk-Literal-signature(mk-Name("null"),

mk-Result(mk-Identifier(Mapping(d.fullQualifier0), d.s-Agent-type-name), REF),

0),

*empty*,

sigDefs, // Signal-definition-set

*d.baseType1.inputSignalSet1* ∪ { sd.identifier1 | sd ∈ sigDefs } // *Signal-identifier*-set

)

endlet

| State-machine then

mk-Interface-definition (d.s-State-name,

mk-Literal-signature(mk-Name("null"),

mk-Result(mk-Identifier(Mapping(d.fullQualifier0), d.s-State-name), REF),

0),

*empty*,

∅ // empty Signal-definition-set

let *parent* =

*parentAS1ofKind*(*d*, Agent-definition ∪ Agent-type-definition) in

let *siblings* =

if *parent* ∈ Agent-definition

then *parent*.s-Agent-type-definition.s-Agent-definition-set

else *parent*.s-Agent-definition-set

endif in

( *parent*.implicitInterface1.s-Signal-identifier-set // parent interface signals

\ { *siblingsig* ∈ *sibling*.implicitInterface1 : *sibling* ∈ *siblings* } ) // except to siblings

∪ *d.baseType1.inputSignalSet1* // incoming signals of composite state type of state machine

endlet // siblings

endlet // parent

)

endcase

The implicit *Interface-definition* for an agent type (or agent or state machine) has a *Sort* with the same *Name* as the agent type (or agent or state machine respectively). See clause 12.1.2 *Semantics* of [ITU‑T Z.101].

The implicit *Interface-definition* defined by a state machine of an agent type contains (in its interface specialization – see [ITU-T Z.104]) the interface defined by the agent type itself except any part of that interface concerned only with contained agents. The interface also contains in its interface specialization all interfaces given in the incoming signal lists associated with any gates of the state machine. The interface also contains in its <interface use list> all signals given in the incoming signal lists associated with gates of the state machine. See clause 12.1.2 *Semantics* of [ITU‑T Z.101].

#### F2.2.4.3 Referenced definition

Concrete syntax

<referenced definition> =

<definition>

<definition> =

| <package definition>

| <agent definition>

| <agent type definition>

| <composite state definition>

| <composite state type definition>

| <procedure definition>

| <operation definition>

<package reference> :: <package<identifier>

<procedure reference> :: <procedure reference heading>

<procedure reference heading> ::

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

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

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

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

<agent type reference> =

<system type reference>

| <block type reference>

| <process type reference>

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

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

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

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

Conditions on concrete syntax

Except as mentioned below (for <formal context parameters>) the following conditions are derived from clause 7.3 of [ITU‑T Z.101].

∀refDef∈<referenced definition>:refDef.referencedBy0≠undefined ∧ refDef.parentAS0 = <sdl specification>

∨ (refDef∈<package definition> ∧ refDef.contextQualifier0 = undefined)

For each <referenced definition> except any outermost <package definition>, there shall be a reference in the associated <package> or <system specification>.

∀ref∈ Reference0 : ref.referencedDefinition0 ≠ undefined

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

∀ref ∈ Reference0 : ∀refDef ∈<referenced definition>:

ref.s-<formal context parameters> ≠ undefined ∧ refDef = ref.referencedDefinition0 ⇒

refDef.s-<formal context parameters> ≠ undefined ∧

isFormalContextParametersMatched0(ref,refDef)

If there is a <formal context parameters> item in the type reference, this shall be the same as the <formal context parameters> of the <referenced definition>. See clause 8.2 of [ITU‑T Z.103].

∀ref ∈ <block reference> ∪ <process reference> : ∀refDef ∈<referenced definition>:

let refDefInstances =

if refDef ∈ <block definition>

then refDef.s-<block heading>.s-<agent instantiation>.s-<number of instances>

else refDef.s-<process heading>.s-<agent instantiation>.s-<number of instances>

endif

in

refDef = ref.referencedDefinition0 ∧ refDefInstances ≠ undefined ∧ ref.s-<number of instances> ≠ undefined

⇒

refDefInstances = ref.s-<number of instances>

endlet // refDefInstances

If both the <number of instances> in the <agent instantiation> of an <agent definition> and the <number of instances> in the agent reference for the <agent definition> are specified, the two <number of instances> shall be equal. See clause 9 *Concrete grammar* of [ITU‑T Z.103].

∀ref1, ref2 ∈<referenced definition>:

ref1.entityName0= ref2.entityName0∧ ref1.entityKind0= ref2.entityKind0 ∧ ref1 ≠ ref2⇒

ref1.contextQualifier0≠ undefined∧ ref2.contextQualifier0≠ undefined∧

¬isPathItemMatched0 (ref1.contextQualifier0.s-<path item>-seq,

ref2.contextQualifier0.s-<path item>-seq ) ∧

¬isPathItemMatched0 (ref2.contextQualifier0.s-<path item>-seq,

ref1.contextQualifier0.s-<path item>-seq )

If two <referenced definition> items 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> is allowed to be omitted.

∀rd∈ <referenced definition>:

( rd∈<package definition>

∧ (∃ ref ∈ <package reference> : ref.s-<identifier>.refersto0 = rd ) )

⇒ rd.contextQualifier0 ≠ undefined

The <qualifier> in a <referenced definition> shall be present if the <referenced definition> is a <package definition> referenced from another context except if the <package definition> represents an outermost package definition.

∀ def ∈ EntityDefinition0: def ∉ <referenced definition> ⇒ def.contextQualifier0= undefined

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

Transformations

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

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

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

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

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

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

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

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

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

op= <operation reference>(\*,\*) =4=> adaptDefinition(op.referencedDefinition0, undefined)

The referenced definition is logically placed at the point of the reference to determine the properties of the system specification (see clause 7.3 of [ITU‑T Z.101]).

Auxiliary functions

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

referencedDefinition0(ref: Reference0 ): <referenced definition> =def

let eKind = ref.referenceKind0 in

let refName = ref.referenceName0 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.contextQualifier0, ref.surroundingScopeUnit0) then d

else

undefined

endif

else

undefined

endif

endlet

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

referencedBy0(rd: <referenced definition>): Reference0 =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-<name> = seq2[i].s-<name> ∧

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

The function adaptDefinition is used to adapt an inserted referenced definition: the qualifier is defined by context therefore is *undefined* in the syntax, and the number of instances is merged.

adaptDefinition(def:<referenced definition>, inst:<number of instances>):<referenced definition> =def

case def of

| <package definition>(uses, <package heading>(\*, name, intf), entities)

then mk-<package definition>(uses, mk-<package heading>(undefined, name, intf), entities)

| <internal procedure definition>(uses,

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

entities, body)

then mk-<internal procedure definition>(uses,

mk-<procedure heading>(h1, undefined, name, h2, h3, h4, h5, h6, h7),

entities, body)

| <block definition>(uses, <block heading>(\*, name, <agent instantiation>(inst1, addi)), body)

then mk-<block definition>(uses, mk-<block heading>(undefined, name,

mk-<agent instantiation>(mergeNumbers(inst, inst1), addi)), body)

| <process definition>(uses,

<process heading>(\*, name, <agent instantiation>(inst1, addi)), body)

then mk-<process definition>(uses, mk-<process heading>(undefined, name,

mk-<agent instantiation>(mergeNumbers(inst, inst1), addi)), body)

| <interface definition> then def

| <composite state definition>(uses, <composite state heading>(\*, name, p), body)

then mk-<composite state definition>(uses, mk-<composite state heading>(undefined, name, p), body)

| <system type definition>(uses, <system type heading>(\*, name, addi), body)

then mk-<system type definition>(uses, mk-<system type heading>(undefined, name, addi), body)

| <block type definition>(uses, <block type heading>(pre, \*, name, addi), body)

then mk-<block type definition>(uses, mk-<block type heading>(pre, undefined, name, addi), body)

| <process type definition>(uses, <process type heading>(pre, \*, name, addi), body)

then mk-<process type definition>(uses, mk-<process type heading>(pre, undefined, name, addi), body)

| <composite state type definition>(compkind)

then

case compkind of

| <composite state type graph>(uses,<composite state type heading>(virt,\*,name,fp,vc, sp,afp),css,**\*)**

then mk-<composite state type definition>(uses,

mk-<composite state type heading>(virt,undefined,name,fp,vc, sp,afp),

css,

name) // composite state type graph

endcase

| <state aggregation type>(uses,<state aggregation type heading>(pre,\*,name,fp,vc, sp,afp),as,**\*)**

then mk-<state aggregation type>(uses,

mk-<state aggregation type heading>(pre,undefined,name,fp,vc, sp,afp),

as,

name) // state aggregation type

| <operation definition>(uses, <operation heading>(opkind, pre, \*, name, pars, res), entities, body)

then mk-<operation definition>(uses,

mk-<operation heading>(opkind, pre, undefined, name, pars, res),

entities,

body)

otherwise undefined

endcase

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

mergeNumbers(inst1: <number of instances>, inst2: <number of instances>):

<number of instances> =def

if inst1 = undefined then inst2

elseif inst2 = undefined then inst1

else

let ini1 = inst1.s-<initial number> in

let max1 = inst1.s-<maximum number> in

let lower1 = inst1.s-<lower bound> in

mk-<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,

if lower1 ≠ undefined then lower1 else inst2.s-<lower bound> endif

) // number of instances

endlet // lower1

endlet // max1

endlet // ini1

endif // both inst1 and inst2 defined

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

isFormalContextParametersMatched0(fcpl1: <name>\*, fcpl2: <name>\*): Boolean =def

if (fcpl1 = empty) then (fcpl2 = empty)

else

isFormalContextParameterMatched0(fcpl1.head, fcpl1.tail) ∧

isFormalContextParametersMatched0 (fcpl1.tail, fcpl2.tail)

endif

isFormalContextParameterMatched0(fcp1: <name>\*, fcp2: <name>\*): Boolean =def

if (fcp1∈<agent type context parameter>) then (fcp2∈<agent type context parameter>)

elseif (fcp1∈<agent context parameter>) then (fcp2∈<agent context parameter>)

elseif (fcp1∈<procedure context parameter>) then (fcp2∈<procedure context parameter>)

elseif (fcp1∈<remote procedure context parameter>) then (fcp2∈<remote procedure context parameter>)

elseif (fcp1∈<signal context parameter>) then (fcp2∈<signal context parameter>)

elseif (fcp1∈<variable context parameter>) then (fcp2∈<variable context parameter>)

elseif (fcp1∈<remotevariable context parameter>) then (fcp2∈<remotevariable context parameter>)

elseif (fcp1∈<timer context parameter list>) then (fcp2∈<timer context parameter list>)

elseif (fcp1∈<synonym context parameter list>) then (fcp2∈<synonym context parameter list>)

elseif (fcp1∈<sort context parameter>) then (fcp2∈<sort context parameter>)

elseif (fcp1∈<compositestate type context parameter>)

then (fcp2∈<compositestatetype context parameter>)

elseif (fcp1∈<gate context parameter>) then (fcp2∈<gate context parameter>)

elseif (fcp1∈<interface context parameter list>) then (fcp2∈<interface context parameter list>)

endif

#### F2.2.4.4 Select definition

Concrete syntax

<select definition> ::

<Boolean<simple expression>

{ <agent definition>

| <agent type definition>

| <agent type reference>

| <block reference>

| <channel definition>

| <channel to channel connection>

| <composite state reference>

| <composite state type definition>

| <composite state type reference>

| <composite state definition>

| <data definition>

| <imported procedure specification>

| <imported variable specification>

| <macro definition>

| <operation definition>

| <package definition>

| <package reference>

| <procedure definition>

| <procedure reference>

| <process reference>

| <remote procedure definition>

| <remote variable definition>

| <select definition>

| <signal definition list>

| <signal list definition>

| <state machine>

| <state partition>

| <timer definition>

| <variable definition> }+

Transformations

<select definition>(cond, defs)

=7=> if value0(cond) = 0 then defs else empty endif

If the result of the <Boolean<simple expression> of a <select definition> is the predefined Boolean value false, the constructs contained in the <select definition> are not selected. In the other case, the constructs are selected. The <select definition> is deleted at transformation and is replaced by the contained selected constructs, if any. See clause 13.1 *Model* of [ITU‑T Z.102].

Auxiliary functions

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

value0(e:<simple expression>): Nat=def

if isSuperType1(getEntityDefinition1(predefinedId1("Boolean"), sort).s-Sort,

value1(e.simpleMapping).s-Result.s-Sort-reference-identifier)

then

if value1(e.simpleMapping).s-Literal-name = mk-Name("true") then 0 else 1 endif

else

value1(e.simpleMapping).s-Literal-natural

endif

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

simpleMapping(e:<simple expression>): Constant-expression =def

case e of

| <operand5>(undefined, <operator application>(ident, params)) then

mk-Operation-application(Mapping(ident), <simpleMapping(x) | x in params>)

| <operand5>(undefined, <literal identifier>(qual, name)) then

mk-Literal-identifier(Mapping (qual), Mapping (name))

| <operand5>(undefined, <conditional expression>(e1, e2, e3)) then

mk-Conditional-expression(simpleMapping (e1), simpleMapping (e2), simpleMapping (e3))

otherwise undefined

endcase

#### F2.2.4.5 Transition option

Concrete syntax

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

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

Transformations

t=<transition action items>(a, undefined)

provided a.last ∉ <transition option> ∧ t.parentAS0.parentAS0.parentAS0 ∈ <transition option> ∧

t.findContinueLabel ≠ undefined

=7=> <transition action items>(a,

<terminator>(undefined,<join>( t.findContinueLabel)))

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

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

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

<transition option>(q, <textual decision body>(answers, elsePart))

=7=> (let matching = { a.s-<transition> | a ∈ answers.toSet: q.value0 ∈ a.s-<answer> } in

if matching ≠ ∅ then matching.take

elseif elsePart ≠ undefined then elsePart.s-<transition>

else undefined

endif

endlet)

The <transition> of a <textual answer part> is selected if the <answer> contains the result of the <alternative question>. If no <answer> contains the result of the <alternative question>, the <transition> of the <else part> is selected.

If no <else part> is provided and none of the outgoing paths are selected, the selection is invalid.

On transformation each <transition> not selected is deleted, and the <transition option> is deleted and replaced by the selected <transition>.

### F2.2.5 Structural concepts

#### F2.2.5.1 Structural type definitions

##### F2.2.5.1.1 Agent types

Abstract syntax

Agent-type-definition :: Agent-type-name

Agent-kind

[ Agent-type-identifier ]

Agent-formal-parameter\*

Data-type-definition-set

Syntype-definition-set

Signal-definition-set

Timer-definition-set

Variable-definition-set

Agent-type-definition-set

Composite-state-type-definition-set

Procedure-definition-set

Agent-definition-set

Gate-definition-set

Channel-definition-set

State-machine

[ *Abstract* ]

Agent-kind = SYSTEM | BLOCK | PROCESS

Conditions on abstract syntax

∀d ∈ Agent-type-definition: (d.agentKind1 = PROCESS) ⇒

(∀ d1 ∈ Agent-type-definition∪Agent-definition: d1.parentAS1= d ⇒ d1. agentKind1= PROCESS)

The contained *Agent-definition*s 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 parameters> ] [<virtuality constraint>] <agent additional heading>

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

Transformations

a = <agent structure>(entities, <agent body>(startItem, items))

provided

a.parentAS0 ∈ <system type definition> ∪ <block type definition> ∪ <process type definition>

∧ inheritedAgentBody0(a.parentAS0) = undefined // no inherited agent body

=12=>

let tbcsn = newName // typebased composite state name

in

let atd = a.parentAS0

in

let afps = expandAgentFormalParams0(

case atd of

| <system type definition> then empty

| <block type definition> then atd.s-<block type heading>.

s-<agent type additional heading>.s-<agent additional heading>.s-<agent formal parameters>

| <process type definition> then atd.s-<process type heading>.

s-<agent type additional heading>.s-<agent additional heading>.s-<agent formal parameters>

endcase)

in

let cst =

mk-<composite state type definition>(

mk-<composite state type graph>(empty,

case atd of

| <system type definition> then

mk-<composite state type heading>(

undefined,

atd.s-<system type heading>.s-<qualifier>,

newName,

atd.s-<system type heading>.s-<agent type additional heading>.

s-<formal context parameters>,

undefined,

undefined,

empty)

| <block type definition> then

mk-<composite state type heading>(

atd.s-<block type heading>.s-<virtuality>,

atd.s-<block type heading>.s-<qualifier>,

if atd.specialization0 = undefined

then newName

else atd.specialization0.s-<type expression>.s-<identifier>.refersto0.s-<agent structure>

.s-<interaction>.s-<type based composite state>.s-<type expression>

.s-<identifier>.s-<name>

endif,

atd.s-<block type heading>.s-<agent type additional heading>.

s-<formal context parameters>,

undefined,

undefined,

afps)

| <process type definition> then

mk-<composite state type heading>(

atd.s-<process type heading>.s-<virtuality>,

atd.s-<process type heading>.s-<qualifier>,

if atd specialization0 = undefined

then newName

else atd specialization0.s-<type expression>.s-<identifier>.refersto0.s-<agent structure>

.s-<interaction>.s-<type based composite state>.s-<type expression>

.s-<identifier>.s-<name>

endif,

atd.s-<process type heading>.s-<agent type additional heading>.

s-<formal context parameters>,

undefined,

undefined,

afps)

endcase,

<composite state structure>(empty,

deriveGateEntities0(entities),

replaceInSyntaxTree0(atd.fullQualifierWithin0, cst.fullQualifierWithin0,

<composite state body>(

if startItem = undefined then empty

else < startItem.s-<start> >

endif,

items

) ) ) ) )

in

mk-<agent structure>( entities ⁀ cst,

mk-<interaction>(deriveChannelEntities0(entities,tbcsn),

mk-<typebased composite state>(tbcsn,

mk-<nextstate parameters>(

actualParamsNextstate0(afps), // <actual parameters> of <nextstate parameters>

undefined), // state entry point <name> of <nextstate parameters>

<type expression>(cst.identifier0, empty)

) ) )

endlet

endlet

endlet

endlet

The following paragraphs are derived from clause 8.1.1.1 *Model* of [ITU‑T Z.103].

If an <agent type definition> (a <system type definition>, <block type definition> or <process type definition>) has an <agent structure> that contains an <agent body> instead of an <interaction>, this is shorthand for an agent type that has no contained agents and a state machine with its type defined by the <agent body>.

A <composite state type definition> is derived from the <agent type definition>. This definition is given a heading with the same <virtuality>, an anonymous name, the same <formal context parameters> and same <agent formal parameters> as the <agent type definition>. The <agent body> is copied as the <composite state body> and any part of a <qualifier> in the body that refers to the enclosing agent type is changed to refer to the composite state type of the agent type. Each <textual gate definition> of the <agent type definition> is copied as a <textual gate definition> of the <composite state type definition> but omitting any <textual endpoint constraint> of the <gate constraint>. Each <textual interface gate definition> of the <agent type definition> is copied as a <textual interface gate definition> of the <composite state type definition> but omitting any <textual endpoint constraint>. A <composite state type reference> for the derived <composite state type definition> is placed in the <agent type definition>. In the <agent type definition>, the <agent body> is replaced by a <state machine> with a <typebased composite state> that has:

an anonymous unique name as the <composite state name>;

the list of names for the <agent formal parameters> of the agent type as the <actual parameters> in the <nextstate parameters> and with no **via** <state entry point> in the <nextstate parameters>;

the (anonymous) name of the composite state type as the <base type> in the <composite state type expression> and the list of names of the <formal context parameters> (if any) of the agent type as the <actual context parameters> in the <composite state type expression>.

Each <channel definition> of the <agent type definition> that had a <channel path> with one <channel endpoint> connected to a <gate> of the environment (**env**) and the other <channel endpoint> connected to the <agent body>, has the <channel endpoint> that was connected to the <agent body> changed to connect to the <gate> with the same name (as the gate of the environment) of the replacement <state machine>. If the <channel definition> has two <channel path> items, the change is made to both <channel path> items.

NOTE – [ITU‑T Z.103] describes (see text above) the insertion of a composite state type reference for the derived composite state type in the agent type, but such references are removed and replaced by the referenced definition (in this case the <composite state type definition> which is in the <interaction> that replaces the <agent body> of the <agent structure>).

If the <agent type definition> is defined as a specialization by inheriting another agent type (the supertype), the composite state type of the state machine of the supertype has to be **virtual** and the composite state type name is the same in both the supertype and subtype, because the copied <agent body> adds to the supertype. See clause 8.1.1.1 *Model* of [ITU‑T Z.103].

Auxiliary functions

The function deriveGateEntities0 derives the gates from the entities of an <agent structure> to insert as entities of the <composite state type definition> according to the clause 8.1.1.1 *Model* of [ITU‑T Z.103].

deriveGateEntities0(entities: <entity in agent>\*):<entity in composite state>\* =def

deriveGateEntity0(entities.head) ⁀

(if entities.length =1 then empty else deriveGateEntities0(entities.tail) endif)

The function deriveGateEntity0 given an <entity in agent>, if it is a gate returns a one element <entity in composite state> list to insert as an entity of <composite state type definition> according to the clause 8.1.1.1 *Model* of [ITU‑T Z.103]. For any other <entity in agent> an empty list is returned.

deriveGateEntity0(entity: <entity in agent>):<entity in composite state>\* =def

case entity of

| <textual gate definition>( g, enc, <gate constraint>(in,\*, sin,out,\*, sout )) then

< mk-<textual gate definition>( g, enc, mk-<gate constraint>(in,undefined, sin,out,undefined, sout )) >

| <textual gate definition>( g, enc, <gate constraint>(inout,\*, sinout)) then

< mk-<textual gate definition>( g, enc, mk-<gate constraint>(inout,undefined, sinout)) >

| < textual interface gate definition>(inout, interface, encoding, \*) then

< mk-<textual interface gate definition>(inout, interface, encoding,undefined) >

otherwise empty

endcase

The function deriveChannelEntities0 derives the channels from the entities of an <agent structure> to insert as entities of an <interaction> that replaces an <agent body> of the <agent structure> according to the clause 8.1.1.1 *Model* of [ITU‑T Z.103]. The <name> parameter is the name of the type based composite state.

deriveChannelEntities0(entities: <entity in agent>\*, tbcsn: <name>):<entity in composite state>\* =def

deriveChannelEntity0(entities.head, tbcsn) ⁀

(if entities.length =1 then empty else deriveChannelEntities0(entities.tail, tbcsn) endif)

The function deriveChannelEntity0 given an <entity in agent> if it is a gate returns a one element <entity in composite state> list to insert in the <interaction> that replaces an <agent body> of the <agent structure> according to the clause 8.1.1.1 *Model* of [ITU‑T Z.103]. For any other <entity in agent> an empty list is returned. The <name> parameter is the name of the type based composite state.

deriveChannelEntity0(entity: <entity in agent>, tbcsn: <name>):<entity in composite state>\* =def

case entity of

| <channel definition>(n, e, d,

<channel path>( <channel endpoint>(e1,g1), <channel endpoint>(e2,g2), sl), undefined)

then

if e1 = env

then

< mk-<channel definition>(n, e, d,

mk-<channel path>(mk-<channel endpoint> env,g1),mk-<channel endpoint>(tbcsn,g1), sl),

undefined) >

elseif e2 = env

< mk-<channel definition>(n, e, d,

mk-<channel path>(mk-<channel endpoint>(tbcsn,g2),mk-<channel endpoint>(env,g2), sl),

undefined) >

else empty

endif

| <channel definition>(n, e, d,

<channel path>( <channel endpoint>(e1,g1), <channel endpoint>(e2,g2), sl1),

<channel path>( <channel endpoint>(e2,g2), <channel endpoint>(e1,g1), sl2) )

then

if e1 = env

then

< mk-<channel definition>(n, e, d,

mk-<channel path>(mk-<channel endpoint>(env,g1),mk-<channel endpoint>(tbcsn,g1), s1l),

mk-<channel path>(mk-<channel endpoint>(tbcsn,g1),mk-<channel endpoint>(env,g1), s12) >

elseif e2 = env

< mk-<channel definition>(n, e, d,

mk-<channel path>(mk-<channel endpoint>(tbcsn,g2),mk-<channel endpoint>(env,g2), s1l),

mk-<channel path>(mk-<channel endpoint>(env,g2),mk-<channel endpoint>(tbcsn,g2), s12) >

else empty

endif

otherwise empty

endcase

The function actualParamsNextstate0 derives the <actual parameters> in the <nextstate parameters> from <agent formal parameters> for a type based composite state in the transform of <agent structure>. The <agent formal parameters> has to be expanded so that each <parameters of sort> contains only one <name>.

actualParamsNextstate0(afps: <agent formal parameters>):<actual parameters> =def

if afps = empty then empty

else

< mk-<operand5>(

mk-<variable access>(mk-<identifier>(empty,afps.head.s-<parameters of sort>.s-<name>-seq[1]))

) > ⁀

if afps.length =1 then empty

else actualParamsNextstate0(afps. tail)

endif

endif

The function expandAgentFormalParams0 expands an <agent formal parameters> so that each <parameters of sort> contains only one <name>.

expandAgentFormalParams0(afps: <agent formal parameters>):<agent formal parameters> =def

if afps = empty then empty

else

if afps.head.s-<parameters of sort>.s-<name>-seq.length = 1

then afps.head

else < afps.head.s-<aggregation kind>,

mk-<parameters of sort>( < afps.head.s-<parameters of sort>.s-<name>-seq[1] >,

afps.head.s-<parameters of sort>.s-<sort>) > ⁀

expandAgentFormalParams0(< afps.head.s-<aggregation kind>,

mk-<parameters of sort>( < afps.head.s-<parameters of sort>.s-<name>-seq.tail>,

afps.head.s-<parameters of sort>.s-<sort>

) >)

endif ⁀

if afps.length =1 then empty

else expandAgentFormalParams0(mk-<agent formal parameters>(afps.tail))

endif

endif

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

validOutputSignalSet0(d: <agent type definition>∪<agent definition>):Signal0 =def

d.localOutputSignalSet0∪ d.inheritedOutputSignalSet0

The function localOutputSignalSet0 gets the local output signal set of an <agent type definition> or an <agent definition>.

localOutputSignalSet0(d: <agent type definition>∪<agent definition>):Signal0 =def

{sid∈Signal0:

(∃gc∈<gate constraint>:

(gc.parentAS0∈<textual gate definition>)∧isDefinedIn0(gc.parentAS0, d)∧

(gc.direction0=out)∧(sid∈signalSet0(gc.s-<signal list item>-seq))) ∨

(∃cp∈<channel path>:

(cp.parentAS0∈<channel definition>)∧isDefinedIn0(cp.parentAS0, d)∧

(cp.destination0.s-env≠undefined)∧(sid∈signalSet0(cp.s-<signal list item>-seq)))}

The function inheritedOutputSignalSet0 gets the inherited output signal set of an <agent type definition> or an <agent definition>.

inheritedOutputSignalSet0(d: <agent type definition>∪<agent definition>):Signal0 =def

if d.specialization0 = undefined then ∅

else d.specialization0.s-<type expression>.baseType0.validOutputSignalSet0

endif

The function inheritedAgentBody0 gets an inherited agent body (if any) of an <agent type definition>. If the agent type is not specialized the result is the undefined value. If the base type of the agent has an agent body this is returned, otherwise it returns the inherited agent body (if any) of the base type. Note that if the base type has an agent body and inherits an agent body, the two bodies are not merged.

inheritedAgentBody0 (atd: <system type definition> ∪ <block type definition> ∪ <process type definition>):

<agent body> =def

let specialization =

(case atd of

| <system type definition> then atd.s-<system type heading>

| <block type definition> then atd.s-<block type heading>

| <process type definition> then atd.s-<process type heading>

endcase).s-<agent type additional heading>.s-<agent additional heading>.s-<specialization>

in

if specialization = undefined then undefined

else

let baseType = specialization.s-<type expression>.baseType0 in

let baseTypeBody = baseType.s-<agent structure>.s-<agent body> in

if baseTypeBody ≠ undefined then baseTypeBody

else inheritedAgentBody0 (baseType)

endif

endlet // baseTypeBody

endlet // baseType

endif

endlet // specialization

##### F2.2.5.1.2 System type

Concrete syntax

<system type definition> ::

<package use clause>\* <system type heading> <agent structure>

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

Conditions on concrete syntax

∀fcp∈<formal context parameter>:

(fcp.surroundingScopeUnit0∈<system type definition>) ⇒

(fcp∉<agent context parameter>∪<variable context parameter list>∪<timer context parameter list>)

A <formal context parameter> of <formal context parameters> of <agent type additional heading> of a <system type definition> shall not be an <agent context parameter>, <variable context parameter list> or <timer context parameter list>. See clause 8.1.1.2 *Concrete* *grammar* of [ITU‑T Z.102].

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

The <agent type additional heading> in a <system type definition> shall not include <agent formal parameters>. See clause 8.1.1.2 *Concrete* *grammar* of [ITU‑T Z.101].

Mapping to abstract syntax

| <system type definition>(\*, <system type heading>(\*, name,

<agent type additional heading>(\*, \*, <agent additional heading>(spec, params))),

<agent structure>(entities, <interaction>(defs, sm))) then

let entitiesSet = { Mapping(entities[i]:i ∈ 1..entities.length} in

let defSet = { Mapping(defs[i]:i ∈ 1..defs.length} in

mk-Agent-type-definition(Mapping(name),

SYSTEM,

Mapping(spec), // specialization

Mapping(params), // agent formal parameters

{ dtd ∈ entitiesSet: (dtd ∈ Data-type-definition)} ∪

{ implicitInterface1(atd) : (atd ∈ entitiesSet ∧ atd ∈ Agent-type-definition) } ∪

{ implicitInterface1(adf) : (adf ∈ entitiesSet ∧ adf ∈ Agent-definition) } ∪

{ implicitInterface1(at) : (at ∈ defSet ∧ at ∈ Agent-type-definition) } ∪

{ implicitInterface1(ad) : (ad ∈ defSet ∧ adf ∈ Agent-definition) }∪

{ implicitInterface1(sm) } ∪

{ choiceForGate1(tgd) : (tgd in entities ∧ tgd ∈ <textual gate definition> } ∪ // choices for *Gate-defs*,

{ choiceForChannel1(ch) : (ch in defs ∧ ch ∈ <channel definition> } // choices for *Chan-defs*,

{ sdf ∈ entitiesSet: (sdf ∈ Syntype-definition)},

{ sgd ∈ entitiesSet: (sgd ∈ Signal-definition)},

{ tdf ∈ entitiesSet: (tdf ∈ Timer-definition)},

{ vdf ∈ entitiesSet: (vdf ∈ Variable-definition)},

{ atd ∈ entitiesSet: (atd ∈ Agent-type-definition)} ∪{ at ∈ defSet: ( at ∈ Agent-type-definition) },

{ std ∈ entitiesSet: (std ∈ Composite-state-type-definition)},

{ pdf ∈ entitiesSet: (pdf ∈ Procedure-definition)},

{ adf ∈ entitiesSet: (adf ∈ Agent-definition)} ∪{ ad ∈ defSet: ( ad ∈ Agent-definition) },

{ gdf ∈ entitiesSet: (gdf ∈ Gate-definition)},

{ cdf ∈ entitiesSet: (cdf ∈ Channel-definition)} ∪{ ch ∈ defSet: ( ch ∈ Channel-definition) },

Mapping(sm)) // State machine

endlet // defSet

endlet // entitiesSet

The implicitly defined interface for a state machine is defined in the same scope unit as the state machine that defined it: that is, inside the agent type. See clause 12.1.2 *Concrete* *grammar* of [ITU‑T Z.101].

##### F2.2.5.1.3 Block type

Concrete syntax

<block type definition> ::

<package use clause>\* <block type heading> <agent structure>

<block type heading> ::

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

Transformations

*The special stopping condition property of* set\_v *and* get\_v *procedures mentioned below is covered by the dynamic semantics.* The two text paragraphs following the transformation code are derived from clause 9.2 *Model* of [ITU‑T Z.102]. The first paragraph is an outline of the transform, and the second paragraph gives the detail. The transform of block definition into a typebased block definition and a block type follows the pattern for transforming agent definitions and is given in clause F2.2.6.1.

atd = <agent type definition>(packages,

heading,

<agent structure>(entities,

<interaction>(defsAndRefs,

<typebased composite state>( statename, undefined, <type expression>(st, empty))

)

)

)

provided

heading ∈ (<system type heading> ∪ <block type heading>)

∧ (∃vardef in entities : vardef ∈ <variable definition>)

∧ (∀vardef in entities : vardef ∈ <variable definition> ⇒ // one variable per definition

vardef.s-<variables of sort>.length =1 ∧ vardef.s-<variables of sort>-seq[1].s-<name>-seq.length =1)

=16=>

let st1 = newName in

<agent type definition>(packages,

heading,

<agent structure>(

< e in entities: e ∉ <variable definition> > // delete variable definitions

⁀ //add remote procedure definitions for set

< mk-<remote procedure definition>(

mk-<name>("#set\_"+ entities[i].s-<variables of sort>-seq[1].s-<name>-seq[1].s-Token),

mk-<procedure signature>( // sort of variable formal parameter, no result

< mk-<formal parameter>(in, entities[i].s-<variables of sort>-seq[1].s-<sort>) >, undefined

) // procedure signature - sort of variable formal parameter, no result

) //remote procedure definition

| i in 1..entities.length : entities[i] ∈ <variable definition> >

⁀ //add remote procedure definitions for get

< mk-<remote procedure definition>(

mk-<name>("#get\_"+ entities[i].s-<variables of sort>-seq[1].s-<name>-seq[1].s-Token),

mk-<procedure signature>(

undefined, mk-<result>(entities[i].s-<variables of sort>-seq[1].s-<sort>)

) // procedure signature - no formal parameters, sort of variable result

) //remote procedure definition

| i in 1..entities.length : entities[i] ∈ <variable definition> >

⁀ //add state type st1

let stdef = getEntityDefinition0 (st, state type) in

mk-<composite state type definition>(

if stdef.s-<composite state type graph> ≠ undefined

then

mk-<composite state type graph>(

stdef.s-<composite state type graph>.s-<package use clause>-seq,

let stheading = stdef.s-<composite state type graph>.s-<composite state type heading> in

mk-<composite state type heading>(

stheading.s-<virtuality>,

empty, // no qualifier

st1, //name

stheading.s-<formal context parameters>,

stheading.s-<virtuality constraint>,

st, // <specialization>, inherit st

stheading.s-<agent formal parameters>

), // <composite state type heading>

endlet // stheading

mk-<composite state type structure>(

empty, // no connection points added

//add variables and exported procedure definitions for set and get

< mkVarSetGet0(entities[i]) | i ∈ 1..entities.length : entities[i] ∈ <variable definition> >,

mk-<composite state body>(empty,empty) // no starts added, no states or free actions added

),

st1 // optional name

) // <composite state type graph>

else// stdef is<state aggregation type>

mk-<state aggregation type>(

stdef.s-<state aggregation type>.s-<package use clause>-seq,

let saheading = stdef.s-<state aggregation type>.s-<state aggregation state type heading> in

mk-<state aggregation type heading>(

saheading.s-<type preamble>

empty, // no qualifier

st1, //name

saheading.s-<formal context parameters>,

saheading.s-<virtuality constraint>,

st, // <specialization>, inherit st

saheading.s-<agent formal parameters>

), // <state aggregation type heading>

endlet // saheading

mk-<aggregation structure>(

//add variables and exported procedure definitions for set and get

< mkVarSetGet0(entities[i]) | i ∈ 1..entities.length : entities[i] ∈ <variable definition> >,

empty // no change to partitions

), // <aggregation structure>

st1 // optional name

) // <state aggregation type>

endif // stdef is <composite state type graph> (else <state aggregation type>)

endlet // stdef

mk-<interaction>(defsAndRefs,

mk-<typebased composite state>( statename, undefined, mk-<type expression>(st1, empty))

) // agent structure

) // agent type definition

endlet // st1

and // transform variable assignments in contained agents.

ass = <assignment>(var, expr)

provided

isAncestorAS0(atd, ass) // is an assignment defined in atd – has to be for global variable to be visible

∧ ¬(isAncestorAS0(getEntityDefinition0(st, state type),ass)) // but ass not in the state type st

∧ var ∉ (<indexed variable> ∪ <field variable>) //removed at step 8, so var should be just identifier

∧ getEntityDefinition0(var, variable) in entities //one of the global block variables

=>

mk-<call statement>(

mk-<remote procedure call body>(

mk-<identifier>( var.fullQualifier0, mk-<name>("#set\_"+ var.s-<name>.s-Token)), //remote proc id

< expr >, // actual parameters

empty // communication constraints – empty list

)

) // call statement

and // transform variable accesses in contained agents.

va = <variable access>(vid)

provided

isAncestorAS0(atd, va) // is a variable access defined in atd – has to be for global variable to be visible

∧ ¬(isAncestorAS0(getEntityDefinition0(st, state type),va)) // but variable access not in the state type st

∧ vid ∈ <identifier>

∧ getEntityDefinition0(vid, variable) in entities //one of the global block variables

=>

mk-<value returning procedure call> (

mk-<remote procedure call body>(

mk-<identifier>( vid.fullQualifier0, mk-<name>("#get\_"+ vid.s-<name>.s-Token)), //remote proc id

empty, // actual parameters

empty // communication constraints – empty list

)

) // call statement

A block with a global variable definition (a <variable definition> as an <entity in agent> in the <agent structure> of the block/system type of the block/system) has a state machine that is interpreted concurrently with agents in the block/system. Access from contained agents in the block/system to a global variable of the block is covered by two implicitly defined remote procedures for setting and getting the data item associated with the variable. These procedures are provided by the state machine of the block.

A block type bt with global variables is transformed by moving the global variables (each <variable definition> as an <entity in agent>) from the <agent structure> of bt to a new (anonymously named) state type st1 for bt that replaces the existing state type st for the state machine of bt. The state type st1 has a <specialization> "inherits st adding", and adds each <variable definition> deleted from the <agent structure> of bt to a <composite state type definition> of st1. For each variable v in b, st1 has two exported procedures with anonymous implicit names, but here are called 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 of bt 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 bt, as these are transformed into procedures local to the calling agents. There is no ambiguity between the remote procedure calls for different instances block type bt, because each instance has implicit remote procedure definitions for both procedures. The set\_v and get\_v procedures have a special property that means they are handled in the stopping condition of instances of bt. This transformation takes place after replacing agent definitions with typebased agent definitions and transforming context parameters, and before the transforming of remote procedures.

Mapping to abstract syntax

| <block type definition>(\*, <block type heading>(\*,\*, name,

<agent type additional heading>(\*,\*,<agent additional heading>(spec,params))),

<agent structure>(entities, <interaction>(defs, sm))) then

let entitiesSet = { Mapping(entities[i]:i ∈ 1..entities.length} in

let defSet = { Mapping(defs[i]:i ∈ 1..defs.length} in

mk-Agent-type-definition(Mapping(name),

BLOCK,

Mapping(spec), // specialization

Mapping(params), // agent formal parameters

{ dtd ∈ entitiesSet: (dtd ∈ Data-type-definition)} ∪

{ implicitInterface1(atd) : (atd ∈ entitiesSet ∧ atd ∈ Agent-type-definition) } ∪

{ implicitInterface1(adf) : (adf ∈ entitiesSet ∧ adf ∈ Agent-definition) }} ∪

{ implicitInterface1(at) : (at ∈ defSet ∧ at ∈ Agent-type-definition) } ∪

{ implicitInterface1(ad) : (ad ∈ defSet ∧ adf ∈ Agent-definition) }∪

{ implicitInterface1(sm) } ∪

{ choiceForGate1(tgd) : (tgd in entities ∧ tgd ∈ <textual gate definition> } ∪ // choices for *Gate-defs*,

{ choiceForChannel1(ch) : (ch in defs ∧ ch ∈ <channel definition> } // choices for *Chan-defs*,

{ sdf ∈ entitiesSet: (sdf ∈ Syntype-definition)},

{ sgd ∈ entitiesSet: (sgd ∈ Signal-definition)},

{ tdf ∈ entitiesSet: (tdf ∈ Timer-definition)},

{ vdf ∈ entitiesSet: (vdf ∈ Variable-definition)},

{ atd ∈ entitiesSet: (atd ∈ Agent-type-definition)} ∪{ at ∈ defSet: ( at ∈ Agent-type-definition) },

{ std ∈ entitiesSet: (std ∈ Composite-state-type-definition)},

{ pdf ∈ entitiesSet: (pdf ∈ Procedure-definition)},

{ adf ∈ entitiesSet: (adf ∈ Agent-definition)} ∪{ ad ∈ defSet: ( ad ∈ Agent-definition) },

{ gdf ∈ entitiesSet: (gdf ∈ Gate-definition)},

{ cdf ∈ entitiesSet: (cdf ∈ Channel-definition)} ∪{ ch ∈ defSet: ( ch ∈ Channel-definition) },

Mapping(sm)) // State machine

endlet // defSet

endlet // entitiesSet

Auxiliary functions

The function mkVarSetGet0 makes a local copy of the global block variable given by the *vardef* parameter and the exported *set* and *get* procedures.

mkVarSetGet0(vardef: <variable definition>, st1: <name>):

(<variable definition>∪<internal procedure definition>)\* =def

let varname = vardef.s-<variables of sort>-seq[1].s-<name>-seq[1] in

let qualInSt1 = vardef.fullQualifier0 ⁀ mk-<path item>( state type, st1) in

< // make a sequence

vardef, //variable definition – no modification needed as does not contain qualifier

mk-<internal procedure definition>( //set

empty, //package use clause

mk-<procedure heading>(

mk-<procedure preamble>(

mk-<type preamble>(finalized, undefined), // finalized, not abstract

mk-<exported>(

mk-<identifier>(vardef.fullQualifier0,

mk-<name>("#set\_"+ varname.s-Token)

) // identifier

) // exported

), // procedure preamble

empty, // defined in context – no qualifier needed

mk-<name>("#set\_"+ varname.s-Token),

undefined, // no formal context parameters

undefined, // defined in context – no virtuality constrant

undefined, // defined in context – no specialization

< mk-<formal variable parameters>(

in, // parameter kind

mk-<parameter aggregation>(mk-<aggregation kind>(part)), // aggregation

mk-<parameters of sort>(< mk-<name>("v") >, vardef.s-<variables of sort>-seq[1].s-<sort>)

) // formal variable parameters

>, // formal variable parameters list

undefined // result

), // <procedure heading>

empty, // entity in procedure list

finalized, // virtuality

mk-<statements>(

<

mk-<non terminating statement>(

undefined, // no connector name before (assignment) non-terminating statement

mk-<assignment statement>(mk-<assignment>(

mk-<identifier>( qualInSt1, varname)// variable defined by local vardef

mk-<operand5>(undefined, mk-<variable access>(

mk-<identifier>( qualInSt1 ⁀

mk-<path item>(procedure,

mk-<name>(

"#set\_"+ varname.s-Token)

), //name

), //path item

mk-<name>("v")) // identifier – v procedure parameter

)) //variable access , operand5 expression

)) // assignment, assignment statement

) // non terminating statement

>, // non terminating statement list

undefined, // no connector name before terminating statement

mk-<return statement>(mk-<return body>(undefined)) //return

) // statements

), // <internal procedure definition> set

mk-<internal procedure definition>( //get

empty, //package use clause

mk-<procedure heading>(

mk-<procedure preamble>(

mk-<type preamble>(finalized, undefined), // finalized, not abstract

mk-<exported>(

mk-<identifier>(vardef.fullQualifier0,

mk-<name>("#get\_"+ varname.s-Token)

) // identifier

) // exported

), // procedure preamble

empty, // defined in context – no qualifier needed

mk-<name>("#get\_"+ varname.s-Token),

undefined, // no formal context parameters

undefined, // no virtuality constrant

undefined, // no specialization

empty, // no formal variable parameters

mk-<result>(

mk-<parameter aggregation>(mk-<aggregation kind>(part)), // aggregation

mk-<parameters of sort>(< mk-<name>("v") >, vardef.s-<variables of sort>-seq[1].s-<sort>)

) // formal variable parameters

>, // formal variable parameters list

undefined // result

), // <procedure heading>

empty, // list of entities in procedure

finalized, // virtuality

mk-<statements>(

undefined, // no connector name before terminating statement

mk-<return statement>(

mk-<return body>(

mk-<operand5>(undefined, mk-<variable access>(

mk-<identifier>( qualInSt1, varname)// variable defined by local vardef

)) //variable access , operand5 expression

) // return body

) //return statement

) // statements

) // <internal procedure definition> get

> // end of sequence

endlet // qualInSt1

endlet // varname

The function isGlobalBlockVar0 returns true if the <variable> given is defined as one of the entities of a block (or system) type and the <variable> usage occurs within the block (or system), but not within the state machine actions of the block (or system) type. The function is used to check that such a global block <variable> is used in a <stimulus>, <timer communication constraint>, <encoded input> or <in choice> only in the state machine of the agent owning the variable.

isGlobalBlockVar0(var: <identifier>): Boolean =def

let su = surroundingScopeUnit0(getEntityDefinition0(var, variable)) in

let struct = su.s-<agent structure>.s-implicit in

let sma = // state machine or actions

if struct ∈ <interaction> then

let sm = struct.s-<interaction>.s-<state machine>

if sm ∈ <typebased composite state> then

getEntityDefinition0(

sm.s-<typebased composite state>.s-<type expression>.s-<base type>,state type)

else // sm ∈ <composite state list item>

take({ cs ∈ <composite state definition> :

(surroundingScopeUnit0(cs) = su)

∧ ( (cs.s-<composite state graph>.s-<composite state heading>.s-<name>

= sm.s-<composite state list item>.s-<name>)

∨ (cs.s-<state aggregation>.s-<composite state heading>.s-<name>

= sm.s-<composite state list item>.s-<name>)

)

})

endif

endlet // sm

else // struct ∈ <agent body>

struct.s-<agent body>

endif

in

su ∈ (<system definition> ∪ <system type definition> ∪ <block definition> ∪ <block type definition>)

∧ isAncestorAS0(su, var) // use within scope surrounding var def

∧ ¬(isAncestorAS0(sma, var)) // but use not within state machine actions of block/system surrounding var def.

endlet // sma

endlet // struct

endlet // su

##### F2.2.5.1.4 Process type

Concrete syntax

<process type definition> ::

<package use clause>\* <process type heading> <agent structure>

<process type heading> ::

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

Mapping to abstract syntax

| <process type definition>(\*, <process type heading>(\*,\*, name,

<agent type additional heading>(\*,\*,<agent additional heading>(spec,params))),

<agent structure>(entities, <interaction>(defs, sm))) then

let entitiesSet = { Mapping(entities[i]:i ∈ 1..entities.length} in

let defSet = { Mapping(defs[i]:i ∈ 1..defs.length} in

mk-Agent-type-definition(Mapping(name),

PROCESS,Mapping(spec), // specialization

Mapping(params), // agent formal parameters

{ dtd ∈ entitiesSet: (dtd ∈ Data-type-definition)} ∪

{ implicitInterface1(atd) | (atd ∈ entitiesSet ∧ atd ∈ Agent-type-definition) } ∪

{ implicitInterface1(adf) | (adf ∈ entitiesSet ∧ adf ∈ Agent-definition) }} ∪

{ implicitInterface1(at) : (at ∈ defSet ∧ at ∈ Agent-type-definition) } ∪

{ implicitInterface1(ad) : (ad ∈ defSet ∧ adf ∈ Agent-definition) }∪

{ implicitInterface1(sm) } ∪

{ choiceForGate1(tgd) : (tgd in entities ∧ tgd ∈ <textual gate definition> } ∪ // choices for *Gate-defs*,

{ choiceForChannel1(ch) : (ch in defs ∧ ch ∈ <channel definition> } // choices for *Chan-defs*,

{ sdf ∈ entitiesSet: (sdf ∈ Syntype-definition)},

{ sgd ∈ entitiesSet: (sgd ∈ Signal-definition)},

{ tdf ∈ entitiesSet: (tdf ∈ Timer-definition)},

{ vdf ∈ entitiesSet: (vdf ∈ Variable-definition)},

{ atd ∈ entitiesSet: (atd ∈ Agent-type-definition)} ∪{ at ∈ defSet: ( at ∈ Agent-type-definition) },

{ std ∈ entitiesSet: (std ∈ Composite-state-type-definition)},

{ pdf ∈ entitiesSet: (pdf ∈ Procedure-definition)},

{ adf ∈ entitiesSet: (adf ∈ Agent-definition)} ∪{ ad ∈ defSet: ( ad ∈ Agent-definition) },

{ gdf ∈ entitiesSet: (gdf ∈ Gate-definition)},

{ cdf e ∈ entitiesSet: (cdf ∈ Channel-definition)} ∪{ ch ∈ defSet: ( ch ∈ Channel-definition) },

Mapping(sm)) // State machine

endlet // defSet

endlet // entitiesSet

##### F2.2.5.1.5 Composite state type

Abstract syntax

Composite-state-type-definition :: State-type-name

[ Composite-state-type-identifier ]

Composite-state-formal-parameter\*

State-entry-point-definition-set

State-exit-point-definition-set

Gate-definition-set

Data-type-definition-set

Syntype-definition-set

Composite-state-type-definition-set

Variable-definition-set

Procedure-definition-set

{ Composite-state-graph | State-aggregation-node }

[ Abstract]

Conditions on abstract syntax

∀d∈Composite-state-type-definition: d.s-Gate-definition-set ≠ ∅⇒

(∃sd∈State-machine:getEntityDefinition1(sd.s-Composite-state-type-identifier, state type) = d)

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

Concrete syntax

<composite state type definition> ::

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

<composite state type graph> ::

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

<composite state type heading> ::

[<virtuality>] <qualifier> <composite state type<name>

[ <formal context parameters> ] [<virtuality constraint>]

[<specialization>] <agent formal parameters>

<state aggregation type> ::

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

<state aggregation type heading> ::

<type preamble> <qualifier> <composite state type<name>

[ <formal context parameters> ] [<virtuality constraint>]

[<specialization>] <agent formal parameters>

Transformations

cstd.defaultEntryPoint

provided

cstd ∈ <composite state type definition> ∧

cstd.defaultEntryPoint = undefined

=11=>

(defaultEntryPoint \{( cstd, undefined )} ) ∪ {( cstd, newName)}

cstd.defaultExitPoint

provided

cstd ∈ <composite state type definition> ∧

cstd.defaultExitPoint = undefined

=11=>

(defaultEntryPoint \{( cstd, undefined )} ) ∪ {( cstd, newName)}

Mapping to abstract syntax

| <composite state type definition>(<composite state type graph>(\*,

<composite state type heading>(\*, \*, name, \*, \*, parent, params),

<composite state structure>(conns, entities, <composite state body>(starts, items))) then

let startSet = { Mapping(starts[i]):i ∈ 1..starts.length} in

let itemSet = { Mapping(items[i]):i ∈ 1..items.length} in

let entitiesSet = { Mapping(entities[i]):i ∈ 1..entities.length} in

mk-Composite-state-type-definition(Mapping(name),

Mapping(parent),

Mapping(params),

Mapping(< cin : (cin in conns) ∧ (cin ∈ <state entry points>) >).bigSeq.toSet,

Mapping(< cout : (cout in conns) ∧ (cout ∈ <state exit points>) >).bigSeq.toSet,

{ gdf ∈ entitiesSet: (gdf ∈ Gate-definition)},

{ dtd ∈ entitiesSet: (dtd ∈ Data-type-definition)} ∪

{ choiceForGate1(tgd) : (tgd in entities ∧ tgd ∈ <textual gate definition> } // choices for *Gate-defs*,

{ sdf ∈ entitiesSet: (sdf ∈ Syntype-definition)},

{ std ∈ entitiesSet: (std ∈ Composite-state-type-definition)},

{ vdf ∈ entitiesSet: (vdf ∈ Variable-definition)},

{ pdf ∈ entitiesSet: (pdf ∈ Procedure-definition)},

mk-Composite-state-graph(

mk-State-transition-graph(< s ∈ startSet: s ∈ State-start-node >.head,

{ nsn ∈ startSet: nsn ∈ Named-start-node },

{ sn ∈ itemSet: sn ∈ State-node },

{ fa ∈ itemSet: fa ∈ Free-action}), // State-transition-graph

< pe ∈ entitiesSet: (pe ∈ Procedure-definition ∧ pe.s-Procedure-name = "entry") >.head,

< px ∈ entitiesSet: (px ∈ Procedure-definition ∧ px.s-Procedure-name = "exit") >.head

) // Composite-state-graph

) // Composite state type definition

endlet // entitiesSet

endlet // itemSet

endlet // startSet

| <composite state type definition>(<state aggregation type>(\*,

<composite state type heading>(\*, \*, name, \*, \*, parent, params), // = <state aggregation type heading>

<aggregation structure>(conns, body))) then

let entitiesSet = { Mapping(conns[i]):i ∈ 1..conns.length ∧ conns[i] ∈ <entity in state aggregation>} in

let csstd = mk-Composite-state-type-definition(Mapping(name),

Mapping(parent),

Mapping(params),

Mapping(< cin : (cin in conns) ∧ (cin ∈ <state entry points>)>).bigSeq.toSet,

Mapping(< cout : (cout in conns) ∧ (cout ∈ <state exit points>)>).bigSeq.toSet,

{ gdf ∈ entitiesSet: (gdf ∈ Gate-definition)},

{ dtd ∈ entitiesSetMapping(entities).toSet: (dtd ∈ Data-type-definition)},

{ sdf ∈ entitiesSet: (sdf ∈ Syntype-definition)},

{ std ∈ entitiesSet: (std ∈ Composite-state-type-definition)},

{ vdf ∈ entitiesSet: (vdf ∈ Variable-definition)},

{ pdf ∈ entitiesSet: (pdf ∈ Procedure-definition ∧ pdf.s-Procedure-name ∉ {"entry","exit"})},

mk-State-aggregation-node(

{ mk-State-partition(Mapping(b.s-<typebased state partition heading>.s-<name>), // partition name

Mapping(b.s-<typebased state partition heading>.s-<type expression>), // inherits (if any)

{ Mapping(c):c in body∧c ∈ <state partition connection entry>∧c.s2-<identifier>=b.identifier0 }

∪ // this union is the Connection-definition-set

{ Mapping(c):c in body∧c ∈ <state partition connection exit>∧c.s2-<identifier>=b.identifier0 }

) // State-partition

: (b in body) ∧ (b ∈ <textual typebased state partition def>) }, // State-partition-set

head(< pe in Mapping(entitiesSet):

pe ∈ Procedure-definition ∧ pe.s-Procedure-name = "entry" >),

head(< px in Mapping(entitiesSet):

px ∈ Procedure-definition ∧ px.s-Procedure-name = "exit") >)

) // State-aggregation-node

) in // Composite-state-type-definition

let unusedSigs = inputSignalSet1(cstd) \

**U** { inputSignalSet1(sp) : sp ∈ cstd.s-State-aggregation-node.s-State-partition} in

if unusedSigs = ∅

then cstd

else // add a state partition to the State-aggregation-node to consume the excess symbols

let agg = cstd.s-State-aggregation-node in

let cstname = Mapping(*newName*) in

let sName = Mapping(*newName*) in

let stEntry = mk-State-start-node (mk-Transition(empty, sNode)) in

let stExit = mk-Action-return-node()in

let opId = // further study is needed to define an operation that ANDs all the

// identifiers representing the non-implicit state partition

let sNode = mk-State-node(sName, ∅,

{ mk-*Input-node*(undefined, sig, undefined*,* undefined, undefined, undefined,

mk-Transition(empty, mk-Terminator(mk-Named-nextstate(sn)))):

sig∈ unusedSigs }, // Input-node-set

∅, // no Spontaneous-transitions

{mk-Continuous-signal(mk-Operation-application(opId, params)),

undefined, mk-Transition(empty,mk-Terminator(stExit)))

}, //Continuous-signal-set

undefined // no State-timer

) in

mk-Composite-state-type-definition( // revise Composite-state-type-definition to deal with unused signals

cstd.s-State-type-name

cstd.s-Composite-state-type-identifier

cstd.s-Composite-state-formal-parameter-seq

cstd.s-State-entry-point-definition-set

cstd.s-State-exit-point-definition-set

cstd.s-Gate-definition-set

cstd.s-Data-type-definition-set

cstd.s-Syntype-definition-set

cstd.s-Composite-state-type-definition-set ∪

{ mk-Composite-state-type-definition( // Composite-state-type-definition for unused signals

cstname, undefined, empty, { stEntry }, { stExit },∅,∅,∅,∅,∅,∅,

mk-Composite-state-graph( // this is the part that effectively discards unused signals

mk-State-transition-graph( stEntry, ∅, { sNode }, ∅, ), // State-transition-graph

undefined*,* // no *Entry-procedure-definition*

undefined // no *Exit-procedure-definition*

) // Composite-state-graph

undefined // Abstract

) } // Composite-state-type-definition

cstd.s-Variable-definition-set ∪

{ mk-Variable-definition(part.s-State-partition.s-Name,

predefinedId1("Boolean"), PART, Mapping(expr)) : part ∈ agg.s-State-partition-set

} // add an implicit Boolean variable for each partition

cstd.s-Procedure-definition-set,

mk-State-aggregation-node(

agg.s-State-partition-set ∪{ mk-State-partition(newName, undefined, ∅) },

agg.s-Entry-procedure-definition,

agg.s-Exit-procedure-definition

), // State-aggregation-node

cstd.s-Abstract

) // Composite-state-type-definition

endlet // sNode

endlet // stExit

endlet // stEntry

endlet // sName

endlet // cstname

endlet // agg

endif

endlet // unusedSigs

endlet // cstd

endlet // entitiesSet

#### F2.2.5.2 Type expression

Concrete syntax

<type expression> :: <base type> <actual context parameter list>

<base type> = <identifier>

Conditions on concrete syntax

∀te∈<type expression>:

te.actualContextParameterList0 ≠ empty ⇒ isParameterizedType0(te.baseType0)

It is valid to have <actual context parameters> if and only if <base type> denotes a parameterized type. See 8.1.2 *Concrete grammar* of [ITU‑T Z.102].

Transformations

te.anonymousTypeName

provided

te ∈ <type expression> ∧

te.s-<actual context parameter>-seq ≠ empty ∧

te.parentAS0 ∉ <specialization> ∧

te.anonymousTypeName = undefined

=11=>

(anonymousTypeName\{( te, undefined )} ) ∪ {( te, newName)}

te=<type expression>(id, params)

provided

te.s-<actual context parameter>-seq ≠ empty ∧

te.parentAS0 ∉ <specialization> ∧

te.anonymousTypeName ≠ undefined

=11=>

mk-<type expression>(

mk-<identifier>(te.surroundingQualifier0,

te.anonymousTypeName),

empty)

and

te.surroundingScopeUnit0.getEntities0 // entities in nearest surrounding scope unit

=>

let basetype = id.refersto0 in

let fcpl = basetype.FormalContextParameterList0 in

te.surroundingScopeUnit0.getEntities0 ⁀

< replaceContextParameters0(fcpl, params,

createNewType0(te.anonymousTypeName, basetype,

getUnboundFormalContextParameterList0(fcpl, params) ) ) >

endlet // fcpl

endlet // basetype

A <type expression> yields either the type identified by the identifier of <base type> in cases where 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>. See clause 8.1.2 *Model* of [ITU‑T Z.102].

Mapping to abstract syntax

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

Auxiliary functions

The function anonymousTypeName is used to store the name of the anonymous type created by applying actual context parameters to a base type of a type expression.

controlled anonymousTypeName: <type expression> → <name>

initially ∀ te ∈ <type expression>: te.anonymousTypeName = undefined

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

isDirectSubType0(ed: EntityDefinition0, td: TypeDefinition0): Boolean=def

∃te∈<type expression>: te. parentAS0=ed. specialization0∧td=te.baseType0

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

isSubtype0(sub: EntityDefinition0, sup: TypeDefinition0 ): Boolean=def

isDirectSubType0(sub, sup) ∨

(∃ttd∈ TypeDefinition0 : isSubtype0(sub, ttd) ∧ isSubtype0(ttd, sup))

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

baseType0(te: <type expression>): TypeDefinition0 =def

let bt = te.**s**-<identifier> in

getEntityDefinition0(bt, idKind0(bt))

endlet

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

isParameterizedType0(td: TypeDefinition0 ): Boolean=def

(td.formalContextParameterList0 ≠ empty)

Get the formal context parameter list of a type definition.

formalContextParameterList0(td: TypeDefinition0 ): <formal context parameter>\* =def

td.inheritedFormalContextParameterList0⁀ td.localFormalContextParameterList0

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

inheritedFormalContextParameterList0(td: TypeDefinition0 ): <formal context parameter>\* =def

let sp=td.specialization0 in

if sp = undefined then empty else

case sp of

| <interface specialization> then

< getUnboundFormalContextParameterList0(tel.baseType0.formalContextParameterList0,

tel.actualContextParameterList0)

| tel in sp.s-<type expression>-seq >

otherwise

let fcpl=sp.s-<type expression>.baseType0.formalContextParameterList0 in

let acpl=sp.s-<type expression>.actualContextParameterList0 in

getUnboundFormalContextParameterList0(fcpl, acpl)

endlet

endcase

endif

endlet

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

getUnboundFormalContextParameterList0(fcpl: <formal context parameter>\*,

acpl: <actual context parameter>\*): <formal context parameter>\* =def

if (fcpl = empty) then empty

else

if (acpl.head= undefined) then < fcpl.head > else empty endif

⁀getUnboundFormalContextParameterList0(fcpl.tail, acpl.tail)

endif

The function completeFormalContextParameter0 inserts the <identifier> for the original formal context parameter for the unbound ones.

completeFormalContextParameter0(fcpl: <name>\*, acpl: <actual context parameter list>):

<actual context parameter list> =def

if (fcpl = empty) then empty

else

if (acpl.head= undefined) then < mk-<identifier>( undefined, fcpl.head) > else <acpl.head> endif

⁀completeFormalContextParameter0(fcpl.tail, acpl.tail)

endif

The function actualContextParameterList0 gets the actual context parameter list of a type expression. Each item in the list corresponds to [ <actual context parameter> ]; that is, the item is either undefined or an <actual context parameter>.

actualContextParameterList0(te: <type expression>): <actual context parameter list> =def

te.s-<actual context parameter list>

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

localFormalContextParameterList0(td: TypeDefinition0 ): <formal context parameter>\* =def

let fcps =take({fcps∈<agent type additional heading> ∪ <composite state type heading> ∪

<procedure heading> ∪ <signal definition> ∪ <data type heading> ∪

<interface heading>: fcps.surroundingScopeUnit0 = td}) in

< fcpl.formalContextParameterSublist0 | fcpl in fcps.**s**-<formal context parameter>-seq >

**endlet**

Expand a formal context parameter to a list of formal context parameters so that each item in the list contains just one name.

formalContextParameterSublist0(fcp: <formal context parameter>):<formal context parameter>\* =def

**case** fcp **of**

| <agent context parameter> then < fcp >

| <agent type context parameter> then < fcp>

| <compositestate type context parameter> then < fcp >

| <gate context parameter> then < fcp >

| <interface context parameter list>(names) then

if names.length = 1 then < fcp >

else < mk-<interface context parameter list>(names.head) > ⁀

formalContextParameterSublist0(< mk-<interface context parameter list>(names.tail) >)

endif

| <procedure context parameter> then < fcp >

| <remote procedure context parameter> then <fcp >

| <remotevariable context parameter list> (items (names, vc)) then

if items.length = 1 then

if names.length = 1 then < fcp >

else

< mk-<remotevariable context parameter list>(

mk-<remotevariable contextparameter names>(names.head, vc)) > ⁀

formalContextParameterSublist0(

< mk-<remotevariable context parameter list>(

mk-<remotevariable contextparameter names>(names.tail, vc)) > )

endif

else

if names.length = 1 then

< mk-<remotevariable context parameter list>(

mk-<remotevariable contextparameter names>(names.head, vc)) > ⁀

formalContextParameterSublist0(items.tail)

else

< mk-<remotevariable context parameter list>(

mk-<remotevariable contextparameter names>(names.head, vc)) > ⁀

formalContextParameterSublist0(

< mk-<remotevariable context parameter list>(

mk-<remotevariable contextparameter names>(names.tail, vc)) >

⁀ items.tail)

endif

endif

| <signal context parameter list>(names) then

if names.length = 1 then < fcp >

else < mk-<signal context parameter list>(names.head) > ⁀

formalContextParameterSublist0(< mk-<signal context parameter list>( names.tail) >)

endif

| <sort context parameter> then < fcp >

| <synonym context parameter list>(names) then

if names.length = 1 then < fcp >

else < mk-<synonym context parameter list>(names.head) > ⁀

formalContextParameterSublist0(< mk-<synonym context parameter list>( names.tail) >)

endif

| <timer context parameter list>(names) then

if names.length = 1 then < fcp >

else < mk-<timer context parameter list>(names.head) > ⁀

formalContextParameterSublist0(< mk-<timer context parameter list>( names.tail) >)

endif

| <variable context parameter list>(names) then

if names.length = 1 then < fcp >

else < mk-<variable context parameter list>(names.head) > ⁀

formalContextParameterSublist0(< mk-<variable context parameter list>(names.tail) >)

endif

otherwise empty

**endcase**

Replace the formal context parameters by actual context parameters.

replaceContextParameters0(fcpl:<formal context parameter>\*, acpl:<actual context parameter>\*,

orig:DefinitionAS0): DefinitionAS0 =def

if fcpl = empty then orig

else replaceContextParameters0(fcpl.tail, acpl.tail,

if acpl.head = undefined then orig else replaceContextParameter0(fcpl.head, acpl.head, orig) endif )

endif

Replace one formal context parameter by an actual context parameter.

replaceContextParameter0(fcp:DefinitionAS0, acp:<actual context parameter>, orig:DefinitionAS0):

DefinitionAS0 =def

replaceInSyntaxTree0(fcp, acp, orig)

Create a new type with the specified formal context parameters.

createNewType0(n: <name>, orig: DefinitionAS0, fcpl: <formal context parameter list> ): DefinitionAS0 =def

case orig of

| <system type definition>(use,

<system type heading>(q, \*, <agent type additional heading>(\*, virt, add)), body)

then <system type definition>(use,

<system type heading>(q, n, <agent type additional heading>( fcpl, virt, add)), body)

| <block type definition>(use,

<block type heading>(pre, q, \*, <agent type additional heading>(\*, virt, add)), body)

then <block type definition>(use,

<block type heading>(pre, q, n, <agent type additional heading>( fcpl, virt, add)), body)

| <process type definition>(use,

<process type heading>(pre, q, \*, <agent type additional heading>(\*, virt, add)), body)

then <process type definition>(use,

<process type heading>(pre, q, n, <agent type additional heading>( fcpl, virt, add)), body)

| <composite state type definition>(use,

<composite state type heading>(v, q, \*, \*, c, spec, par), body)

then <composite state type definition>(use,

<composite state type heading>(v, q, n, fcpl, c, spec, par), body)

| <data type definition>(use, pre, <data type heading>(k, \*, \*, v), spec, body)

then <data type definition>(use, pre, <data type heading>(k, n, fcpl, v), spec, body)

| <internal procedure definition>(use, <procedure heading>(pre, q, \*, \*, c, spec, par, res), ent, body)

then <internal procedure definition>(use,

<procedure heading>(pre, q, n, fcpl, c, spec, par, res), ent, body)

| <interface definition>(use, virt, <interface heading>(\*, \*, v), spec, ent, l)

then <interface definition>(use, virt, <interface heading>(n, fcpl, v)), spec, ent, l)

| <signal definition>( pre, \*, v, spec, l)

then <signal definition>( pre, n, fcpl, v, spec, l)

otherwise undefined

endcase

#### F2.2.5.3 Definitions based on types

Concrete syntax

<textual typebased agent definition> =

<textual typebased system definition>

| <textual typebased block definition>

| <textual typebased process definition>

Conditions on concrete syntax

∀ad∈<textual typebased agent definition>: ∀te∈<type expression>:

(te.parentAS0.parentAS0= ad) ⇒

(∃s∈<start>: (s∈te.baseType0.startSet0)∧(s.s-<name> = undefined))

The agent type denoted by <base type> in the type expression of a <textual typebased agent definition> shall contain an unlabelled start transition in its state machine. See clause 9 *Concrete grammar* of [ITU‑T Z.101].

##### F2.2.5.3.1 System definition based on system type

Concrete syntax

<textual typebased system definition>::

<typebased system heading>

<typebased system heading> ::

<system<name> <system<type expression>

Mapping to abstract syntax

| <textual typebased system definition>(<typebased system heading>(name,<type expression>(b,\*))) then

mk-Agent-definition(Mapping(name), mk-Number-of-instances(1,1,0), Mapping(b))

##### F2.2.5.3.2 Block definition based on block type

Concrete syntax

<textual typebased block definition> ::

<typebased block heading>

<typebased block heading> ::

<block<name> <number of instances> <block<type expression>

Mapping to abstract syntax

| <textual typebased block definition>

(<typebased block heading>(name, inst, <type expression>(b,\*))) then

mk-Agent-definition(Mapping(name), Mapping(inst), Mapping(b))

##### F2.2.5.3.3 Process definition based on process type

Concrete syntax

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

<typebased process heading> ::

<process<name> <number of instances> <process<type expression>

Mapping to abstract syntax

| <textual typebased process definition> (name, inst, <type expression>(b,\*)) then

mk-Agent-definition(Mapping(name), Mapping(inst), Mapping(b))

##### F2.2.5.3.4 Composite state definition based on composite state type

Concrete syntax

<typebased composite state> ::

<composite state<name> <nextstate parameters> <composite state<type expression>

<textual typebased state partition def> =

<typebased state partition heading>

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

Mapping to abstract syntax

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

#### F2.2.5.4 Abstract type

Abstract syntax

Abstract :: {}

Concrete syntax

<abstract> :: ()

Conditions on concrete syntax

∀pd∈<procedure definition>: isAbstractType0(pd) ⇒

¬(∃pc∈<procedure call>: pd = pc.calledProcedure0)

An abstract procedure shall not be called.

NOTE − The above text is derived from "An abstract type shall not be instantiated" (see below); a procedure is a type and calling a procedure instantiates the procedure.

∀ad∈<textual typebased agent definition>: ∀te∈<type expression>:

te.parentAS0.parentAS0= ad ⇒ ¬isAbstractType0(te.baseType0)

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

NOTE − The above text is derived from "An abstract type shall not be instantiated" (see below); a typebased agent is an instantiation of the agent type.

∀td∈TypeDefinition0 : isAbstractType0(td) ⇒

¬(∃d∈<textual typebased agent definition> ∪ <textual typebased state partition def> ∪

<typebased composite state>:∃te∈<type expression>:

((te.parentAS0.parentAS0= d)∨(te.parentAS0= d))∧ (te.baseType0=td))

An abstract type shall not be instantiated. See clause 8.1.3 *Semantics* of [ITU‑T Z.102]).

NOTE − The abstract property of a type is not inherited; therefore instantiation of a subtype of an abstract data type is permitted, if the subtype is not itself abstract.

Mapping to abstract syntax

| <abstract> then Abstract

Auxiliary functions

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

isAbstractType0(td: TypeDefinition0 ): Boolean=def

(∃ab∈<abstract>: ab.surroundingScopeUnit0 = td)

∨ (∃te ∈<type expression>: te.surroundingScopeUnit0 = td ∧

getUnboundFormalContextParameterList0(te.baseType0.formalContextParameterList0,

te.actualContextParameterList0) ≠ empty)

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

calledProcedure0(pc: <procedure call>∪<value returning procedure call>

∪<procedure call body>∪<remote procedure call body>):

<procedure definition>=def

case pc of

| <procedure call>∪<value returning procedure call> then

let t = pc.s-<procedure call body>.s-implicit in

if t∈<identifier> then getEntityDefinition0(t, procedure)

else t.baseType0 // t∈<type expression>

| <procedure call body> then

let t = pc.s-implicit in

if t∈<identifier> then getEntityDefinition0(t, procedure)

else t.baseType0 // t∈<type expression>

| <remote procedure call body> then getEntityDefinition0(pc, remoteprocedure)

otherwise undefined

endcase

#### F2.2.5.5 Type reference

Type references do not have semantics in SDL‑2010.

#### F2.2.5.6 Gate

Abstract syntax

Gate-definition :: Gate-name

[ Encoding-rules ]

In-signal-identifier-set

Out-signal-identifier-set

In-signal-identifier = Signal-identifier

Out-signal-identifier = Signal-identifier

∀gd ∈ Gate-definition:

**let** supertype = gd.parentAS1.baseType1 **in**

supertype = undefined

∨ (∀stgd ∈ (supertype.**s**-Gate-definition-set):

(((gd.**s**-Gate-name = stgd.**s**-Gate-name) ∧ (stgd.**s**-Encoding-rules ≠ undefined))

⇒ (gd.**s**-Encoding-rules = stgd.**s**-Encoding-rules))

**endlet** // supertypeid

The *Encoding-rules* associated with a *Gate-definition* of a type based on a supertype shall specify the same set of *Encoding-rules* as the *Encoding-rules* of the corresponding gate definition in the supertype if that gate has *Encoding-rules*.

Concrete syntax

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

<textual gate definition> ::

<gate> [ <encoding rules> ] <gate constraint>

<textual interface gate definition> ::

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

<gate> = <gate<name>

<gate constraint> ::

**in** [ <textual endpoint constraint> ] [ <signal list> ]

[**out** [ <textual endpoint constraint> ] [ <signal list> ] ]

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

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

<as gate> :: <gate<identifier>

NOTE – The derived AS0 for any SDL/PR <gate constraint> with **out** followed by **in** is the AS0 <gate constraint> with **in** followed by **out**.

Conditions on concrete syntax

∀gd∈<textual gate definition>: ∀gc∈<gate constraint>:

(gc.parentAS0= gd) ⇒

(**let** td = gd.surroundingScopeUnit0 **in**

(∃td1∈TypeDefinition0 :

td1 = getEntityDefinition0(gc.**s**-<textual endpoint constraint>, td.entityKind0)) endlet)

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

∀gc ∈ <gate constraint>:(∀ce ∈ <channel endpoint>:(∀ce1 ∈ <channel endpoint>:(

(gc.parentAS0.**s**-<gate> = ce.**s**-<gate>) ∧

(ce ≠ ce1) ∧ (ce.parentAS0= ce1.parentAS0) ∧ (ce.parentAS0∈<channel path>) ∧

(gc.parentAS0∈<textual gate definition>) ) ) )⇒

( **let** td = getEntityDefinition0(gc.**s**-<textual endpoint constraint>,

gc.surroundingScopeUnit0.entityKind0) **in**

∃td1 ∈ EntityDefinition0:

((td1= ce1.channelEndpointReferTo0) ∧ ((td = td1) ∨ isSubtype0(td1, td)) )

endlet) ∧

(ce.parentAS0.**s**-<signal list item>-seq.signalSet0 ⊆ gc.**s**-<signal list item>-seq.signalSet0)

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

∀tbd∈<textual typebased block definition> ∪<textual typebased process definition>:

∀te∈<type expression>: (te.parentAS0.parentAS0= tbd) ⇒

(**let** td = te.baseType0 **in**

(td.channelDefinitionSet0 ≠ ∅) ⇒

(∀gc∈<gate constraint>:∀sig ∈ Signal0:

(gc.parentAS0∈td.gateDefinitionSet0) ∧

(sig∈gc.s-<signal list item>-seq.signalSet0) ⇒

(∃cp∈<channel path>:

(cp.parentAS0∈td.channelDefinitionSet0) ∧

((gc.direction0=in)⇒

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

(cp.origination0.s-implicit=env )) ∧

((gc.direction0=out)⇒

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

(cp.destination0.s-implicit=env)) ∧

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

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

∀gc, gc1∈<gate constraint>:

(gc ≠ gc1) ∧ (gc.parentAS0= gc1.parentAS0) ⇒

(gc.**s**-<textual endpoint constraint> = gc1.**s**-<textual endpoint constraint>) ∧

((gc.direction0**=**out) ∧ (gc1.direction0**=**in)∨(gc.direction0**=**in)∧(gc1.direction0**=**out))

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

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

(**let** td = gd.surroundingScopeUnit0 **in**

∃td1∈TypeDefinition0 : ∃gd1∈<textual gate definition>:

isSubtype0(td, td1) ∧ (gd1∈ td1.gateDefinitionSet0) ∧

(gd1.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,ec1∈<textual endpoint constraint>:

isSubtype0(ec.surroundingScopeUnit0, ec1.surroundingScopeUnit0) ∧

(ec.parentAS0∈<gate constraint>) ∧ (ec1.parentAS0∈<gate constraint>) ∧

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

(ec1.parentAS0.parentAS0∈<textual gate definition>) ∧

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

(((ec.direction0**=**out) ∧ (ec1.direction0**=**in))∨((ec.direction0**=**in) ∧ (ec1.direction0**=**out))) ∧

(ec.parentAS0.parentAS0.**s**-<gate>= ec1.parentAS0.parentAS0.**s**-<gate>) ⇒

(**let** td = getEntityDefinition0(ec.**s**-<identifier>, ec.surroundingScopeUnit0.entityKind0) in

let td1 = getEntityDefinition0(ec1.s-<identifier>, ec1.surroundingScopeUnit0.entityKind0) **in**

(td = td1) ∨ isSubtype0(td, td1) endlet)

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

∀gd∈<textual interface gate definition>:∃td∈<interface definition>:

td = getEntityDefinition0(gd.**s**-<identifier>, interface)

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

Transformations

t= <textual interface gate definition>(out, id=<identifier>(\*,n))

=8=>

mk-<textual gate definition>(n, undefined, mk-<gate constraint>(out, undefined, id))

t= <textual interface gate definition>(in, id=<identifier>(\*,n))

=8=>

mk-<textual gate definition>(n, undefined, mk-<gate constraint>(in, undefined, id))

<textual interface gate definition> is shorthand for a <textual gate definition> having the name of the interface as <gate name> and the <interface identifier> as the <signal list>. See clause 8.1.4 *Model* of [ITU‑T Z.103].

gd.asGateName

provided

gd ∈ <textual gate definition>

∧ gd.asGateName = undefined

=8=>

(asGateName \{( gd, undefined )} ) ∪ {( gd, newName )}

NOTE – Each <textual gate definition> defines a *Data-type-definition* of a choice data type with a unique anonymous *Sort* name stored in asGateName.

<as gate>(gid)

provided

gid.refersto0.asGateName ≠ undefined

=8=>

mk-<type expression>(mk-<identifier>(gid.**s**-<qualifier>, gid.refersto0.asGateName), undefined)

An <as gate> represents the sort of the choice data type introduced by the identified gate definition. See clause 12.1 *Concrete grammar* of [ITU‑T Z.104].

Mapping to abstract syntax

| tgd = <textual gate definition>(name, \*,

<gate constraint>(in, \*, inlist, out, \*, outlist) ) then

mk-Gate-definition(

Mapping(name),

Mapping(encodingRules0(tgd)),

{ Mapping(inlist[i]): i ∈ 1..inlist.length},

{ Mapping(outlist[i]): i ∈ 1..outlist.length})

| tgd = <textual gate definition>(name, \*,

<gate constraint>(in, \*, inlist, undefined,\*,\*) ) then

mk-Gate-definition(

Mapping(name),

Mapping(encodingRules0(tgd)),

{ Mapping(inlist[i]): i ∈ 1..inlist.length,

∅)

| tgd = <textual gate definition>(name, \*,

<gate constraint>(out, \*, outlist) ) then

mk-Gate-definition(

Mapping(name),

Mapping(encodingRules0(tgd)),

∅,

{ Mapping(outlist[i]): i ∈ 1..outlist.length)

If there is no set of <encoding rules> that is associated with a <textual gate definition> of a type based on a supertype, and there is a set of *Encoding-rules* for the gate in the supertype, the inherited gate has this set of *Encoding-rules*. If there is no set of *Encoding-rules* for the gate in the supertype, the presence and value of the inherited gate *Encoding-rules* is determined by the presence and value of the <encoding rules>. See clause 8.1.4 *Concrete* *grammar* of [ITU‑T Z.104].

NOTE – The mapping to a *Data-type-definition* for the choice data type defined by a <textual gate definition> in the following paragraph takes place by invoking the function choiceForGate1 (defined below), in the mapping to *Agent-type-definition* in F2.2.5.1.2, F2.2.5.1.3 and F2.2.5.1.4, or in the mapping of a <composite state type definition> with a <composite state structure> to a *Composite-state-type-definition* in F2.2.5.1.5.

A <textual gate definition> defines a *Data-type-definition* of a choice data type with a unique anonymous *Sort* name in the context that the *Gate-definition* is visible. A <basic sort> that is <as gate> where <gate identifier> identifies the *Gate-definition* represents this *Data-type-definition*. The *Data-type-definition* is equivalent to defining a choice data type with a <choice of sort> for each distinct *Signal-definition* identified by the *In-signal-identifier****-set*** and *Out-signal-identifier****-set***. Each <choice of sort> has as its <field sort> the data type denoted by <as signal> for the identified signal definition (the NULL sort for a signal without parameters; otherwise a structure data type with an anonymous unique name – for an identified signal signal\_id with parameters the <choice of sort> has a <field sort> with identity as denoted by **as** **signal** signal\_id). If an identified *Signal-definition* has a name distinct from any other identified distinct *Signal-definition*, the <choice of sort> has the same <field name> as the name of the corresponding identified signal. If an identified *Signal-definition* has the same name as another identified *Signal-definition*, the <choice of sort> has <field name> with the same name as the anonymous unique name for the <as signal> structure data type. See clause 8.1.4 *Concrete* *grammar* of [ITU‑T Z.104].

Auxiliary functions

The function origination0 gets the originating channel endpoint of a <channel path>.

origination0(p: <channel path>): <channel endpoint> =def

p.s-<channel endpoint>

The function destination0 gets the destination channel endpoint of a <channel path>.

destination0(p: <channel path>): <channel endpoint> =def

p.s2-<channel endpoint>

The function gateDefinitionSet0 gets the <gate in definition>-set defined in an <agent type definition>, a <composite state type definition>, a <agent definition> or a <composite state definition>.

gateDefinitionSet0(td: <agent type definition> ∪ <composite state type definition> ∪

<agent definition> ∪ <composite state definition> ): <gate in definition>-set =def

td.localGateDefinitionSet0 ∪ td.inheritedGateDefinitonSet0

The function localGateDefinitionSet0 gets the <gate in definition>-set defined by the <gate in definition> items of an <agent type definition>, a <composite state type definition>, a <agent definition> or a <composite state definition>.

localGateDefinitionSet0(td: <agent type definition> ∪ <composite state type definition> ∪

<agent definition> ∪ <composite state definition> ): <gate in definition>-set =def

{gd∈<gate in definition>: gd.surroundingScopeUnit0 = td}

The function inheritedGateDefinitonSet0 gets the <gate in definition>-set inherited by an <agent type definition>, a <composite state type definition>, a <agent definition> or a <composite state definition>.

inheritedGateDefinitonSet0(td: <agent type definition> ∪ <composite state type definition> ∪

<agent definition> ∪ <composite state definition> ): <gate in definition>-set =def

let sp=td.specialization0 in

if sp=undefined then ∅

else sp.s-<type expression>.baseType0.gateDefinitionSet0

endif

endlet

The function encodingRules0 determines the <encoding rules> item to be used for the mapping from a <textual gate definition> to a Gate-definition.

encodingRules0(tgd: <textual gate definition>): <encoding rules> =def

if tgd.s-<encoding rules> ≠ undefined then tgd.s-<encoding rules>

else

let stgates =

inheritedGateDefinitonSet0(

parentAS0ofKind(tgd,

<agent type definition>∪<composite state type definition>∪<agent definition> ∪<composite state definition>))

in

if stgates = ∅ then undefined

else

if (∃ stgd ∈ stgates: ((tgd.**s**-<name> = stgd.**s**-<name>) ∧ (stgd.**s**-<encoding rules> ≠ undefined))

then

take(stgd ∈ stgates: tgd.**s**-<name> = stgd.**s**-<name>).**s**-<encoding rules>

else undefined

endif

endif

endlet

endif

The function channelDefinitionSet0 gets the <channel definition> items defined in an <agent type definition> or an <agent definition>.

channelDefinitionSet0(td: <agent type definition> ∪ <agent definition> ): <channel definition>-set =def

td.localChannelDefinitionSet0∪ td.inheritedChannelDefinitionSet0

The function localChannelDefinitionSet0 gets the <channel definition> items defined locally (rather than inherited) in an <agent type definition> or an <agent definition>.

localChannelDefinitionSet0(td: <agent type definition> ∪ <agent definition> ):

<channel definition>-set =def

{cd∈<channel definition>: cd.surroundingScopeUnit0 = td}

The function inheritedChannelDefinitionSet0 gets the <channel definition> items defined inherited by an <agent type definition> or an <agent definition>.

inheritedChannelDefinitionSet0(td:<agent type definition> ∪ <agent definition> ):

<channel definition>-set =def

let sp=td.specialization0 in

if sp=undefined then ∅

else sp.s-<type expression>.baseType0.channelDefinitionSet0

endif

endlet // sp

The function signalSet0 produces the set of identifiers for entities of kind signal, timer, remote procedure or remote variable (that is, of domain Signal0) for the given <signal list item> list. A <signal list item> of kind signal, timer, remote procedure or remote variable has its identifier included in the set. A <signal list item> of kind interface is expanded to include the identifiers of each <signal list item> defined and used by the interface.

signalSet0(sl:<signal list item>\* ): Signal0=def

case sl.head.idKind0 of

| {signal, timer, remote procedure, remote variable} then {sl.head} ∪ sl.tail.signalSet0

| {interface} then

let id = getEntityDefinition0(sl.head, interface) in

id.usedSignalSet0 ∪ {sd.identifier0 | sd ∈ id.definedSignalSet0} ∪ sl.tail.signalSet0

endlet // id

otherwise ∅

endcase

The function choiceForGate1 produces the choice Data-type-definition that corresponds to a Gate-definition and is used when an <as gate> is given for a <basic sort>.

choiceForGate1(gd: <textual gate definition>): Data-type-definition =def

Mapping(mk-<data type definition>( mk-<type preamble>(undefined,undefined), // type preamble

mk-<data type heading>( gd.asGateName, // name for choice type

undefined, undefined, // no formal parameters or virtuality constraint

), // data type heading

mk-<data type definition body>( empty, // no entities in data type

mk-<choice definition>( undefined, // visibility

< mk-<choice of sort>( undefined, // visibility

mk-<field of kind>( part,

if sig.s-<name> ∉ { othersigs.s-<name>

: othersigs ∈ ((inoutSignals0(gd, in) ∪ inoutSignals0(gd, out)) \ sig )

then // unique signal name

sig.s-<name>

else // not unique signal name – name same as <as signal>

sig.asSignal.s-<name>// name of structure sort for signal

endif

), // <field of kind>

mk-<field sort>(

if sig.refersto0.s-implicit // <sort list> | <named fields sort list> | <sort<identifier>

= undefined // no sort

then predefinedId0("NULL")

else sig.asSignal // structure sort for signal

endif

) // <field sort>

) // <choice of sort> item

: sig ∈ (inoutSignals0(gd, in) ∪ inoutSignals0(gd, out))

> // <choice of sort> list

) <choice definition>(

), // data type definition body

undefined // default initialization

)) // Mapping of data type definition

The function asGateName associates each <textual gate definition> with a unique anonymous *Sort* name for the corresponding *Data-type-definition* of the choice data type invoked by <as gate>.

controlled asGateName: <textual gate definition>→ <name>

initially ∀ gd ∈ <textual gate definition>: gd.asGateName = undefined

#### F2.2.5.7 Context parameters

Concrete syntax

<actual context parameter list> = [ <actual context parameter> ]\*

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

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

<formal context parameter list> :: <formal context parameter>+

<formal context parameter> =

<agent context parameter>

| <agent type context parameter>

| <compositestate type context parameter>

| <gate context parameter>

| <interface context parameter list>

| <procedure context parameter>

| <remote procedure context parameter>

| <remotevariable context parameter list>

| <signal context parameter list>

| <sort context parameter>

| <synonym context parameter list>

| <timer context parameter list>

| <variable context parameter list>

Conditions on concrete syntax

∀fcp∈ FormalContextParameter0:∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ∧ (acp∈<primary>) ⇒

(fcp∈<synonym context parameter name> )

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

(∀te∈<type expression>: te.baseType0∉FormalContextParameter0) ∧

(∀fcp∈ FormalContextParameter0:

fcp.contextParameterAtleastDefinition0∉ FormalContextParameter0)

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

∀fcp∈<agent type context parameter>∪<agent context parameter>∪<procedure context parameter>∪

<signal context parameter name> ∪<sort context parameter>∪

<compositestate type context parameter> ∪<interface context parameter name>:

∀acp∈<actual context parameter>: isContextParameterCorresponded0(fcp, acp) ⇒

(∀td1∈TypeDefinition0 : (td1 = fcp.contextParameterAtleastDefinition0) ⇒

(td1∉FormalContextParameter0) ∧ ¬(isParameterizedType0(td1)) ∧

(∃td∈ TypeDefinition0 :

(td = getEntityDefinition0(acp, td1.entityKind0) ∧

((td = td1) ∨ isSubtype0(td, td1))))

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

<formal context parameter list>(fcpl)

provided

fcpl.length < fcpl.expandFormalContextParams0.length

=11=>

<formal context parameter list>( fcpl.expandFormalContextParams0)

The formal context parameter list is expanded so that each <formal context parameter> that is a list (<remotevariable context parameter list>, <signal context parameter list>, <synonym context parameter list>, <timer context parameter list> or <variable context parameter list>) contains only one named parameter.

<composite state type heading>(v, q, n, cPar, vc,

<specialization>(<type expression>(base, actPar)), p)

provided

getUnboundFormalContextParameterList0

(actParams, base.refersto0.s-<formal context parameter>) ≠ empty

=11=>

(let nCPar = cPar ⁀

getUnboundFormalContextParameterList0

(actParams, base.refersto0.s-<formal context parameter>)

in

let nActPar =

completeFormalContextParameter0(actParams, base.refersto0.s-<formal context parameter>)

in

<composite state type heading>

(v, q, n, nCPar, vc, <specialization>(<type expression>(b, nActPar)), p)

endlet

endlet)

<agent type additional heading>(cPar, vc,

<agent additional heading>(<specialization>(<type expression>(base, actPar)), p))

provided

getUnboundFormalContextParameterList0

(actParams, base.refersto0.s-<formal context parameter>) ≠ empty

=11=>

(let nCPar = cPar ⁀

getUnboundFormalContextParameterList0

(actParams, base.refersto0.s-<formal context parameter>)

in

let nActPar =

completeFormalContextParameter0(actParams, base.refersto0.s-<formal context parameter>)

in

<agent type additional heading>(nCPar, vc,

<agent additional heading>

(<specialization>(<type expression>(base, nActPar), undefined), p))

endlet

endlet)

<procedure heading>(v, q, n, cPar, vc, <specialization>(<type expression>(base, actPar)), p, r, x)

provided

getUnboundFormalContextParameterList0

(actParams, base.refersto0.s-<formal context parameter>)

≠ empty

=11=>

(let nCPar = cPar ⁀

getUnboundFormalContextParameterList0

(actParams, base.refersto0.s-<formal context parameter>)

in

let nActPar =

completeFormalContextParameter0(actParams, base.refersto0.s-<formal context parameter>)

in

<procedure heading>(v, q, n, nCPar, vc,

<specialization>(<type expression>(b, nActPar), undefined), p, r, x)

endlet

endlet)

<signal definition>(n, cPar, vc, <specialization>(<type expression>(base, actPar)), p)

provided

getUnboundFormalContextParameterList0

(actParams, base.refersto0.s-<formal context parameter>) ≠ empty

=11=>

(let nCPar = cPar ⁀

getUnboundFormalContextParameterList0

(actParams, base.refersto0.s-<formal context parameter>)

in

let nActPar =

completeFormalContextParameter0(actParams, base.refersto0.s-<formal context parameter>)

in

<signal definition>

(n,nCPar, vc, <specialization>(<type expression>(b, nActPar), undefined\*), p)

endlet

endlet)

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

Auxiliary functions

Expand a <formal context parameter> list so that each <formal context parameter> that is a list (<remotevariable context parameter list>, <signal context parameter list>, <synonym context parameter list>, <timer context parameter list> or <variable context parameter list>) contains only one named parameter.

expandFormalContextParams0(fcpl: <formal context parameter>+): <formal context parameter>+ =def

< fcpl.head >.formalContextParameterSublist0 ⁀

if fcpl.length = 1

then < >

else fcpl.tail.expandFormalContextParams0

endif

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

contextParameterAtleastDefinition0(fcp: FormalContextParameter0): EntityDefinition0 =def

case fcp of

| <agent type context parameter> then

if (fcp.**s**-<agent type constraint>∈ <identifier>) then

getEntityDefinition0(fcp.s-<agent type constraint>.<identifier>, agent type)

else undefined

endif

| <agent context parameter> then

if (fcp.**s**-<agent constraint>∈<agent constraint atleast>) then

getEntityDefinition0(fcp.s-<agent constraint>.**s**-<identifier>, agent type)

else if (fcp.**s**-<agent constraint>∈<agent constraint exactly>) then

getEntityDefinition0(fcp.s-<agent constraint>.**s**-<identifier>, agent type)

else undefined

endif

endif

| <procedure context parameter> then

if (fcp.**s**-<procedure constraint>∈<identifier>) then

getEntityDefinition0(fcp.s-<procedure constraint>, procedure)

else undefined

endif

| <compositestate type context parameter> then

if (fcp.**s**-<composite state type constraint>∈<identifier>) then

getEntityDefinition0(fcp.s-<composite state type constraint>,

state type)

else undefined

endif

| <signal context parameter name> then

if (fcp.**s**-<signal constraint>∈<identifier>) then

getEntityDefinition0(fcp.s-<signal constraint>, signal)

else undefined

endif

| <sort context parameter> then

if (fcp.**s**-<sort constraint>∈<sort>) then

getEntityDefinition0(fcp.s-<sort constraint>, type)

else undefined

endif

| <interface context parameter name> then

if (fcp.**s**-<interface constraint> ≠ undefined) then

getEntityDefinition0(fcp.s-<interface constraint>.**s**-<identifier>, interface)

else undefined

endif

otherwise undefined

endcase

##### F2.2.5.7.1 Agent type context parameter

Concrete syntax

<agent type context parameter> ::

<agent kind> <agent type<name> [<agent type constraint>]

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

<agent type constraint> =

<agent type<identifier> | <agent signature>

Conditions on concrete syntax

∀fcp∈<agent type context parameter>:∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ∧ (fcp.s-<agent type constraint> ≠ undefined) ⇒

(let td=getEntityDefinition0(acp, agent type) in

(∃td1∈<agent type definition>:

(td1=fcp.contextParameterAtleastDefinition0) ∧

(td.agentLocalFormalParameterList0=empty) ∧ isSubtype0(td,td1)) ∨

((td.entityKind0 = fcp.entityKind0) ∧

(let pl = td.agentFormalParameterList0 in

let sl = fcp.s-<agent type constraint>.agentSignatureSortList0 in

(pl.length = sl.length) ∧

(∀i∈1..pl.length: isSameSort0(pl[i].parentAS0.s-<sort>, sl[i])) endlet)) endlet)

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

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

Auxiliary functions

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

agentSignatureSortList0(as: <agent signature>):<sort>\* =def

as.s-<sort>-seq

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

agentFormalParameterList0(td: <agent type definition> ∪<agent definition> ∪

<composite state type definition>∪<composite state definition> ): <name>\* =def

td.agentLocalFormalParameterList0⁀ td.agentInheritedFormalParameterList0

agentLocalFormalParameterList0(td: <agent type definition> ∪<agent definition> ∪

<composite state type definition>∪<composite state definition> ): <name>\* =def

let fp=take({fp∈ <agent additional heading>∪

<composite state type heading>∪

<composite state heading>:

fp.surroundingScopeUnit0 = td}) in

if (fp.s-<agent formal parameters>= undefined)

then empty

else < psl.s-<name> | psl in fp.s-<agent formal parameters>.s-<parameters of sort>-seq >

endif

endlet

agentInheritedFormalParameterList0(td: <agent type definition> ∪<agent definition> ∪

<composite state type definition>∪<composite state definition> ): <name>\* =def

let sp=td.specialization0 in

if (sp = undefined) then empty

else sp.s-<type expression>.baseType0.agentFormalParameterList0

endif

endlet

Determine if a formal context parameter corresponds to an actual context parameter; that is, the position of the formal context parameter in the list of inherited and local formal context parameters for the base type of the type expression containing the actual context parameter is the same as the position of the actual context parameter in the list of actual context parameters.

NOTE – The actual context parameter list is allowed to be shorter than the formal parameter list when items are omitted from the end of the list.

isContextParameterCorresponded0( fcp: FormalContextParameter0,

acp: <actual context parameter>): Boolean=def

let te = parentAS0ofKind(acp, <type expression>) in

let fcpl = te.baseType0.formalContextParameterList0 in

let acpl = te.actualContextParameterList0 in

(fcpl.length ≥ acpl.length)∧

(∃!i∈1..acpl.length: (fcpl[i]=fcp)∧(acpl[i]=acp)

endlet

endlet

endlet

##### F2.2.5.7.2 Agent context parameter

Concrete syntax

<agent context parameter> ::

<agent kind> <agent<name> [<agent constraint>]

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

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

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

<agent signature> :: <sort list>

Conditions on concrete syntax

∀fcp∈<agent context parameter>: ∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ⇒

(let td = getEntityDefinition0(acp, agent) in

let td1 = fcp.contextParameterAtleastDefinition0 in

((fcp.s-<agent constraint>∈ <agent constraint atleast>)) ⇒

isSubtype0(td, td1) ∧ (td.agentLocalFormalParameterList0 = empty)) ∧

((fcp.s-<agent constraint> ∈ <agent constraint exactly>)) ⇒ (td = td1)) ∧

((fcp.s-<agent constraint> ∈ <agent signature>) ⇒

(getEntityDefinition0(acp, agent).entityKind0 = fcp.entityKind0) ∧

(let pl = td.agentFormalParameterList0 in

let sl = fcp.s-<agent constraint>.agentSignatureSortList0 in

(pl.length = sl.length) ∧

(∀i∈1..pl.length: isSameSort0(pl[i].parentAS0.s-<sort>, sl[i])) endlet)) endlet)

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

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

##### F2.2.5.7.3 Procedure context parameter

Concrete syntax

<procedure context parameter> ::

<procedure<name> <procedure constraint>

<procedure constraint> =

<procedure<identifier> | <procedure signature>

Conditions on concrete syntax

∀fcp∈<procedure context parameter>:∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ⇒

(**let** td = getEntityDefinition0(acp, procedure) **in**

((fcp.**s**-<procedure constraint>∈<identifier>) ⇒

(**let** td1 = fcp.contextParameterAtleastDefinition0 **in**

isDirectSubType0(td, td1) endlet)) ∧

((fcp.**s**-<procedure constraint>∈<procedure signature>) ⇒

(let fpl = td.procedureFormalParameterList0 in

let fpl1 = fcp.s-<procedure constraint>.**s**-<formal parameter>-seq in

(fpl.length = fpl1.length) ∧

(∀i∈1..fpl.length:

(fpl[i]. parentAS0.parentAS0.s-<parameter kind> = fpl1[i].s-<parameter kind>) ∨

(fpl[i]. parentAS0.parentAS0.s-<parameter kind>∈{in out, out}) ⇒

isSameSort0(fpl[i]. parentAS0.s-<sort>, fpl[i].s-<sort>))

endlet) ∧

(let result = td.s-<procedure heading>.**s**-<procedure result> in

let result1= fcp.s-<procedure constraint>.**s**-<result> in

((result = undefined) ∧ (result1= undefined)) ∨

((result≠ undefined) ∧ (result1 ≠ undefined)∧isSameResult0(result, result1))

endlet))

endlet)

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

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

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

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

##### F2.2.5.7.4 Remote procedure context parameter

Concrete syntax

<remote procedure context parameter>::

<remote procedure<name> <procedure signature>

Conditions on concrete syntax

∀fcp∈<remote procedure context parameter>: ∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp,acp) ⇒

(**let** ps = getEntityDefinition0(acp, remote procedure).**s**-<procedure signature> in

let ps1= fcp.**s**-<procedure signature> **in**

isSameProcedureSignature0(ps, ps1) endlet)

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

Auxiliary functions

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

isSameProcedureSignature0(ps, ps1: <procedure signature>): Boolean =def

**let** fpl = ps.procedureSignatureParameterList0 in

let fpl1 = ps1.procedureSignatureParameterList0 in

(fpl. length = fpl1.length) ∧

(∀i∈1..fpl.length:

isSameSort0(fpl[i].s-<sort>, fpl1[i].s-<sort>) ∧

(fpl[i].s-<parameter kind> = fpl1[i].s-<parameter kind>))∧

isSameResult0(ps.s-<result>, ps1.s-<result>))

endlet

##### F2.2.5.7.5 Signal context parameter

Concrete syntax

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

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

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

<signal signature> = <sort list>

Conditions on concrete syntax

∀fcp∈<signal context parameter name>: ∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ⇒

(**let** sd = getEntityDefinition0(acp, signal) **in**

((fcp.**s**-<signal constraint>∈<identifier>) ⇒

(**let** sd1 =fcp.contextParameterAtleastDefinition0 **in**

isSubtype0(sd, sd1) endlet)) ∧

((fcp.**s**-<signal constraint>∈<signal signature>) ⇒

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

endlet)

In the case of **atleast** <signal identifier>, an actual signal parameter shall identify a signal definition that is the same as, or a subtype of, the signal type of the constraint. In the case of <signal signature>, an actual signal parameter shall identify a signal definition that is compatible with the formal <signal signature>. A signal definition is compatible with the formal <signal signature> if each parameter of the signal definition has the same aggregation kind and sort as the corresponding parameter of the <signal signature>. See clause 8.3.5 *Concrete grammar* of [ITU‑T Z.102].

##### F2.2.5.7.6 Variable context parameter

Concrete syntax

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

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

<variable constraint> = <sort>

Conditions on concrete syntax

∀fcp∈<name>:∀acp∈<actual context parameter>:

(fcp.parentAS0∈<variable context parameter names> ) ∧

(fcp.parentAS0.parentAS0∈<variable context parameter list>) ∧

isContextParameterCorresponded0(fcp, acp) ⇒

(**let** vd = getEntityDefinition0(acp, variable) **in**

isSameSort0(fcp.parentAS0.**s**-<sort>, vd.**s**-<sort>)

endlet)

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

##### F2.2.5.7.7 Remote variable context parameter

Concrete syntax

<remotevariable context parameter list> ::

<remotevariable contextparameter names>+

<remotevariable contextparameter names> ::

<remote variable<name>+ <variable constraint>

Conditions on concrete syntax

∀fcp ∈ FormalContextParameter0:∀acp∈<actual context parameter>:

(fcp.parentAS0∈<remotevariable contextparameter names> ) ∧

(fcp.parentAS0.parentAS0∈<remotevariable context parameter list> ) ∧

isContextParameterCorresponded0(fcp, acp) ⇒

(**let** vd = getEntityDefinition0(acp, remote variable) **in**

isSameSort0(fcp.parentAS0.**s**-<sort>, vd.**s**-<sort>)

endlet)

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

##### F2.2.5.7.8 Timer context parameter

Concrete syntax

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

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

<timer constraint> = <sort list>

Conditions on concrete syntax

∀fcp∈<timer context parameter name>: ∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ⇒

(**let** td = getEntityDefinition0(acp, timer) **in**

(fcp.**s**-<timer constraint> ≠undefined) ⇒

isSameSortList0(fcp.s-<timer constraint>,td.**s**-<sort>-seq)

endlet)

An actual timer parameter shall 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. See clause 8.3.8 *Concrete grammar* of [ITU‑T Z.102].

##### F2.2.5.7.9 Synonym context parameter

Concrete syntax

<synonym context parameter list> ::

<synonym context parameter name>+

<synonym context parameter name> ::

<synonym<name> <synonym constraint>

<synonym constraint> = <sort>

Conditions on concrete syntax

∀fcp∈<synonym context parameter name>: ∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ⇒

isSameSort0(acp.staticSort0, fcp.s-<synonym constraint>)

An actual synonym parameter shall be a constant expression of the same sort as the sort of the constraint. See clause 8.3.9 *Concrete grammar* of [ITU‑T Z.102].

##### F2.2.5.7.10 Sort context parameter

Concrete syntax

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

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

<sort signature> ::

<literal signature>\* <operation signature>\* <operation signature>\*

Conditions on concrete syntax

∀fcp∈<sort context parameter>:∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ∧ (fcp.s-<sort constraint>≠undefined)⇒

(**let** td = getEntityDefinition0(acp, type).derivedDataType0 **in**

((fcp.**s**-<sort constraint>∈<sort>)⇒

(**let** td1 = fcp.contextParameterAtleastDefinition0 **in**

(td.s-<data type specialization>.**s**-<renaming>= undefined) ∧

isSubtype0(td, td1) endlet)) ∧

((fcp.**s**-<sort constraint>∈<sort signature>)⇒

(∀ls∈<literal signature>: (ls.parentAS0= fcp.s-<sort constraint>)⇒

∃ls1∈<literal signature>:

(ls1.surroundingScopeUnit0= td) ∧ isSameLiteralSignature0(ls,ls1)) ∧

(∀os∈<operation signature>:

(os.parentAS0= fcp.s-<sort constraint>)⇒

∃os1∈<operation signature>:

(os1.parentAS0∈<operator list> ∪ <method list>) ∧

(os1.surroundingScopeUnit0=td) ∧ isSameOperationSignature0(os, os1)))

endlet)

In the case of a <sort constraint> that is **atleast** <sort>, an actual sort parameter shall be the sort given by <sort> or a subtype (without renaming) of this sort. In the case of a <sort constraint> that is <sort signature>, an actual sort parameter shall be compatible with the formal sort signature. A sort is compatible with the formal sort signature if the literals of the sort include the literals in the formal sort signature and the operations defined by the data type that introduced the sort include the operations in the formal sort signature and the operations have the same signatures. See clause 8.3.10 *Concrete grammar* of [ITU‑T Z.102].

∀ls∈<literal signature>:

(ls.parentAS0∈<sort signature>) ∧ (ls.parentAS0.parentAS0∈<sort context parameter>) ⇒

(ls∉<named number>)

The <literal signature> of the <sort signature> shall not contain <named number>. See clause 8.3.10 *Concrete grammar* of [ITU‑T Z.102].

Auxiliary functions

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

derivedDataType0(sd: <syntype definition>∪<data type definition>):<data type definition> =def

if (sd∈<syntype definition>) then sd.parentDataType0

else sd

endif

Get the parent data type definition of a syntype definition.

parentDataType0(sd: <syntype definition>):<data type definition> =def

if (sd.**s**-<parent sort identifier> = undefined) then sd

else

**let** pd = getEntityDefinition0(sd.**s**-<parent sort identifier>, type) **in**

if (pd∈<data type definition>) then pd

else pd.parentDataType0

endif

endlet

endif

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

isSameLiteralSignature0(ls: <literal signature>, ls1: <literal signature>): Boolean=def

((ls∈<literal name>) ∧ (ls1∈<literal name>) ⇒ (ls = ls1)) ∧

((ls∈<named number>) ∧ (ls1∈<named number>) ⇒

(ls.s-<literal name> = ls1.s-<literal name>) ∧

(ls.s-<simple expression>.value0= ls1.s-<simple expression>.value0))

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

isSameOperationSignature0(os: <operation signature>, os1: <operation signature> ): Boolean=def

(os.virtuality0 = os1. virtuality0) ∧

(os.visibility0 = os1.visibility0) ∧

(os.s-implicit∈<operation name>⇒os.s-implicit = os1.s-implicit ) ∧

(let fpl = os.operationSignatureParameterList0,

fpl1 = os1.operationSignatureParameterList0 in

(fpl.length = fpl1.length) ∧

(∀i∈1..fpl.length:

(fpl[i].s-<parameter kind> = fpl1[i].s-<parameter kind>) ∧

(isSameSort0(fpl[i].s-<sort>, fpl1[i].s-<sort>)) ∧

(isSameResult0(os.s-<result>, os1.s-<result>)

endlet)

##### F2.2.5.7.11 Composite state context parameter

Concrete syntax

<compositestate type context parameter>::

<composite state type<name> [<composite state type constraint>]

<composite state type constraint> =

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

<composite state type signature> = <sort list>

Conditions on concrete syntax

∀fcp∈<compositestate type context parameter>: ∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp)∧

(fcp.**s**-<composite state type constraint>≠ undefined) ⇒

(**let** td = getEntityDefinition0(acp, state type) **in**

((fcp.**s**-<composite state type constraint>∈<identifier>) ⇒

(**let** td1 = fcp.contextParameterAtleastDefinition0 **in**

(td.agentLocalFormalParameterList0 = empty) ∧ isSubtype0(td, td1) endlet)) ∧

((fcp.**s-**<composite state type constraint>∈<composite state type signature>) ⇒

( **let** sl = fcp.s-<composite state type constraint> in

let pl = td.agentFormalParameterList0 **in**

(sl.length = pl.length) ∧

(∀i∈1..sl.length: isSameSort0(sl[i], pl[i].parentAS0.**s**-<sort>))

endlet))

endlet)

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

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

##### F2.2.5.7.12 Gate context parameter

Concrete syntax

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

Conditions on concrete syntax

∀fcp∈<gate context parameter>:∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ⇒

(let gd=getEntityDefinition0(acp, gate) in

(gd.**s**-<gate> = fcp.**s**-<gate>) ∧

(∀gc∈<gate constraint>:∀gc1∈<gate constraint>:

(gc.parentAS0 = gd) ∧ (gc1.parentAS0= fcp) ∧

((gc.direction0=out) ∧ (gc1. direction0=out) ⇒

gc1.**s**-<sort>-seq .signalSet0) ⊆gc.**s**-<sort>-seq .signalSet0) ∧

((gc. direction0=in) ∧ (gc1. direction0=in) ⇒

gc.**s**-<sort>-seq .signalSet0⊆gc1.**s**-<sort>-seq .signalSet0)

endlet)

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

##### F2.2.5.7.13 Interface context parameter

Concrete syntax

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

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

<interface constraint> :: <interface<identifier>

Conditions on concrete syntax

∀fcp∈<interface context parameter name>: ∀acp∈<actual context parameter>:

isContextParameterCorresponded0(fcp, acp) ∧ (fcp.**s-**<interface constraint>≠ undefined) ⇒

(∃td∈<interface definition>:

(td =getEntityDefinition0(acp.**s**-<identifier>, interface)) ∧

isSubtype0(td, fcp.contextParameterAtleastDefinition0))

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

#### F2.2.5.8 Specialization

##### F2.2.5.8.1 Adding properties

Concrete syntax

<specialization> :: <type expression>

Mapping to abstract syntax

| <specialization>(x) then Mapping(x)

Auxiliary functions

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

specialization0(def: EntityDefinition0):

<specialization>∪<data type specialization>∪<interface specialization> =def

take({ s∈<specialization>∪<data type specialization>∪<interface specialization>:

s.surroundingScopeUnit0 = def})

##### F2.2.5.8.2 Virtual type

Concrete syntax

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

<virtuality constraint> :: <identifier>

Conditions on concrete syntax

∀td∈<block type definition>∪<process type definition>∪<procedure definition>∪

<composite state type definition>:

isVirtualType0(td) ⇒ td.virtualTypeAtleastDefinition0.entityKind0 = td.entityKind0

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

∀td∈<block type definition>∪<process type definition>∪<procedure definition>∪

<composite state type definition>:

td.isVirtualType0 ⇒

¬( td.isParameterizedType0) ∧ ¬(isParameterizedType0(td.virtualTypeAtleastDefinition0))

A virtual type and its constraints cannot have context parameters.

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

Only virtual types may have <virtuality constraint> specified.

∀r∈<block type reference> ∪<process type reference> ∪<composite state type reference> ∪

<procedure reference>:

∀d∈<block type definition>∪<process type definition>∪<composite state type definition>∪

<procedure definition>:

(r.referencedDefinition0 = d)∧(r.virtuality0 ≠ undefined)∧(d.virtuality0 ≠ undefined) ⇒

(r.virtuality0 = d.virtuality0)

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

∀td∈<block type definition>∪<process type definition>: isVirtualType0(td) ⇒

(**let** td1= td.virtualTypeAtleastDefinition0 **in**

(**let** fpl =td.agentFormalParameterList0in

let fpl1 = td1.agentFormalParameterList0 **in**

isSameAgentFormalParameterList0(fpl, fpl1) endlet) ∧

(∀gd1∈<gate in definition>: gd1∈ td1.gateDefinitionSet0 ⇒

∃gd∈<gate in definition>: (gd∈ td.gateDefinitionSet0) ∧ isSameGate0(gd, gd1)) ∧

(∀fd1∈<interface definition>: fd1∈ td1.interfaceDefinitionSet0 ⇒

∃fd∈<interface definition>: (fd∈ td.interfaceDefinitionSet0) ∧ isSameInterface0(fd, fd1)) ∧

(∀ed∈EntityDefinition0: ed.surroundingScopeUnit0 = td⇒

∃ed1∈EntityDefinition0: (ed1.surroundingScopeUnit0 = td1) ∧ (ed= ed1)) endlet)

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

∀td∈<composite state type definition>: isVirtualType0(td) ⇒

(**let** td1 = td.virtualTypeAtleastDefinition0 **in**

∀scp1∈<state connection points>: scp1∈td1.stateConnectionPointSet0 ⇒

∃scp∈<state connection points>:

(scp∈td.stateConnectionPointSet0) ∧ isSameStateConnectionPoint0(scp, scp1) endlet)

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

∀td∈<procedure definition>: isVirtualType0(td) ⇒

(**let** fpl = td.procedureFormalParameterList0 in

let fpl1 = td.virtualTypeAtleastDefinition0.procedureFormalParameterList0 **in**

isSameProcedureFormalParameterList0(fpl, fpl1) endlet)

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

∀td∈<block type definition>∪<process type definition>∪<procedure definition>∪

<composite state type definition>:

isVirtualType0(td) ⇒

(∀sp∈<specialization>:∀vc∈<virtuality constraint>:

(sp.surroundingScopeUnit0 = td) ∧ (vc.surroundingScopeUnit0 = td) ⇒

((td.virtualTypeInheritsDefinition0= td.virtualTypeAtleastDefinition0) ∨

isSubtype0(td.virtualTypeInheritsDefinition0, td.virtualTypeAtleastDefinition0)))

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

∀td, td1∈<block type definition>∪<process type definition>∪<procedure definition>∪

<composite state type definition>:

isRedefinedType0(td) ∧ (td1=td.superCounterpart0) ⇒

(let sp=td.specialization0in

let vc = take({vc∈<virtuality constraint>: vc.surroundingScopeUnit0 = td1}) in

(sp≠undefined) ∧ (vc=undefined) ⇒

isSubtype0(td.virtualTypeInheritsDefinition0, td1.virtualTypeAtleastDefinition0) endlet)

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

Mapping to abstract syntax

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

Auxiliary functions

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

isVirtualType0(td: <block type definition>∪<process type definition>∪<internal procedure definition>∪

<composite state type definition>): Boolean=def

td.virtuality0∈{virtual, redefined}

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

isRedefinedType0(td: <block type definition>∪<process type definition>∪<internal procedure definition>

∪<composite state type definition>): Boolean=def

td.virtuality0∈{redefined, finalized}

Get the virtuality for a definition.

virtuality0(td: DefinitionAS0): {virtual,redefined, finalized} =def

case d of

| <block type definition> then

d.s-<block type heading>.s-<type preamble>.**s**-<virtuality>

| <composite state type definition> then

d.s-<composite state type heading>.**s**-<virtuality>

| <interface definition> then d.**s**-<virtuality>

| <internal procedure definition> then

d.s-<procedure heading>.s-<procedure preamble>.s-<type preamble>.**s**-<virtuality>

| <operation definition> then

d.s-<operation heading>.s-<operation preamble>.**s**-<virtuality>

| <operation signature> then d.s-<operation preamble>.**s**-<virtuality>

| <process type definition> then

d.s-<process type heading>.s-<type preamble>.**s**-<virtuality>

| <statements> then d.parentAS0.**s**-<virtuality>

| <block type reference> ∪<process type reference> ∪<composite state type reference> ∪

<procedure reference> then d.s-<type preamble>.**s**-<virtuality>

| <start>∪<input part>∪<priority input>∪<save part>∪<spontaneous transition>∪

<continuous signal>∪<connect part>∪<default initialization> then d.**s**-<virtuality>

otherwise undefined

endcase

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

virtualTypeAtleastDefinition0(td: <block type definition>∪<process type definition>∪

<procedure definition>∪<composite state type definition>): <block type definition>∪

<process type definition>∪<procedure definition>∪<composite state type definition> =def

let vc=take({vc∈<virtuality constraint>: vc.surroundingScopeUnit0 = td}) in

if vc ≠ undefined then getEntityDefinition0(vc, td.entityKind0) else td endif

endlet

Determine if two agent formal parameter lists are the same.

isSameAgentFormalParameterList0(fpl: <name>\*, fpl1:<name>\*): Boolean=def

(fpl.length = fpl1.length) ∧

(∀i∈1..fpl.length:(fpl[i]=fpl1[i])∧isSameSort0(fpl[i].parentAS0.s-<sort>,fpl1[i].parentAS0.s-<sort>))

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

isSameGate0(gd: <gate in definition>, gd1: <gate in definition> ): Boolean=def

if (gd∈<textual gate definition>)∧(gd1∈<textual gate definition>) then

(gd.s-<gate>=gd1.s-<gate>)∧

(∀gc∈ gd.s-<gate constraint>:∃gc1∈ gd1.s-<gate constraint>:

(gc.direction0 = gc1.direction0)∧

(gc.s-<textual endpoint constraint> =gc1.s-<textual endpoint constraint>)∧

isSameSortList0(gc.s-<sort>-seq, gc1.s-<sort>-seq ))∧

(∀gc1∈ gd1.s-<gate constraint>:∃gc∈ gd.s-<gate constraint>:

(gc.direction0 = gc1.direction0)∧

(gc.s-<textual endpoint constraint> =gc1.s-<textual endpoint constraint> )∧

isSameSortList0(gc.s-<sort>-seq, gc1.s-<sort>-seq ))

else if (gd∈<textual interface gate definition>)∧(gd1∈<textual interface gate definition>) then

gd.s-<identifier>=gd1.s-<identifier>

else false

endif

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

isSameInterface0(id:<interface definition>, id1:<interface definition>): Boolean=def

(id.virtuality0 = id1.virtuality0)∧

(id.entityName0 = id1.entityName0)∧

(id.entityDefinitionSet0= id1.entityDefinitionSet0)

Get all the entity definitions defined in a scope unit.

entityDefinitionSet0(su: ScopeUnit0): EntityDefinition0 =def

{ed∈EntityDefinition0: isDefinedIn0(ed, su)}

Get all the interface definitions defined in a scope unit.

interfaceDefinitionSet0(d: ScopeUnit0):<interface definition>-set =def

{fd∈<interface definition>: isDefinedIn0(fd, d)}

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

stateConnectionPointSet0(td:<composite state type definition>∪<composite state definition> ):

<state connection points>-set =def

{scp∈<state connection points>:

(scp.parentAS0=td)∧

(∃td1∈<composite state type definition>: isSubtype0(td, td1)∧ (scp.parentAS0=td1))}

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

isSameStateConnectionPoint0(scp: <state connection points>, scp1: <state connection points>):

Boolean=def

{n∈<name>: isAncestorAS0(scp,n)} = {n1∈<name>: isAncestorAS0(scp1,n1)}

Determine if two procedure formal parameter lists are the same.

isSameProcedureFormalParameterList0(fpl: <name>\*, fpl1: <name>\*): Boolean=def

(fpl.length = fpl1.length) ∧

(∀i∈1..fpl.length:

(fpl[i].parentAS0.parentAS0.**s**-<parameter kind> =

fpl1[i].parentAS0.parentAS0.**s**-<parameter kind>) ∧

(fpl[i] = fpl1[i]) ∧ isSameSort0(fpl[i].parentAS0.**s**-<sort>, fpl1[i].parentAS0.**s**-<sort>))

Get the entity definition specialized by a virtual type.

virtualTypeInheritsDefinition0(td: <block type definition>∪<process type definition>∪

<procedure definition>∪<composite state type definition>): <block type definition>∪

<process type definition>∪<procedure definition>∪<composite state type definition>=def

let sp=td.specialization0 in

if (sp ≠ undefined) then sp.s-<type expression>.baseType0

else

let vc=take({vc∈<virtuality constraint>: vc.surroundingScopeUnit0 = td}) in

case td.virtuality0 of

| virtual then

if (vc = undefined) then undefined else td.virtualTypeAtleastDefinition0 endif

| redefined then

if (vc = undefined) then td.superCounterpart0.virtualTypeAtleastDefinition0

else td.virtualTypeAtleastDefinition0

endif

| finalized then td.superCounterpart0.virtualTypeAtleastDefinition0

endcase

endlet

endif

endlet

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

superCounterpart0(td: <block type definition>∪<process type definition>∪<procedure definition>∪

<composite state type definition>∪<operation definition>∪<operation signature> ):

(<block type definition>∪<process type definition>∪<procedure definition>∪

<composite state type definition>∪<operation definition>)-set =def

{td1∈<block type definition>∪<process type definition>∪<procedure definition>∪

<composite state type definition>∪<operation definition>∪<operation signature>:

isSuperCounterpart0(td1, td)}

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

isSuperCounterpart0(td: DefinitionAS0, td1: <block type definition>∪<process type definition>∪

<procedure definition>∪<composite state type definition>∪

<operation definition>): Boolean =def

(td.name0= td1.name0) ∧ (td.kind0= td1.kind0) ∧

(td.virtuality0∈{virtual, redefined})∧( td1.virtuality0∈{redefined, finalized}) ∧

isDirectSubType0(td1.surroundingScopeUnit0, td.surroundingScopeUnit0)

##### F2.2.5.8.3 Virtual transitions/save

Conditions on concrete syntax

∀ad∈<agent definition> ∪<textual typebased agent definition> ∪<composite state definition> ∪

<typebased composite state>:

(∀s∈<start>: s∈ad.startSet0⇒s.virtuality0 = undefined) ∧

(∀s∈<state>: (s∈ad.stateSet0)⇒

(s.s-<input part>.virtuality0 = undefined) ∧

(s.s-<priority input>.virtuality0 = undefined) ∧

(s.s-<save part>.virtuality0 = undefined) ∧

(s.s-<spontaneous transition>.virtuality0 = undefined) ∧

(s.s-<continuous signal>.virtuality0 = undefined))

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

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

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

(∀ip∈<input part>:

(ip.virtuality0 ≠ undefined) ⇒ ip.s-<input list>.s-<asterisk input list> = undefined) ∧

(∀sp∈<save part>:

(sp.virtuality0 ≠ undefined) ⇒ sp.s-<save list>.s-<asterisk save list> = undefined)

An input or save with <virtuality> shall not contain <asterisk>. See item(b) clause 8.4 *Concrete grammar* [ITU‑T Z.103].

Auxiliary functions

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

startSet0(td: <agent definition> ∪<textual typebased agent definition> ∪<agent type definition> ∪

<composite state definition> ∪ <typebased composite state>∪<composite state type definition>∪

<textual typebased state partition def> ∪<state partition>∪

<internal procedure definition>):<start>-set =def

case td of

| <agent definition>∪

<agent type definition>∪

<composite state type definition>∪

<internal procedure definition> then

td.localStartSet0∪td.inheritedStartSet0

| <textual typebased agent definition> ∪ <textual typebased state partition def> then

let te=take({te∈<type expression>: te.parentAS0.parentAS0 = td}) in

te.baseType0.startSet0

endlet

| <composite state definition> then

if td.**s**-<composite state structure>.s-implicit ∈<composite state body> then

{s∈<start>: s.parentAS0=td.**s**-<composite state structure>.s-implicit}

else {s∈sp.startSet0:

sp∈<state partition>∧sp.parentAS0= td.**s**-<composite state structure>.s-implicit}

endif

| <typebased composite state> then td.s-<type expression>.baseType0.startSet0

| <composite state reference> then td.referencedDefinition0.startSet0

otherwise ∅

endcase

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

localStartSet0(td: <agent definition> ∪<agent type definition> ∪<composite state type definition>∪

<internal procedure definition>):<start>-set =def

case td of

| <agent definition> ∪ <agent type definition> then

if td.s-<agent structure>.**s**-implicit ∈ <agent body> then

{td.**s**-<agent structure>.**s**-implicit.s-<start>}

else td.s-<agent structure>.**s**-implicit.**s**-<state partition>.startSet0

endif

| <composite state type definition> then

if td.**s**-<composite state structure>.**s**-implicit∈<composite state body> then

{s∈<start>: s.parentAS0=td.**s**-<composite state structure>.**s**-implicit }

else {s∈sp.startSet0:

sp∈<state partition>∧sp.parentAS0=td.**s**-<composite state structure>.**s**-implicit}

endif

| <internal procedure definition> then

if td.s-implicit∈<procedure body> then { td.s-implicit.s-<start>}

else ∅

endif

otherwise ∅

endcase

Get the set of <start> items inherited by a given definition.

inheritedStartSet0(td: <agent definition> ∪<agent type definition> ∪

<composite state type definition> ∪<internal procedure definition>): <start>-set =def

let sp=td.specialization0 in

if sp=undefined then ∅

else sp.**s**-<type expression>.baseType0.startSet0

endif

endlet

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

stateSet0(td: <agent definition> ∪<textual typebased agent definition> ∪<agent type definition> ∪

<composite state definition> ∪<typebased composite state>∪<composite state type definition>∪

<textual typebased state partition def> ∪

<internal procedure definition>):<state>-set =def

case td of

| <agent definition> ∪

<agent type definition> ∪

<composite state type definition>∪

<internal procedure definition> then

td.localStateSet0∪td.inheritedStateSet0

| <textual typebased agent definition> ∪<textual typebased state partition def> then

let te = take({te ∈ <type expression>: te.parentAS0.parentAS0 = td}) in

te.baseType0.stateSet0

endlet

| <composite state definition> then

if td.**s**-<composite state structure>.**s**-implicit ∈ <composite state body> then

{s∈<state>: s.parentAS0=td.**s**-<composite state structure>.**s**-implicit}

else {s∈sp.stateSet0:

sp∈<state partition>∧sp.parentAS0=td.**s**-<composite state structure>.**s**-implicit}

endif

| <typebased composite state> then td.s-<type expression>.baseType0.stateSet0

| <composite state reference> then td.referencedDefinition0.stateSet0

otherwise ∅

endcase

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

localStateSet0(td: <agent definition> ∪<agent type definition> ∪<composite state type definition>∪

<internal procedure definition>): <state>-set =def

case td of

| <agent definition> ∪ <agent type definition> then

if td.s-<agent structure>.**s**-implicit ∈ <agent body> then

{s∈<state>: s.parentAS0 = td.**s**-<agent structure>.**s**-implicit}

else td.s-<agent structure>.s-<state machine>.**s**-implicit.stateSet0

endif

| <composite state type definition> then

if td.**s**-<composite state structure>.**s**-implicit∈<composite state body> then

{s∈<state>: s.parentAS0= td.**s**-<composite state structure>.**s**-implicit }

else {s∈sp. stateSet0:

sp∈<state partition>∧sp.parentAS0=td.**s**-<composite state structure>.**s**-implicit}

endif

| <internal procedure definition> then

if td.s-implicit∈<procedure body> then { s∈<state>: s.parentAS0= td.s-implicit}

else ∅

endif

otherwise ∅

endcase

Get the set of <state> items inherited by a given definition.

inheritedStateSet0(td: <agent definition> ∪<agent type definition> ∪

<composite state type definition>∪<internal procedure definition>):<state>-set =def

let sp=td.specialization0 in

if sp=undefined then ∅

else sp.**s**-<type expression>.baseType0.stateSet0

endif

endlet

##### F2.2.5.8.4 Virtual method

Conditions on concrete syntax

∀os∈<operation signature>:

(os.entityKind0=method) ∧ (os.virtuality0∈{redefined,finalized}) ⇒

∃os1∈<operation signature>: (os1= os.superCounterpart0) ∧

isOperationSignatureCompatible0(os, os1)∧

(**let** fpl = os.operationSignatureParameterList0 in

(let fpl1 = os1.operationSignatureParameterList0 **in**

∀i∈1..fpl.length:

(fpl[i].s-<formal parameter>.**s**-<parameter kind>=

fpl1[i].s-<formal parameter>.**s**-<parameter kind>)

endlet)

endlet)

∀od∈<operation definition>:

(od.entityKind0=method) ∧ (od.virtuality0∈{redefined,finalized}) ⇒

∃od1∈<operation definition>:

( (od1= od.superCounterpart0) ∧

( od1.s-<operation heading>.s-<operation result>.s-<result aggregation> =

od.s-<operation heading>.s-<operation result>.s-<result aggregation>))

When a method is redefined in a specialization, its signature shall 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 shall only denote a sort B such that B is sort compatible to A. A redefinition of a virtual method shall not change the <parameter kind> in any <argument> of the inherited <operation signature>. A redefinition of a virtual method shall not change the <aggregation kind> of the <operation result> of the inherited <operation definition>. See clause 12.1.3 *Concrete grammar* of [ITU‑T Z.107].

Auxiliary functions

Determine if two <operation signature>s are compatible.

isOperationSignatureCompatible0(os: <operation signature>, os1: <operation signature>):

Boolean=def

isSortCompatible0(os.s-<result>.**s**-<sort>, os1.s-<result>.**s**-<sort>) ∧

(**let** fpl = os.operationSignatureParameterList0 in

(let fpl1 = os1.operationSignatureParameterList0 **in**

(fpl.length = fpl1.length) ∧

(∀i∈1..fpl.length:

(isSortCompatible0(fpl[i].s-<sort>,fpl1[i].s-<sort>)) ∧ (isSameSort0(fpl[i].s-<sort>, fpl1[i].s-<sort>)))

endlet)

endlet)

### F2.2.6 Agents

Abstract syntax

Agent-definition :: Agent-name

Number-of-instances

Agent-type-identifier

Number-of-instances :: Initial-number

[ Maximum-number ]

Lower-bound

Initial-number = Nat

Maximum-number = Nat

Lower-bound = Nat

Agent-formal-parameter = Parameter

Parameter :: Variable-name

Sort-reference-identifier

Parameter-aggregation

Parameter-aggregation :: Aggregation-kind

State-machine :: State-name

Nextstate-parameters

Composite-state-type-identifier

State-transition-graph ::

[ State-start-node ]

Named-start-node-set

State-node-set

Free-action-set

Conditions on abstract syntax

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

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

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

d.s-Number-of-instances.s-Initial-number =1 ∧ d.s-Number-of-instances.s-Maximum-number = 1

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

∀ni ∈ Number-of-instances:

( ni.s-Maximum-number ≠ undefined ⇒ ni.s-Initial-number ≤ ni.s-Maximum-number )

∧ ( ni.s-Lower-bound ≤ ni.s-Initial-number )

If there is a Maximum-number, the Initial-numberof instances shall be less than or equal to Maximum-numberand Maximum-numbershall be greater than zero. The Lower-boundshall be less than or equal to the Initial-number. See clause 9 of [ITU‑T Z.101].

Concrete syntax

<agent definition> =

<system definition>

| <block definition>

| <process definition>

<agent structure> ::

<entity in agent>\*

{ <interaction> | <agent body> }

<interaction> ::

{ <channel to channel connection>

| <channel definition>

| <agent definition>

| <block reference>

| <process reference>

| <textual typebased agent definition>}\*

[<state machine>]

<state machine> =

<typebased composite state>

| <composite state list item>

<agent instantiation> ::

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

<agent additional heading> ::

[<specialization>] [<agent formal parameters>]

<entity in agent> =

<block type definition>

| <block type reference>

| <composite state definition>

| <composite state type definition>

| <composite state type reference>

| <data definition>

| <gate in definition>

| <procedure definition>

| <procedure reference>

| <process type definition>

| <process type reference>

| <remote procedure definition>

| <remote variable definition>

| <select definition>

| <signal definition list>

| <signal list definition>

| <timer definition>

| <valid input signal set>

| <variable definition>

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

<number of instances> ::

[<initial number>] [<maximum number>] [ <lower bound> ]

<initial number> = <Natural<simple expression>

<maximum number> = <Natural<simple expression>

<lower bound> = <Natural<simple expression>

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

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

<agent body> ::

[<start>] {<state> | <free action>}\*

NOTE – In [ITU‑T Z.106] <entity in agent> includes <imported procedure specification>/<imported variable specification>, but these have no SDL‑2010 meaning (see clause 9 of [ITU‑T Z.103]), and <macro definition> is removed by step 1 transformations, therefore none of these three items appears in AS0 <entity in agent>.

Conditions on concrete syntax

∀ainst∈<agent instantiation>:

ainst.s-<agent additional heading>.s-<agent formal parameters> ≠ undefined ⇒

ainst.s-<number of instances> ≠ undefined

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

∀afps∈<agent formal parameters>:

afps.parentAS0 ∈ (<composite state type heading> ∪ <state aggregation type heading> ∪

<agent additional heading> ∪ <composite state heading>) ⇒

afps.length > 0

When used in the concrete syntax (occurs in places in the condition above), the <agent formal parameters> list shall have at least one item. However, it is defined with "\*" rather than "+" so that it can used as an empty list within conditions, transformations, mappings and functions.

Transformations

< <system definition>(uses, <system heading>(n, addHead), body) >

=12=>

let nn=newName in

< <textual typebased system definition>(

<typebased system heading>(n, <type expression>(<identifier>(undefined, nn), empty))),

<system type definition>(uses,

<system type heading>(empty, nn,

<agent type additional heading>(empty, undefined, addHead)),

body) >

endlet // nn

< <block definition>(uses, <block heading>(\*, n, <agent instantiation>(inst, addHead)), body) >

=12=> // \* is optional qualifier which is ignored

let nn=newName in

< <textual typebased block definition>(

<typebased block heading>(n, inst, <type expression>(<identifier>(undefined, nn), empty))),

<block type definition>(uses,

<block type heading>(empty, undefined, nn,

<agent type additional heading>(empty, undefined, addHead)),

body) >

endlet // nn

< <process definition>(uses, <process heading>(\*, n, <agent instantiation>(inst, addHead)), body) >

=12=> // \* is optional qualifier which is ignored

let nn=newName in

< <textual typebased process definition>(

<typebased process heading>(n, inst, <type expression>(<identifier>(undefined, nn), empty))),

<process type definition>(uses,

<process type heading>(empty, undefined, nn,

<agent type additional heading>(empty, undefined, addHead)),

body) >

endlet // nn

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

NOTE – In F2.2.5.1.1 an <agent structure> with an <agent body> transformed into an <agent structure> with an <interaction>.

<agent formal parameters>(params)

provided ∃ i ∈ 1..params.length: params[i].s-<parameters of sort>.s-<name>-seq.length ≠1

=5=>

mk-<agent formal parameters>(

< if params[i].s-<parameters of sort>.s-<name>-seq.length ≠1

then // replacement list elements – two (aggregation kind, parameters of sort) pairs

( params[i].s-<aggregation kind>, //

mk-< parameters of sort > (

< params[i].s-<parameters of sort>.s-<name>-seq.head >,

params[i].s-<parameters of sort>.s-<sort>)

), // first (aggregation kind, parameters of sort) pair

( params[i].s-<aggregation kind>,

mk-< parameters of sort > (

params[i].s-<parameters of sort>.s-<name>-seq.tail, // transform again if >1 name in tail

params[i].s-<parameters of sort>.s-<sort>)

) // second (aggregation kind, parameters of sort) pair

else params[i] // only one name, no change needed

endif

: i ∈ 1..params.length

> // list of (aggregation kind, parameters of sort) pairs

)

An <agent formal parameters> list item with a <parameters of sort> that defines multiple parameter names is replaced by a sequence of <agent formal parameters> list items with the same <aggregation kind> each <parameters of sort> defining one name. See clause 8.1.1.1 *Model* of [ITU‑T Z.101].

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

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

*The following transformation is covered by the dynamic semantics.*

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

a) the agent instance (self);

b) the creating agent instance (parent);

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

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

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

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

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

Mapping to abstract syntax

| <number of instances>(init, max, lower) then

mk-Number-of-instances(

if init ≠ undefined then Mapping(init) else 1 endif,

if max ≠ undefined then Mapping(max) else undefined endif,

if lower ≠ undefined then Mapping(lower) else 0 endif)

If <initial number> is omitted, then Initial-numberis 1. If <maximum number> is omitted, then Maximum-numberis unbounded (it is omitted in Number-of-instances). If the <lower bound> is omitted, the Lower-boundis zero.

| <agent formal parameters>(param) then < Mapping(param[i]) : i ∈ 1..param.length>

| param=<parameters of sort>(< name >, s) then

mk-Parameter(Mapping(name), // Variable-name

Mapping(s), // Sort-reference-identifier

case parent=param.parentAS0 of // Parameter aggregation

| <agent formal parameters> then Mapping(parent.s-<aggregation kind>)

| <formal variable parameters> then Mapping(parent.s-<parameter aggregation>)

endcase // param

)

#### F2.2.6.1 System

Concrete syntax

<system definition> ::

<package use clause>\* <system heading> <agent structure>

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

Conditions on concrete syntax

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

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

Transformations

let nn= newName in

< c=<channel definition>(n, d,

<channel path>(ep1=<channel endpoint>(env, undefined), ep2, list1),

undefined) >

provided c.parentAS0.parentAS0.parentAS0 ∈ <system definition> =8=>

< <textual gate definition>(nn),

<channel definition>(n, d,

<channel path>(<channel endpoint>(env, nn), ep2, list1), undefined) >

endlet // nn

let nn= newName in

< c=<channel definition>(n, d,

<channel path>(ep1=<channel endpoint>(env, undefined), ep2, list1),

<channel path>(ep2, ep1, list2)) >

provided c.parentAS0.parentAS0.parentAS0 ∈ <system definition> =8=>

< <textual gate definition>(nn),

<channel definition>(n, d,

<channel path>(<channel endpoint>(env, nn), ep2, list1),

<channel path>(ep2, <channel endpoint>(env, nn), list2)) >

endlet // nn

let nn= newName in

< c=<channel definition>(n, d,

<channel path>(ep1, ep2=<channel endpoint>(env, undefined), list1),

undefined) >

provided c.parentAS0.parentAS0.parentAS0 ∈ <system definition> =8=>

< <textual gate definition>(nn),

<channel definition>(n, d,

<channel path>(ep1, <channel endpoint>(env, nn), list1), undefined) >

endlet // nn

let nn= newName in

< c=<channel definition>(n, d,

<channel path>(ep1, ep2=<channel endpoint>(env, undefined), list1),

<channel path>(ep2, ep1, list2)) >

provided c.parentAS0.parentAS0.parentAS0 ∈ <system definition> =8=>

< <textual gate definition>(nn),

<channel definition>(n, d,

<channel path>(ep1, <channel endpoint>(env, nn), list1),

<channel path>(<channel endpoint>(env, nn), ep1, list2)) >

endlet // nn

For each <channel definition> in a system mentioning **env**, a gate with an anonymous name is added to the Agent-definition. The channel definition is changed to mention this gate in the <channel path> directed to the system environment.

#### F2.2.6.2 Block

Concrete syntax

<block definition> ::

<package use clause>\* <block heading> <agent structure>

<block heading> ::

<qualifier> <block<name> <agent instantiation>

#### F2.2.6.3 Process

Concrete syntax

<process definition> ::

<package use clause>\* <process heading> <agent structure>

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

#### F2.2.6.4 Procedure

Abstract syntax

Procedure-definition :: Procedure-name

Procedure-formal-parameter\*

[ Result ]

[ Procedure-identifier ]

Data-type-definition-set

Syntype-definition-set

Variable-definition-set

Composite-state-type-definition-set

Procedure-definition-set

Procedure-graph

[ Abstract ]

Procedure-formal-parameter = In-parameter

| Inout-parameter

| Out-parameter

In-parameter :: Parameter

Inout-parameter :: Parameter

Out-parameter :: Parameter

Procedure-graph ::

[ Procedure-start-node ]

State-node-set

Free-action-set

Result :: Sort-reference-identifier

*Result-aggregation*

*Result-aggregation* :: *Aggregation-kind*

Concrete syntax

<procedure definition> =

<external procedure definition>

| <internal procedure definition>

<internal procedure definition> ::

<package use clause>\* <procedure heading> <entity in procedure>\*

{ <procedure body> | [ <virtuality> ] <statements> }

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

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

<procedure heading> ::

<procedure preamble> <qualifier> <procedure<name>

[ <formal context parameters> ]

[<virtuality constraint>]

[<specialization>]

<procedure formal parameters>

[<procedure result>]

<procedure formal parameters> =

<formal variable parameters>\*

<entity in procedure> =

<variable definition>

| <data definition>

| <procedure reference>

| <procedure definition>

| <composite state definition>

| <composite state type definition>

| <composite state type reference>

| <select definition>

<procedure body> ::

[<start>] {<state> | <free action>}\*

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

<formal variable parameters> :: <parameter kind> <parameter aggregation> <parameters of sort>

<parameter aggregation> :: <aggregation kind>

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

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

<result aggregation> :: <aggregation kind>

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

Conditions on concrete syntax

∀pd∈<procedure definition>:pd.isExported0 ⇒

pd.formalContextParameterList0 = empty∧

pd.surroundingScopeUnit0 ∈

<agent type definition> ∪ <composite state type definition> ∪

<agent definition> ∪ <composite state definition>

An exported procedure shall not have formal context parameters and its enclosing scope shall be an agent type or a composite state type. See clause 9.4 *Concrete grammar* of [ITU‑T Z.102].

NOTE – Although the condition requires the enclosing scope to be an agent type or composite state type, the alternatives for agent and composite state are included, so that these are valid before they are transformed to an agent type or composite state type that surrounds the exported procedure.

∀vd∈<variable definition>:

vd.surroundingScopeUnit0∈<procedure definition>⇒ ¬vd.isExported0

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

∀pd1,pd2∈<internal procedure definition>:

(parentAS0ofKind(pd1, <agent type definition> ) =

parentAS0ofKind(pd2, <agent type definition> ) ∧

pd1.isExported0∧ pd2.isExported0∧ pd1 ≠ pd2)⇒

(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 type or any enclosed composite state type of the agent type shall not mention the same <remote procedure identifier>. See clause 9.4 *Concrete grammar* of [ITU‑T Z.102].

NOTE – The parentAS0ofKind search does not need to include enclosed composite state types, because the agent type is a parent of these also. An agent definition that contains an exported procedure definition is transformed to a typebased agent that instantiates an agent type definition, so the check on the agent type definition is sufficient.

∀te∈<type expression>: te.baseType0∉ <external procedure definition> ∧

∀procCons∈<procedure constraint>: procCons.s-<identifier> ≠ undefined⇒

getEntityDefinition0(procCons.s-<identifier>, procedure) ∉

<external procedure definition>

An external procedure cannot be mentioned in a <type expression> or in a <procedure constraint>.

Transformations

<formal variable parameters>(undefined,aggr,params)

=5=>

<formal variable parameters>(in,aggr,params)

A formal parameter with no explicit <parameter kind> has the implicit <parameter kind> in. See clause 9.5 *Model* of [ITU‑T Z.103].

fvp=<formal variable parameters>(kind, ak, params)

provided

params.s-<name>-seq.length ≠1

=5=>

mk-<formal variable parameters>( kind, ak, < params.s-<name>-seq.head >)

⁀

mk-<formal variable parameters>( kind, ak, params.s-<name>-seq.tail ) // transform again if >1 name in tail

A <formal variable parameters> with a <parameters of sort> that defines multiple parameter names is replaced by a sequence of <formal variable parameters> with the same <parameter kind> and <aggregation kind>, and each <parameters of sort> defining one name. See clause 9.4 *Model* of [ITU‑T Z.101].

ph = <procedure heading>(pre,q,n,fcp,vc,sp,fpars,res)

=5=>

provided expandFormalVarParams0(fpars).length > fpars.length

<procedure heading>(pre,q,n,fcp,vc,sp,expandFormalVarParams0(fpars),res)

If any <parameters of sort> in a formal parameter list defines more than one name, the formal parameter list is expanded so that each <parameters of sort> defines only one name. Derived from clause 8.1.1.1 *Concrete grammar* of [ITU‑T Z.101].

<internal procedure definition>(uses,

h=<procedure heading>(\*, \*, \*, \*, \*, \*, \*,

<procedure result>(resName, resSort), \*), entities, body)

provided resName ≠ undefined ∧

replaceInSyntaxTree0(<return body>(undefined),

<return body>(<expression>(<operand5>(<variable access>(<identifier>( undefined, resName))))),

body) ≠ body

=8=>

<internal procedure definition>(uses, h, entities,

replaceInSyntaxTree0(<return body>(undefined),

<return body>(<expression>(<operand5>(<variable access>(<identifier>( undefined, resName))))),

body))

When a <variable<name> is present in <procedure result>, each <return body> within the procedure graph without an <expression> is replaced by a <return body> containing the <variable<name> as the <expression>.

p=<internal procedure definition>(uses,

h=<procedure heading>(\*, \*, \*, \*, \*, \*, \*, <procedure result>(resName, resSort), \*),

entities, body)

provided resName ≠ undefined ∧

resName ∉ {n in v.s-<name>-seq | v ∈ <variables of sort>: v.parentAS0.parentAS0.parentAS0 = p }

=8=>

<internal procedure definition>

( uses, h,

entities ⁀ <variable definition>

( undefined,

< <variables of sort>

(< resName >, resSort, undefined)

>

),

body)

A <procedure result> with <variable name> is derived syntax for a <variable definition> with <variable name> and <sort> in <variables of sort>. If there is a <variable definition> involving <variable name> no further <variable definition> is added.

*The following statement is covered by the dynamic semantics.*

A <procedure start area> which contains <virtuality> of virtual or redefined, a <start> of a <procedure body> which contains <virtuality> of virtual or redefined, or a <statements> in an <internal procedure definition> following <virtuality> of **virtual** or **redefined** is called a virtual procedure start. Virtual procedure start is further described in clause 8.4.3 of [ITU-T Z.102].

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

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

Mapping to abstract syntax

| <internal procedure definition>(\*,<procedure heading>(\*,\*,n,\*,\*,parent,pars,result,\*),entities,body) then

let entitiesSet = { Mapping(entities[i]:i ∈ 1..entities.length} in

mk-Procedure-definition(Mapping(n),

< Mapping(pars[j]): j ∈ 1..pars.length} >,

Mapping(result),

Mapping(parent),

{ e ∈ entitiesSet: e ∈ Data-type-definition },

{ e ∈ entitiesSet: e ∈ Syntype-definition },

{ e ∈ entitiesSet: e ∈ Variable-definition },

{ e ∈ entitiesSet: e ∈ Composite-state-type-definition },

{ e ∈ entitiesSet: e ∈ Procedure-definition },

Mapping(body) )

endlet // entitiesSet

| <formal variable parameters>(in, param) then mk-In-parameter(Mapping(param))

| <formal variable parameters>(out, param) then mk-Out-parameter(Mapping(param))

| <formal variable parameters>(inout, param) then mk-Inout-parameter(Mapping(param))

| <procedure body>(start, nodes) then

let nodeSet = { Mapping(nodes[i]:i ∈ 1..nodes.length} in

mk-Procedure-graph(

Mapping(start),

{ sn ∈ nodeSet: sn ∈ State-node } // State-node-set

{ fa ∈ nodeSet: fa ∈ Free-action } // Free-action-set

)

endlet // nodeSet

Auxiliary functions

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

isExported0(vp: <variable definition>∪<internal procedure definition>): Boolean=def

case vp of

| <variable definition>(vlist) then

if vlist.head ∈ <exported variables of sort> then true else false endif

| <internal procedure definition> then

if vp.s-<procedure heading>.s-<procedure preamble>.s-<exported> = undefined

then false

else true

endif

otherwise false

endcase

Get the formal parameter list for an <internal procedure definition>.

procedureFormalParameterList0(pd: <internal procedure definition>): <name>\*=def

pd.localProcedureFormalParameterList0⁀ pd.inheritedProcedureFormalParameterList0

localProcedureFormalParameterList0(pd: <internal procedure definition>): <name>\* =def

**let** fpl=pd.**s**-<procedure heading>.**s**-<formal variable parameters>-seq **in**

if fpl = empty then empty

else

< f.s-<parameters of sort>.s-<variable<name>-seq | f in fpl >

endif

endlet

inheritedProcedureFormalParameterList0(pd: <internal procedure definition>): <name>\* =def

let sp = pd.specialization0 in

if sp = undefined then empty

else sp.s-<type expression>.baseType0.procedureFormalParameterList0

endlet

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

procedureSignatureParameterList0(ps: <procedure signature>):<formal parameter>\* =def

ps.**s**-<formal parameter>-seq

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

formalParameterList1

(d: Agent-type-definition∪Composite-state-type-definition∪Procedure-definition):

Parameter\*=def

d.localFormalParameterList1⁀d.inheritedFormalParameterList1

localFormalParameterList1

(d: Agent-type-definition∪Composite-state-type-definition∪Procedure-definition):

Parameter\*=def

case d of

| Agent-type-definition then d.s-Agent-formal-parameter-seq

| Composite-state-type-definition then d.s-Composite-state-formal-parameter-seq

| Procedure-definition then d.s-Procedure-formal-parameter-seq

otherwise empty endif

endcase

inheritedFormalParameterList1(d: Agent-type-definition∪Composite-state-type-definition∪

Procedure-definition): Parameter\*=def

let d1 = take({d1∈Agent-type-definition∪Composite-state-type-definition∪Procedure-definition:

isDirectSuperType1(d1, d)}) in

if d1≠undefined then d1. formalParameterList1

else empty endif

endlet

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

isActualAndFormalParameterMatched1

(expl:[Expression]\*, fpsl:Sort-reference-identifier\*):Boolean= def

(expl.length = fpsl.length)∧

(∀i∈1..expl.length:expl[i]=undefined∨isCompatibleTo1(expr[i].staticSort1, fpsl[i]))

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

formalParameterSortList1

(d: Agent-type-definition∪Composite-state-type-definition∪Procedure-definition∪

Operation-signature):Sort-reference-identifier\*= def

case d of

| Agent-type-definition∪Composite-state-type-definition∪Procedure-definition then

<param.s-Sort-reference-identifier | param in d.formalParameterList1>

| Operation-signature then <fa.s-Argument | fa in d.s-Formal-argument-seq>

| Signal-definition∪Timer-definition then d.s-Sort-reference-identifier-seq

endcase

Get the set of state nodes included in a type definition, state node or a state partition.

stateNodeSet1(d: TypeDefinition1∪State-node∪State-partition): State-node-set =def

case d of

| TypeDefinition1 then d.localStateNodeSet1∪d.inheritedStateNodeSet1

| State-node then

if d.s-Composite-state-type-identifier ≠ undefined then d.baseType1.stateNodeSet1∪{d}

else {d}

| State-partition then d.baseType1.stateNodeSet1

otherwise ∅

endcase

Get the set of state nodes defined locally in a type definition.

localStateNodeSet1(d: TypeDefinition1): State-node-set =def

case d of

| Agent-type-definition then

if d.State-machine ≠ undefined then

d. State-machine.baseType1.stateNodeSet1

else ∅

| Procedure-definition then

{sn.stateNodeSet1: sn∈d.s-Procedure-graph.s-State-node-set}

| Composite-state-type-definition then

if d.s-implicit∈Composite-state-graph then

{sn.stateNodeSet1: sn∈d.s-implicit.s-State-transition-graph.s-State-node-set}

else // d.s-implicit∈State-aggregation-node

{sp.stateNodeSet1: sp∈d.s-implicit.s-State-partition-seq.toSet}

otherwise ∅

endcase

Get the set of state nodes defined in a super type.

inheritedStateNodeSet1(d: TypeDefinition1): State-node-set =def

case d of

| Agent-type-definition then

if d.s-Agent-type-identifier≠ undefined then

getEntityDefinition1(d.s-Agent-type-identifier, agent type).stateNodeSet1

else ∅

| Procedure-definition then

if d.s-Procedure-identifier≠ undefined then

getEntityDefinition1(d.s-Procedure-identifier, procedure).stateNodeSet1

else ∅

| Composite-state-type-definition then

if d.s-Composite-state-type-identifier ≠ undefined then

getEntityDefinition1(d.s-Composite-state-type-identifier, state type).stateNodeSet1

else ∅

otherwise ∅

endcase

The function expandFormalVarParams0 expands an <formal variable parameters> list so that each <parameters of sort> contains only one <name>.

expandFormalVarParams0(fvps: <formal variable parameters>\*):<formal variable parameters>\* =def

if fvps = empty then empty

else

if fvps.head.s-<parameters of sort>.s-<name>-seq.length = 1

then fvps.head

else < fvps.head.s-<parameter kind>, fvps.head.s-<parameter aggregation>,

mk-<parameters of sort>( < fvps.head.s-<parameters of sort>.s-<name>-seq.head >,

fvps.head.s-<parameters of sort>.s-<sort>) > ⁀

expandAgentFormalParams0(< fvps.head.s-<parameter kind>,

fvps.head.s-<parameter aggregation>,

mk-<parameters of sort>( < fvps.head.s-<parameters of sort>.s-<name>-seq.tail>,

fvps.head.s-<parameters of sort>.s-<sort>

) >)

endif ⁀

if fvps.length =1 then empty

else expandAgentFormalParams0(fvps.tail)

endif

endif

### F2.2.7 Communication

#### F2.2.7.1 Channel definition

Abstract syntax

Channel-definition :: Channel-name

[ *Encoding-rules* ]

[ NODELAY ]

Channel-path-set

Channel-path :: Channel-endpoint

Originating-gate

Channel-endpoint

Destination-gate

Signal-identifier-set

Originating-gate = Gate-identifier

Destination-gate = Gate-identifier

Channel-endpoint :: Agent-identifier

| State-identifier

| ENV

Conditions on abstract syntax

See 10.1 *Abstract grammar* of [ITU-T Z.101] except the condition on *Encoding-rules*.

∀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 ⇒

(∀p1,p2∈c.s-Channel-path: p1≠p2 ⇒

p1.s-Originating-gate = p2.s-Destination-gate ∧ p2.s-Originating-gate = p1.s-Destination-gate)

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

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

(p.s1-Channel-endpoint = p.s2-Channel-endpoint) ∧ \\ same agent

∀g1, g2 ∈ Gate-definition: (p.parentAS1= c) ∧ (g1.parentAS1= d) ∧ (g2.parentAS1= d) ∧ (g1 ≠ g2) ∧

(p.s-Originating-gate=g1.identifier1) ∧ (p.s-Destination-gate= g2.identifier1) ⇒

|c.s-Channel-path-set|=1

If the Originating-gate and the Destination-gate are gates of the same agent, the channel shall be unidirectional (there shall be only one element in the Channel-path-set).

∀c ∈ Channel-definition: ∀p ∈ Channel-path: ∀d ∈ Composite-state-type-definition:

(p.s1-Channel-endpoint = p.s2-Channel-endpoint) ∧ \\ same state machine

∀g1, g2 ∈ Gate-definition: (p.parentAS1= c) ∧ (g1.parentAS1= d) ∧ (g2. parentAS1= d) ∧ (g1 ≠ g2) ∧

(p.s-Originating-gate=g1.identifier1) ∧ (p.s-Destination-gate= g2.identifier1) ⇒

|c.s-Channel-path-set|=1

If the Originating-gate and the Destination-gate are both gates of the same state machine, the channel shall be unidirectional (there shall be only one element in the Channel-path-set).

∀c ∈ Channel-definition: ∀p ∈ Channel-path: ∃g1, g2 ∈ Gate-definition:

(p.parentAS1= c) ∧

( ((p.s-Originating-gate=g1.identifier1) ∧

(g1.parentAS1= c.parentAS1 ∨ g1.parentAS1.parentAS1= c.parentAS1)) ∧

((p.s-Destination-gate= g2.identifier1) ∧

(g2*.*parentAS1=c*.*parentAS1 ∨ g2.parentAS1.parentAS1= c.parentAS1))

)

The Originating-gate and Destination-gate shall be defined in the same scope unit (which includes directly enclosed scopes) in the abstract syntax in which the channel is defined.

∀c ∈ Channel-definition: ∀p ∈ c.s-Channel-path-set:

getEntityDefinition1(p.s-Originating-gate,gate).s-Encoding-rules = c.s-Encoding-rules ∧

getEntityDefinition1(p.s-Destination-gate,gate).s-Encoding-rules = c.s-Encoding-rules

The *Originating-gate* or *Destination-gate* shall have the same *Encoding-rules* as the *Channel-definition*. If the *Channel-definition* has no *Encoding-rules*, neither the *Originating-gate* nor *Destination-gate* shall have *Encoding-rules*. See clause 10.1 *Abstract grammar* of [ITU-T Z.104]

Concrete syntax

<channel definition> ::

[<channel<name> [ <encoding rules>]] [nodelay] <channel path> [<channel path>]

<channel path> :: <channel endpoint> <channel endpoint> [ <signal list> ]

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

<as channel> :: <channel<identifier>

Conditions on concrete syntax

∀ce∈<channel endpoint>:

let g = ce.s-<gate> in

case id = ce.s-implicit of

| <identifier> then // agent identifier or state identifier

case d = getEntityDefinition0(id, agent ∪ state) of

| <textual typebased agent definition> ∪ <typebased composite state> then

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

g ≠ undefined ∧ // the gate is specified in the channel endpoint

(∃gd∈<textual gate definition>: ∃gc∈<gate constraint>: // exists a gate definition and constraint

gd.name0=g∧ // gate definition has the gate name

gd∈td.localGateDefinitionSet0∧ // gate definition defined in the agent/state type

gc.parentAS0 =gd∧ // is a gate constraint of the gate definition

// at least one common element in the signal lists

gc.s-<signal list item>-seq.signalSet0 ∩ ce.parentAS0.s-<signal list item>-seq.signalSet0≠∅∧

gc.direction0 =ce.direction0 )// in the same direction

endlet // td

otherwise true // agent/state machine not typebased, condition must hold when transformed to typebased

endcase // d

| **env** then // environment

let su = ce.surroundingScopeUnit0 in

su ∈ <agent type definition> ) ⇒

g ≠ undefined ∧ // the gate is specified in the channel endpoint

(∃gd∈<textual gate definition>: ∃gc∈<gate constraint>: // exists a gate definition and constraint

gd.name0=g∧ // gate definition has the gate name

gd∈su.localGateDefinitionSet0∧ // gate definition defined in the agent type

gc.parentAS0 =gd∧ // is a gate constraint of the gate definition

// at least one common element in the signal lists

gc.s-<signal list item>-seq.signalSet0 ∩ ce.parentAS0.s-<signal list item>-seq.signalSet0≠∅∧

gc.direction0 =ce.direction0 ) // in the same direction

endlet // su

| **this** then // the state machine of the enclosing agent

let su = ce.surroundingScopeUnit0 in

let sm = su.s-<agent structure>.s-<interaction>.s-<typebased composite state> in // the state machine

let td = sm.s-<type expression>.baseType0 in

g ≠ undefined ∧ // the gate is specified in the channel endpoint

(∃gd∈<textual gate definition>: ∃gc∈<gate constraint>: // exists a gate definition and constraint

gd.name0=g∧ // gate definition has the gate name

gd∈td.localGateDefinitionSet0∧ // gate definition defined in the state type

gc.parentAS0 =gd∧ // is a gate constraint of the gate definition

// at least one common element in the signal lists

gc.s-<signal list item>-seq.signalSet0 ∩ ce.parentAS0.s-<signal list item>-seq.signalSet0≠∅∧

gc.direction0 =ce.direction0 )// in the same direction

endlet // td

endlet // sm

endlet // su

otherwise false // condition not met -

endcase // id

endlet // g

<gate> shall be specified if:

a) <channel endpoint> with an <agent<identifier> denotes a connection to a <textual typebased agent definition> in which case the <gate> shall be defined directly in the agent type for that agent; or

b) <channel endpoint> with a <state<identifier> identifies the state machine of the enclosing agent, in which case the <gate> shall be defined directly in the state type for that state, and as there is only one state machine it can alternatively be identified by this; or

c) env is specified and the channel is defined in an agent type in which case the <gate> shall be defined in this agent type.

If gate is specified the channel is connected to that gate. The gate and channel shall have at least one common element in their signal lists in the same direction. See clause 5.6.1 of [ITU‑T Z.106].

∀c ∈ <channel definition>:

(c.s1-<channel path>.s-<signal list> = undefined ⇒ c.s1-<channel path>.derivedSignalList0 ≠ empty )

∧ (c.s2-<channel path> ≠ undefined ) ⇒

c.s2-<channel path>.s-<signal list> = undefined ⇒ c.s2-<channel path>.derivedSignalList0 ≠ empty )

If any associated <signal list> is omitted from the <channel path> of a <channel definition>, the corresponding set of signals shall be derivable. See clause 10.1 *Concrete grammar* of [ITU‑T Z.103].

Derivation of an omitted channel <signal list> is possible if at least one <channel endpoint> identifies a <textual typebased agent definition> or <typebased composite state> (or this and the state machine is a <typebased composite state>) or env, and the gate for the <channel endpoint> has a defined set of signals in the direction for the <signal list>. The set of signals is defined for the gate if it identifies a <textual gate definition> that has a <gate constraint> with a defined <signal list>, or if a signal set is defined for all internal channels connected to this gate. The set is defined for a gate connected to <external channel identifiers> if for each external channel either no <signal list area> is omitted or the set for that external channel is derivable. See clause 5.6.1 of [ITU‑T Z.106] which describes derivation for the textual grammar and is based on clause 10.1 *Concrete grammar* of [ITU‑T Z.103].

Transformations

<channel definition>(undefined, delay, p1, p2)

=5=>

<channel definition>(newName, delay, p1, p2)

If the <channel name> is omitted from a <channel definition>, the channel is implicitly and uniquely named. See clause 10.1 *Model* of [ITU‑T Z.103].

<channel definition>(name, delay, p1, p2)

provided

p1.s-<signal list> = undefined

=8=>

<channel definition>(

name,

delay,

mk-<channel path>( p1.s1-<<channel endpoint>, p1.s2-<<channel endpoint>, p1.derivedSignalList0),

p2)

<channel definition>(name, delay, p1, p2)

provided

p2 ≠ undefined

∧ p2.s-<signal list> = undefined

=13=>

<channel definition>(

name,

delay,

p1,

mk-<channel path>( p2.s1-<<channel endpoint>, p2.s2-<<channel endpoint>, p2.derivedSignalList0)

)

If an associated <signal list> is omitted from a <channel definition>, the <signal list> is replaced by a <signal list> derived from the channel connection (see condition on <signal list> in a <channel definition> above), that corresponds to the *In-signal-identifier****-set*** or *Out-signal-identifier****-set*** for the *Destination-gate* or *Originating-gate*. Modified from clause 10.1 *Model* of [ITU‑T Z.103].

NOTE − This transform takes place after the transform of shorthand notations <agent definition>, <agent body> and <composite state definition> in step12, so that the channel endpoint is typebased or this or env, and a gate has to be specified.

t = <textual typebased agent definition>

provided unconnectedGates(t) = undefined

=9=>

t

and

unconnectedGates(t)=>

let id = take({ te.s-<base type> | te ∈ <type expression>: te.parentAS0.parentAS0 = t }) in

{ g ∈ id.refersto0.getEntities0.toSet:

g ∈ <gate in definition> ∧ ¬isConnected(g, undefined, id.refersto0.getEntities0.toSet) }

endlet

t = <agent type definition>

provided unconnectedGates(t) = undefined

=9=>

t

and

unconnectedGates(t)

=>

{ g ∈ t.getEntities0.toSet: g ∈ <gate in definition> ∧ ¬isConnected(g, undefined, t.getEntities0.toSet) }

< a1 = <textual typebased agent definition> > ⁀ something ⁀ < a2 = <textual typebased agent definition> >

provided missingConnections(a1,a2) ≠ ∅

=10=>

let c = <channel definition>(newName, undefined, missingConnections(a1,a2).take, undefined) in

< a1 > ⁀ something ⁀ < a2 > ⁀ < c >

endlet

< a1 = <textual typebased agent definition> > ⁀ something ⁀ < a2 = <textual typebased agent definition> >

provided missingConnections(a2,a1) ≠ ∅

=10=>

let c = <channel definition>(newName, undefined, missingConnections(a2,a1).take, undefined) in

< a1 > ⁀ something ⁀ < a2 > ⁀ < c >

endlet

< a = <textual typebased agent definition> >

provided missingConnections(a,a.parentAS0.parentAS0) ≠ ∅

=10=>

let c = <channel definition>(newName, undefined,

missingConnections(a,a.parentAS0.parentAS0).take, undefined) in

< a > ⁀ < c >

endlet

< a = <textual typebased agent definition> >

provided missingConnections(a.parentAS0.parentAS0,a) ≠ ∅

=10=>

let c = <channel definition>(newName, undefined,

missingConnections(a.parentAS0.parentAS0,a).take, undefined) in

< a > ⁀ < c >

endlet

*The text for the three transforms below is from clause 10.1 Model of [ITU‑T Z.103].*

If an agent or agent type contains explicit or implicit gates that are not attached by explicit channels, implicit channels are derived according to the following three transforms, which are applied after the transform for typebased creation is applied.

Transform 1:

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

Transform 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.

Transform 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 transforms one signal list element (interface, signal, remote procedure or remote variable) matches another signal list element if:

a) both denote the same interface, signal, remote procedure or remote variable; or

b) the first denotes a signal or remote procedure or remote variable, and the second denotes an interface and the interface includes the signal or remote procedure or remote variable; or

c) both denote interfaces and the second signal list element inherits the first signal list element.

Transform 1: Insertion of implicit channels between entities inside one agent or agent type

a) If an element of the outgoing signal list associated with a gate of an instance in an agent (or agent type) matches an element of an incoming signal list associated with a gate of another instance in the same agent (or agent type respectively); and

b) if neither of these gates has an explicit channel attached to it,

then

a) if no implicit channel exists between the two gates, a unidirectional implicit channel is created from the gate where the element is outgoing to the gate where the element is incoming, and this channel is non-delaying if it is within a process (or process type) and otherwise it is delaying; and

b) the element is added to the signal list of the implied channel.

Transform 2: Insertion of implicit channels from the gates on an agent or agent type

a) If an element of the incoming signal list associated with a gate outside an agent (or agent type) matches an element of an incoming signal list associated with a gate of an instance in the agent (or agent type respectively); and

b) if there is no explicit channel inside the agent (or agent type respectively) attached to the gate outside the agent (or agent type respectively) and no explicit channel attached to the gate of the instance inside the agent (or agent type respectively),

then

a) if no implicit channel exists between the two gates, a unidirectional implicit channel is created from the gate outside the agent (or agent type respectively) to the gate of the instance inside the agent (or agent type respectively), and this channel is non-delaying if it is within a process (or process type) and otherwise it is delaying; and

b) the element is added to the signal list of the implied channel.

Transform 3: Insertion of implicit channels from the gates on instances

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:

a) If an element of the outgoing signal list associated with a gate outside an agent (or agent type) matches an element of an outgoing signal list associated with a gate of an instance in the agent (or agent type respectively); and

b) if there is no explicit channel attached to the gate outside the agent (or agent type respectively) and no explicit channel connected to the gate of the instance inside the agent (or agent type respectively),

then

a) if no implicit channel exists between the two gates in the direction to the gate outside the agent (or agent type respectively), a unidirectional implicit channel is created from the gate of the instance inside the agent (or agent type respectively) to the gate outside the agent (or agent type respectively), and this channel is non-delaying if it is within a process (or process type) and otherwise it is delaying; and

b) the element is added to the signal list of the implied channel.

*The following statement is modelled in the dynamic semantics.*

A channel with both endpoints being gates of one <textual typebased agent definition> represents individual channels from each of the agents in this set to all agents in the set, including the originating agent. Any resulting bidirectional channel connecting an agent in the set to the agent itself is split into two unidirectional channels. Modified from clause 10.1 *Model* of [ITU‑T Z.103].

cd.asChannelName

provided

cd ∈ <channel definition>

∧ cd.asChannelName = undefined

=8=>

(asChannelName \{( cd, undefined )} ) ∪ {( cd, newName )}

NOTE – Each <channel definition> defines a *Data-type-definition* of a choice data type with a unique anonymous *Sort* name stored in asChannelName.

<as channel>(cid)

provided

cid.refersto0.asChannelName ≠ undefined

=8=>

mk-<type expression>(mk-<identifier>(gid.**s**-<qualifier>, gid.refersto0.asChannelName), undefined)

An <as channel> represents the sort of the choice data type introduced by the identified channel definition. See clause 12.1 *Concrete grammar* of [ITU‑T Z.104].

Mapping to abstract syntax

| <channel definition>(name, encoding, delayProperty, path1, path2) then

mk-Channel-definition(Mapping(name),

if encoding ≠ undefined

then Mapping(encoding)

else // encoding rules omitted in channel definition

if path1.pathEncodings0 ≠ undefined then Mapping(path1.pathEncodings0)

else

if path2 ≠ undefined

then

if path2.pathEncodings0 ≠ undefined then Mapping(path2.pathEncodings0)

else undefined

endif // path2 encoding defined/undefined

else undefined

endif // path2 defined/undefined

endif // path1 encoding defined/undefined

endif, // encoding rules in channel definition

if delayProperty = nodelay then NODELAY else undefined endif,

if path2=undefined

then { Mapping(path1) }

else { Mapping(path1), Mapping(path2) }

endif)

If the <encoding rules> item is omitted in a <channel definition> and a <channel path> is connected to a <gate> that identifies a <gate definition> with <encoding rules>, the *Channel-definition* has the same *Encoding-rules* as the *Gate-definition* of the <textual gate definition>. If the *Gate-definition* has no *Encoding-rules,* the *Channel-definition* has no *Encoding-rules*. Derived from See clause 10.1 *Model* of [ITU‑T Z.104].

NOTE – The mapping to a *Data-type-definition* for the choice data type defined by a <channel definition> in the following paragraph takes place by invoking the function choiceForChannel1 (defined below), in the mapping to *Agent-type-definition* in F2.2.5.1.2, F2.2.5.1.3 and F2.2.5.1.4.

A *Channel-definition* implicitly defines a *Data-type-definition* of a choice data type with a unique anonymous *Sort* name in the context that the *Channel-definition* is visible. A <basic sort> that is <as channel> where <channel<identifier> identifies the *Channel-definition* represents this *Data-type-definition*. The *Data-type-definition* is equivalent to defining a choice data type with a <choice of sort> for each distinct *Signal-definition* identified by each *Signal-identifier***-set**of the *Channel-path***-set**. The <field name> and <field sort> for each <choice of sort> is determined in the same way as a <choice of sort> for each distinct *Signal-definition* of the *Data-type-definition* defined by a <gate definition>. See clause 10.1 *Concrete* *grammar* of [ITU‑T Z.104].

| <channel path>(endp1(\*, g1), endp2(\*, g2), with) then

mk-Channel-path(Mapping(endp1), Mapping(g1), Mapping(endp2), Mapping(g2), Mapping(with))

| <channel endpoint>( end, \*) then

mk-Channel-endpoint(

case end of

| <identifier> then Mapping(end)

| env then ENV

| this then

let atd = parentAS0ofKind(end,

<system type definition> ∪ <block type definition> ∪ <process type definition>)

in

mk-Identifier(Mapping(atd.identifier0.s-<qualifier>) ⁀

< mk-Agent-type-qualifier(Mapping(atd.identifier0.s-<name>) >,

Mapping(atd.s-<agent structure>.s-<interaction>.s-<state machine>.s-<name>))

endlet

endcase)

Auxiliary functions

SignalDirection0 =def { out, in }

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

direction0(p: <gate constraint>∪<channel endpoint>): SignalDirection0=def

case p of

| <channel endpoint> then if p=p.parentAS0.s1-<channel endpoint> then out else in endif

| <gate constraint> then if p.s-in ≠ undefined then in else p.s-out endif // s-out could be undefined

otherwise undefined

endcase

The function channelEndpointReferTo0 is used to get the entity definition that the <channel endpoint> referred to.

channelEndpointReferTo0(ep:<channel endpoint>): EntityDefinition0=def

let end = ep.s-implicit in

case end of

| <identifier> then getEntityDefinition0(end, end.idKind0)

| this then parentAS0ofKind(end, <agent definition> ∪<agent type definition> )

| env then undefined

endcase

endlet

The function unconnectedGates is used to store the gates that are not explicitly connected.

controlled unconnectedGates: <textual typebased agent definition> ∪ <agent type definition> →

<gate in definition>-set

initially ∀ def ∈ (<textual typebased agent definition> ∪ <agent type definition>):

def.unconnectedGates = undefined

The function missingConnections is used to compute the missing implicit connections between two agents.

missingConnections(ag1: <textual typebased agent definition> ∪ <agent type definition>,

ag2: <textual typebased agent definition> ∪ <agent type definition>): <channel path>-set =def

let entities =

if ag1 ∈ <agent type definition> then ag1.getEntities0

elseif ag2 ∈ <agent type definition> then ag2.getEntities0

else parentAS0ofKind(ag1, <agent type definition>).getEntities0

endif

in

let id1 =

if ag1 ∈ <agent type definition> then env else ag1.identifier0 endif

in

let id2 =

if ag2 ∈ <agent type definition> then env else ag2.identifier0 endif

in

**U** { { mk-<channel path>(

mk-<channel endpoint>(id1, g1),

mk-<channel endpoint>(id2, g2),

< siglistitem ∈ (inoutSignals0(g1, out) ∪ inoutSignals0(g2, in)) > ) // channel path

: g2 ∈ ag2.unconnectedGates ∧ ((inoutSignals0(g1, out) ∪ inoutSignals0(g2, in)) ≠ ∅) ∧

¬isConnected(g1, g2, entities)}

: g1 ∈ ag1.unconnectedGates }

The function isConnected is used to check whether two gates are connected.

isConnected(g1:<gate in definition>, g2: <gate in definition>, ent: DefinitionAS0-set): Boolean =def

let allPaths =

**U**{ { e.s1-<channel path> } ∪

if e.s2-<channel path> = undefined then ∅ else { e.s2-<channel path> } endif

| e ∈ ent.toSet: e ∈ <channel definition> }

in

∃ p ∈ allPaths: g1=p.s1-<channel endpoint>.s-<gate>.refersto0 ∧

(g2=p.s2-<channel endpoint>.s-<gate>.refersto0 ∨ g2 = undefined)

endlet

The function inoutSignals0 is used to compute the outbound or inbound signals of a gate based on the <gate constraint>.

inoutSignals0(g: <textual gate definition>, kind: SignalDirection0 ): <signal list item>-set =def

if kind: = in then

{ sli : sli in gc.s1-<signal list item>-seq ∧ gc ∈ g.s-<gate constraint> ∧ gc.direction0 = in }

else // kind is out

{ sli : sli in gc.s2-<signal list item>-seq ∧ gc ∈ g.s-<gate constraint> ∧ gc.direction0 = in } ∪

{ sli : sli in gc.s1-<signal list item>-seq ∧ gc ∈ g.s-<gate constraint> ∧ gc.direction0 = out }

endif // kind

The function pathEncodings0 is used to get the encoding rules (if any) specified by one of gates (if any) at the channel endpoints of the path. If both channel endpoints have a gate with encoding rules, these rules are required to be the same.

pathEncodings0(cp: <channel path>): <encoding rules> =def

let localQualifier = cp.surroundingScopeUnit0.fullQualifierWithin0 in

let g1 = cp.s1-<channel endpoint>.s-<gate> in

let g2 = cp.s2-<channel endpoint>.s-<gate> in

if g1 ≠ undefined then

let rules1 = mk-<identifier>(localQualifier,g1).refersto0.s-<encoding rules> in

if rules1≠ undefined then rules1

else

if g2 ≠ undefined then mk-<identifier>(localQualifier,g2).refersto0.s-<encoding rules>

else undefined // g2 undefined, g1 defined but no encoding rules

endif // rules2 undefined

endif

endlet // rules1

else // g1 undefined

if g2 ≠ undefined then

let rules2 = mk-<identifier>(localQualifier,g2).refersto0.s-<encoding rules> in

if rules2≠ undefined then rules2 else undefined endif

endlet // rules2

else undefined // g1 undefined and g2 undefined

endif

endlet // g2

endlet // g1

endlet // localQualifier

The function choiceForChannel1 produces the choice Data-type-definition that corresponds to a Channel-definition and is used when an <as channel> is given for a <basic sort>, where each <choice of sort> is for each distinct signal definition identified by each signal identifier that is a signal list item of the signal list of a channel path of the channel definition.

choiceForChannel1(cd: <channel definition>): Data-type-definition =def

let chanSigSet =

{ sig : sig in cd.s1-<channel path>.s-<signal list item>-seq

∨ ( cd.s2-<channel path> ≠ undefined

∧

sig in cd.s2-<channel path>.s-<signal list item>-seq

)

} // chanSigSet

in

Mapping(mk-<data type definition>( mk-<type preamble>(undefined,undefined), // type preamble

mk-<data type heading>( cd.asChannelName, // name for choice type

undefined, undefined, // no formal parameters or virtuality constraint

), // data type heading

mk-<data type definition body>( empty, // no entities in data type

mk-<choice definition>( undefined, // visibility

< mk-<choice of sort>( undefined, // visibility

mk-<field of kind>( part,

if sig.s-<name> ∉ { othersigs.s-<name>

: othersigs ∈ (chanSigSet \ sig )}

then // unique signal name

sig.s-<name>

else // not unique signal name – name same as <as signal>

sig.asSignal.s-<name>// name of structure sort for signal

endif

), // <field of kind>

mk-<field sort>(

if sig.refersto0.s-implicit // <sort list> | <named fields sort list> | <sort<identifier>

= undefined // no sort

then predefinedId0("NULL")

else sig.asSignal // structure sort for signal

endif

) // <field sort>

) // <choice of sort> item

: sig ∈ chanSigSet

> // <choice of sort> list

) <choice definition>(

), // data type definition body

undefined // default initialization

)) // Mapping of data type definition

endlet // chanSigSet

The function derivedSignalList0 produces a derived list of signal <identifier> items, to be used in a <channel path> of a <channel definition> when a <signal list> is omitted from the <channel path> of a <channel definition>. The description of the derivation is in the text for the condition for the missing <signal list> items to be derivable.

derivedSignalList0(cp: <channel path>): <signal<identifier>\* =def

let orig = cp.s1-<channel endpoint> in

let origgate = orig.s-<gate> in

let origgateDef =

case orig.s-implicit of // first item of channel endpoint

| <identifier> then mk-<identifier>(orig.s-implicit.refersto0.fullQualifierWithin0, origgate).refersto0

| this then

mk-<identifier>(

this.surroundingScopeUnit0.s-<typebased composite state>.refersto0.fullQualifierWithin0,

origgate

).refersto0

| env then

mk-<identifier>(env.surroundingScopeUnit0.fullQualifierWithin0, origgate).refersto0

otherwise undefined // any other cases

endcase // first item of channel endpoint

in

let dest = cp.s2-<channel endpoint> in

let destgate = dest.s-<gate> in

let destgateDef =

case dest.s-implicit of // first item of channel endpoint

| <identifier> then mk-<identifier>(dest.s-implicit.refersto0.fullQualifierWithin0, destgate).refersto0

| this then

mk-<identifier>(

this.surroundingScopeUnit0.s-<typebased composite state>.refersto0.fullQualifierWithin0,

destgate

).refersto0

| env then

mk-<identifier>(env.surroundingScopeUnit0.fullQualifierWithin0, destgate).refersto0

otherwise undefined // any other cases

endcase // first item of channel endpoint

in

if orig.s-implicit ∈ <identfier>

∧ orig.s-implicit.refersto0 ∈ <textual typebased agent definition> ∪ <typebased composite state>

∧ ( (origgate ≠ undefined ∧ inoutSignals0(origgateDef, out) ≠ ∅) // orig defined with out signals

∨ // or orig connected to channels in agent that have out signals

( ( ∀ connectedchf ∈ connectedChanSet(orig, true) // connected forward path –

: connectedchf.s1-<channel path>.s-<signal list> ≠ undefined

) // forward channel path – all connected channels – out signals defined

∧

( ∀ connectedchr ∈ connectedChanSet(orig, false) // connected reverse path – that is

: connectedchr.s2-<channel path>.s-<signal list> ≠ undefined

) // reverse channel paths – all connected channels – out signals defined, if path defined

) // true if out signals defined for all forward paths, and all reverse paths defined

) // true if orig defined with signals, or channels in orig agent have signals defined

then

if (origgate ≠ undefined ∧ inoutSignals0(origgateDef, out) ≠ ∅)

then // orig defined with out signals

< sig : sig ∈ inoutSignals0(origgateDef, out) >

else // connection to agent with signals defined for all connected chans

< sig : sig ∈ toSet(bigSeq(

< connectedchf.s1-<channel path>.s-<signal list> : connectedchf ∈ connectedChanSet(orig, true) > ⁀

< connectedchr.s2-<channel path>.s-<signal list> : connectedchr ∈ connectedChanSet(orig, false) >

)) > // bigSeq, toSet, sequence constructor

endif

elseif

orig.s-implicit = this ∧ origgate ≠ undefined

∧ this.surroundingScopeUnit0.s-<typebased composite state> ≠ undefined

then

< sig : sig ∈ inoutSignals0(origgateDef, out) >

elseif

orig.s-implicit = env ∧ origgate ≠ undefined

then

< sig : sig ∈ inoutSignals0(origgateDef, in) >

elseif // cannot derive from orig, therefore try dest

dest.s-implicit ∈ <identfier>

∧ dest.s-implicit.refersto0 ∈ <textual typebased agent definition> ∪ <typebased composite state>

∧ ( (destgate ≠ undefined ∧ inoutSignals0(destgateDef, in) ≠ ∅) // dest defined with in signals

∨ // or dest connected to channels in agent that have out signals

( ( ∀ connectedchf ∈ connectedChanSet(dest, true) // connected forward path –

: connectedchf.s1-<channel path>.s-<signal list> ≠ undefined

) // forward channel path – all connected channels – in signals defined

∧

( ∀ connectedchr ∈ connectedChanSet(dest, false) // connected reverse path – that is

: connectedchr.s2-<channel path>.s-<signal list> ≠ undefined

) // reverse channel paths – all connected channels – in signals defined, if path defined

) // true if out signals defined for all forward paths, and all reverse paths defined

) // true if dest defined with signals, or channels in dest agent have signals defined

then

if (destgate ≠ undefined ∧ inoutSignals0(destgateDef, in) ≠ ∅)

then // dest defined with in signals

< sig : sig ∈ inoutSignals0(destgateDef, in) >

else // connection to agent with signals defined for all connected chans

< sig : sig ∈ toSet(bigSeq(

< connectedchf.s2-<channel path>.s-<signal list> : connectedchf ∈ connectedChanSet(dest, true) > ⁀

< connectedchr.s1-<channel path>.s-<signal list> : connectedchr ∈ connectedChanSet(dest, false) >

)) > // bigSeq, toSet, sequence constructor

endif

elseif

dest.s-implicit = this ∧ destgate ≠ undefined

∧ this.surroundingScopeUnit0.s-<typebased composite state> ≠ undefined

then

< sig : sig ∈ inoutSignals0(destgateDef, in) >

elseif

dest.s-implicit = env ∧ destgate ≠ undefined

then

< sig : sig ∈ inoutSignals0(destgateDef, out) >

else // signals not derivable

empty

endif

The function connectedChanSet produces a set of channels connected to a channel endpoint in the agent identified by the channel endpoint in the forward (or reverse) direction.

connectedChanSet(ce: <channel endpoint>, forward: Boolean): <channel defnition>-set =def

{ch : ch ∈ <channel defnition> // channels

∧ ce.s-implicit.refersto0 = ch.surroundingScopeUnit0 // ch internal to agent at ce

∧ if forward

then // forward paths

ch.s1-<channel path>.s2-<channel endpoint>.s-implicit = env // to env

∧ ch.s1-<channel path>.s2-<channel endpoint>.s-<gate> = ce.s-<gate> // via ce

else // reverse paths

ch.s2-<channel path> ≠ undefined // reverse path defined

∧ ch.s2-<channel path>.s2-<channel endpoint>.s-implicit = env // to env

∧ ch.s2-<channel path>.s2-<channel endpoint>.s-<gate> = ce.s-<gate> // via ce

endif

} // all internal channels connected to ce with forward (reverse) path

The function asChannelName associates each <channel definition> with a unique anonymous *Sort* name for the corresponding *Data-type-definition* of the choice data type invoked by <as channel>.

controlled asChannelName: <channel definition>→ <name>

initially ∀ cd ∈ <channel definition>: gd.asChannelName = undefined

#### F2.2.7.2 Connections

Concrete syntax

<channel to channel connection> ::

<external channel identifiers> <channel<identifier>+

<external channel identifiers> = <channel<identifier>+

Conditions on concrete syntax

∀c2c ∈ <channel to channel connection>: c2c.surroundingScopeUnit0 ∉ <agent type definition>

A <channel to channel connection> shall not be directly contained within an <agent type definition>. See clause 5.5 of [ITU-T Z.106].

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

( let ids1= c1.s2-<identifier>-seq.toSet in

let ids2= c2.s2-<identifier>-seq.toSet in

c1.surroundingScopeUnit0= c2.surroundingScopeUnit0∧ c1 ≠ c2⇒ ids1∩ ids2 = ∅

endlet

endlet)

It is not allowed to mention a channel after the keyword **and** in more than one <channel to channel connection> of a given scope unit. See clause 5.6.2 of [ITU-T Z.106].

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

(let ids1= c1.s-<identifier>-seq.toSet in

let ids2= c2.s-<identifier>-seq.toSet in

(c1.surroundingScopeUnit0= c2.surroundingScopeUnit0∧ c1 ≠ c2)

⇒(ids1 = ids2 ∨ ids1∩ ids2 = ∅)

endlet

endlet)

For any pair of <channel to channel connection> items of a given scope unit, the <external channel identifiers>s shall either mention the same set of channels, or shall have no channels in common. See clause 5.6.2 of [ITU-T Z.106].

Transformations

ctcc.myImplicitGateIdentifier

provided

ctcc ∈ <channel to channel connection>

∧ ctcc. myImplicitGateIdentifier = undefined

=8=>

(myImplicitGateIdentifier \{( ctcc, undefined )} )

∪ {( ctcc, mk-<identifier>(surroundingQualifier0(ctcc), newName))}

ctcc = <channel to channel connection>(\*, \*)

provided

ctcc.myImplicitGateIdentifier ≠ undefined

=8=>

mk-<textual gate definition>( ctcc.myImplicitGateIdentifier.s-<name>, undefined, // name, no encoding rules

if allSignalsIn(ctcc).length >1 then

if allSignalsOut(ctcc).length >1 then

mk-<gate constraint>(in, allSignalsIn(ctcc), out, allSignalsOut(ctcc))

else

mk-<gate constraint>(in, allSignalsIn(ctcc),undefined)

endif

else

if allSignalsOut(ctcc).length >1 then

mk-<gate constraint>( out, allSignalsOut(ctcc))

else mk-<gate constraint>(undefined) // should not occur

endif

)

and

cendpt=<channel endpoint>(id, undefined)

provided

findconnect(cendpt.parentAS0.parentAS0, id) ≠ undefined

=>

mk-<channel endpoint>(id, findconnect(cendpt.parentAS0.parentAS0, id))

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. 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. See clause 5.6.2 of [ITU-T Z.106].

Auxiliary functions

The auxiliary function myImplicitGateIdentifier stores the implicitly generated gate identifier of a connection.

controlled myImplicitGateIdentifier: <channel to channel connection> → <identifier>

initially ∀ ctcc ∈ <channel to channel connection>: ctcc.myImplicitGateIdentifier = undefined

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

findconnect(ch:<channel definition>, id: DefinitionAS0): <identifier> =def

if id=env then

let matchingGateIds =

{ c.myImplicitGateIdentifier | c ∈ <channel to channel connection>:

c.parentAS0 = ch.parentAS0 ∧ fullIdentifier0(ch) in c.s2-<identifier>-seq} in

matchingGateIds.take

else

let matchingGateIds =

{ c.myImplicitGateIdentifier | c ∈ <channel to channel connection>:

c.parentAS0 = id.refersto0 ∧ fullIdentifier0(ch) in c.s-<identifier>-seq} in

matchingGateIds.take

endif

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

allSignalsIn(c: <channel to channel connection>): <signal list item>\* =def

< sig : sig ∈ toSet( // removes duplicate signal list items

bigSeq( < // all external channels to scope round ctcc, or from scope round ctcc

if extchan.refersto0.s1-<channel path>.s2-<channel endpoint> = ctcc.surroundingScopeUnit0

then extchan.refersto0.s1-<channel path>.s-<signal list item>-seq

elseif extchan.refersto0.s2-<channel path> ≠ undefined

then extchan.refersto0.s2-<channel path>.s-<signal list item>-seq

else empty

endif // external channels to/from scope round ctcc

: extchan in ctcc.s1-<identifier>-seq // external channels of ctcc

> ) // list of signal list item lists, bigSeq – which flattens the list of lists

)> // sig, list of signal list items, toSet removes duplicates

⁀

bigSeq(

< // all internal channels from **env** to something, or from something to **env**

if intchan.refersto0.s1-<channel path>.s1-<channel endpoint> = **env**

then intchan.refersto0.s1-<channel path>.s-<signal list item>-seq

elseif intchan.refersto0.s2-<channel path> ≠ undefined

then intchan.refersto0.s2-<channel path>.s-<signal list item>-seq

else empty

endif // all internal channels to/from env

: intchan in ctcc.s2-<identifier>-seq // internal channels of ctcc

> ) // list of signal list item lists, bigSeq – which flattens the list of lists

) > // toSet, make set into sequence

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

allSignalsOut(ctcc: <channel to channel connection>): <signal list item>\* =def

< sig : sig ∈ toSet( // removes duplicate signal list items

bigSeq( < all external channels from scope round ctcc, or to scope round ctcc

if extchan.refersto0.s1-<channel path>.s1-<channel endpoint> = ctcc.surroundingScopeUnit0

then extchan.refersto0.s1-<channel path>.s-<signal list item>-seq

elseif extchan.refersto0.s2-<channel path> ≠ undefined

then extchan.refersto0.s2-<channel path>.s-<signal list item>-seq

else empty

endif

: extchan in ctcc.s1-<identifier>-seq // external channels of ctcc

> ) // list of signal list item lists, bigSeq – which flattens the list of lists

⁀

bigSeq(

< // all internal channels from **env** to something, or from something to **env**

if intchan.refersto0.s1-<channel path>.s1-<channel endpoint> = **env**

then

if intchan.refersto0.s2-<channel path> ≠ undefined

then intchan.refersto0.s2-<channel path>.s-<signal list item>-seq

else empty

endif

else intchan.refersto0.s1-<channel path>.s-<signal list item>-seq

endif

: intchan in ctcc.s2-<identifier>-seq // internal channels of ctcc

> ) // list of signal list item lists, bigSeq – which flattens the list of lists

) > toSet, make set into seqeunce

#### F2.2.7.3 Signal

Abstract syntax

Signal-definition:: Signal-name

Signal-parameter\*

[ Signal-identifier ]

[ Abstract ]

Signal-parameter :: Aggregation-kind Sort-reference-identifier

Concrete syntax

<signal definition list> ::<signal definition>+

<signal definition><signal definition> :: <type preamble>

<signal<name>

[ <formal context parameters> ]

[ <virtuality constraint>]

[ <specialization>]

[ <sort list> | <named fields sort list> | <sort<identifier>]

<named fields sort list> :: [ <visibility> ]

{ [ <visibility> ] <aggregation kind> <field<name> <sort> }+

<sort list> :: { <aggregation kind> <sort> }+

<as signal> :: <signal<identifier>

Conditions on concrete syntax

∀sdi∈<signal definition>:sdi.specialization0.s-<type expression>.s-<base type>.idKind0 = signal

The <base type> part of <specialization> shall be a <signal identifier>. See clause 12.3 *Concrete grammar* of [ITU‑T Z.102].

∀sid ∈ <identifier>:

sid.parentAS0∉<type expression>∧ sid.parentAS0∉<signal constraint>⇒

¬getEntityDefinition0(sid, signal).isAbstractType0

An abstract type shall not be instantiated. See clause 8.1.3 *Semantics* of [ITU‑T Z.102]. As a consequence, an abstract signal can only be used in specialization and signal constraints.

∀sdi ∈ <signal definition>:

sdi.s-implicit ∈ <identifier> ⇒ // sort identifier

sdi.s-implicit.refersto0.derivedDataType0.s-<data type definition body>.s-<data type constructor>

∈ <structure definition>

The <sort<identifier> of **struct** <sort<identifier> of a <signal definition> shall identify the sort of a structure data type. See clause 10.3 *Semantics* of [ITU‑T Z.104].

Transformations

<signal definition list>( < item > ⁀ rest) provided rest ≠ undefined

=5=> <signal definition list>( < item >) ⁀ <signal definition list>( rest)

If several <signal definition> items are specified in one <signal definition list>, this is equivalent to individual <signal definition list>s for each of them. See clause 10.3 *Concrete grammar* of [ITU‑T Z.101].

sigId.asSignal

provided

sigId.asSignal = undefined ∧

sigId ∈ {id ∈ <identifier>: id.refersto0 ∈ <signal definition>}

=5=>

case sortlist = sigId.refersto0.s-implicit of // sortlist

| undefined then // parameterless signal

(asSignal\{( sigId, undefined )} ) ∪ {( sigId, mk-<identifier>( sigId.s-<qualifier>, newName))}

| (<sort list> ∪ <named fields sort list>) then

(asSignal\{( sigId, undefined )} ) ∪ {( sigId, mk-<identifier>( sigId.s-<qualifier>, newName)}

| <identifier> then // sortlist = sort identifier

(asSignal\{( sigId, undefined )} ) ∪ {( sigId, sortlist }

endcase // sortlist

The following three paragraphs from clause 10.3 *Concrete grammar* of [ITU‑T Z.104], describes the implicit type associated (as above) with a <signal definition> by the function *asSignal* and mapped to the abstract syntax as described below.

A <signal definition> that defines a *Signal-definition* with an empty *Signal-parameter* list (a signal definition> without a <sort list> or <named fields sort list> or **struct** <sort identifier>), defines in the same context as the *Signal-definition* a *Syntype-definition* with a unique anonymous *Syntype- name*, NULL, as the *Parent-sort-identifier*, and empty *Range-condition* and no *Default-initialization*. In this case, an <as signal> (for the signal definition) when used as a <basic sort> denotes the data type NULL.

A <signal definition> that defines a *Signal-definition* with a non-empty *Signal-parameter* list defined by a <sort list> or <named fields sort list>, defines in the same context as the *Signal-definition* a *Data-type-definition* for a structure data type with a unique anonymous *Sort* name. In this case, an <as signal> (for the signal definition) when used as a <basic sort> denotes the *Sort* of the *Data-type-definition*. The *Data-type-definition* is equivalent to defining a structure data type with an **optional** <field> for each <aggregation kind> and <sort> item (in order) of the <sort list> or <named fields sort list>, where the <aggregation kind> and <field sort> is the same as those of the <sort list> or <named fields sort list>. If the <signal definition> has a <sort list>, each field has a unique anonymous name and therefore has to be identified using a <field number> and the field present operation is not accessible because its name is unknown. If the <signal definition> has a <named fields sort list>, each <field> has the name given by the <field name> of the <named fields sort list> item.

For a <signal definition> that defines a *Signal-definition* with a non-empty *Signal-parameter* list defined by **struct** <sort identifier>, an <as signal> (for the signal definition) when used as a <basic sort> denotes the same *Sort* as the <sort identifier> of **struct** <sort identifier>.

Mapping to abstract syntax

| <signal definition list>(< item >) then Mapping(item)

| sd = <signal definition>(\*,name,\*,\*,\*,sortlist) then

// type preamble virtuality/virtuality constraint ignored – type preamble abstract derived from sd

// specialization derived from sd: formal params not used here

<

if sortlist ∈ <identifier>

then // sortlist is a structure identifier

mk-Signal-definition(Mapping(name),

let structfields =

derivedDataType0(sortlist.refersto0

).s-<data type definition body>.s-<data type constructor>.s-<field list>

in

< mk-Signal-parameter(

Mapping(structfields[i].s-implicit.s-<fields of sort>.s-<field of kind>.head.s-<aggregation kind>),

Mapping(structfields[i].s-implicit.s-<fields of sort>.s-<field sort>)

) // Signal-parameter

: i ∈ 1..structfields.length

> // list of Signal-parameters

endlet // structfields

Mapping(sd.specialization0.s-<type expression>.baseType0),

if isAbstractType0(sd) then Abstract else Undefined endif

) // Signal-definition

else // sortlist is not a structure identifier – and there is a Signal-definition and a data type definition

mk-Signal-definition(Mapping(name),

if sortlist= undefined

then empty

else // sortlist is a sort list or a named sort list – sort identifier handled above

< mk-Signal-parameter(

Mapping(sortlist[i].s-<aggregation kind>),

Mapping(sortlist[i].s-<sort>))

: i ∈ 1..sortlist.length > // list of Signal-parameters

endif,

Mapping(sd.specialization0.s-<type expression>.baseType0),

if isAbstractType0(sd) then Abstract else Undefined endif

), // Signal definition followed by data type definition

case sortlist // undefined, sort list or named fields sort list

| undefined then // parameterless signal

mk-Syntype-definition(sd.identifier0.asSignal, // Syntype name

predefinedId0("NULL"), // NULL as the *Parent-sort-identifier*,

mk-Range-condition(∅), // empty *Range-condition*

undefined // default initialization

) // Syntype definition

| (<sort list> ∪ <named fields sort list>) then

let dtd = mk-<data type definition>(

mk-<type preamble>(undefined),

mk-<data type heading>(sd.identifier0.asSignal, undefined, undefined),

mk-<data type definition body>( empty, // entity in data type list

mk-<data type constructor>(

mk-<structure definition>( undefined, // visibility of structure

< mk-<optional field>(mk-<fields of sort>(undefined, // visibility of field

< mk-<field of kind>( sortlist[i].s-<aggregation kind>,

if sortlist ∈ <sort list> then newName else sortlist[i].s-<name> endif)

> ,

sortlist[i].s-<sort>)):// fields of sort, optional field

i ∈ 1..sortlist.length

> // list of fields

) // structure definition

), // data type constructor

mk-<operations>(

mk-<operation signatures>( undefined, undefined), mk-<operation definitions>( empty )

), // operations

undefined // default initialization

) // data type definition body

) // data type definition

in

Mapping(dtd)

endcase // sortlist

>

Auxiliary functions

The function asSignal is used to store the name of the (possibly anonymous) data type or syntype corresponding to a signal definition.

controlled asSignal: <identifier> → <identifier> // signal identifier to sort identifier

initially ∀ sigId ∈ {id ∈ <identifier>: id.refersto0 ∈ <signal definition>}: sigId.asSignal = undefined

#### F2.2.7.4 Signal list definition

Concrete syntax

<signal list definition> ::

<interface<name> <signal list>

<signal list> =

<signal list item>+

<signal list item> ::

<signal<identifier>

| <timer<identifier>

| <interface<identifier>

| <remote procedure<identifier>

| <remote variable<identifier>

Transformations

<signal list definition>(n,sl)

=8=>

<interface definition>(empty, undefined,

<interface heading>(n, empty, undefined), undefined, <entity in interface>(<interface use list>(sl)))

A <signal list definition> is an alternative concrete syntax, and is transformed into an <interface definition> using the keyword **interface** with no <package use clause>, no <virtuality>, an <interface heading> containing the <interface name> (but no <formal context parameters> or <virtuality constraint>), no <interface type expression> specialization and no <entity in interface>. The <interface definition> has an <interface use list> with a <signal list> that is the same as the <signal list> of the <signal list definition>. See clause 12.1.2 *Model* of [ITU‑T Z.103].

Mapping to abstract syntax

| < <signal list item>( id) > then Mapping(id)

#### F2.2.7.5 Remote procedures

Concrete syntax

<remote procedure definition> ::

<remote procedure<name> <procedure signature>

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

<remote procedure call body> ::

<remote procedure<identifier> [ <actual parameters> ] <communication constraints>

<timer communication constraint> ::

<timer<identifier> [ <variable> ]\* [ <connector<name> ]

Conditions on concrete syntax

See clause 10.5 *Concrete grammar* [ITU‑T Z.102] unless another reference is given.

∀pd∈<procedure definition>:

let rpi = pd.s-<procedure heading>.s-<procedure preamble>.s-<exported>.s-<identifier> in

let rpd =getEntityDefinition0(rpi, remote procedure) in

pd.isExported0 ∧ rpi≠ undefined⇒

isSameProcedureAndSignature0(pd, rpd.s-<procedure signature>)

endlet

The <remote procedure identifier> following **as** in an exported procedure definition shall denote a <remote procedure definition> with the same signature as the exported procedure. See clause 9.4 *Concrete grammar* [ITU‑T Z.102].

∀pd∈<procedure definition>:

let rpi = pd.s-<procedure heading>.s-<procedure preamble>.s-<exported>.s-<identifier> in

let rpd = resolveByContainer0(pd.surroundingScopeUnit0,

mk-<identifier>(empty, pd.name0),

remote procedure)

in

pd.isExported0 ∧ rpi = undefined⇒

(rpd ≠ undefined∧isSameProcedureAndSignature0(pd, rpd.s-<procedure signature>))

endlet

endlet

In an exported procedure definition with no **as** clause, there shall be a <remote procedure definition> in a surrounding scope with the same name and signature as the exported procedure and the nearest such <remote procedure definition> is used. See clause 9.4 *Concrete grammar* [ITU‑T Z.102].

∀rpc∈<remote procedure call>:

let d = parentAS0ofKind(rpc, <agent definition> ∪<agent type definition> ) in

rpc.s-<identifier>∈d.validOutputSignalSet0

endlet

A remote procedure mentioned in a <remote procedure call body> shall be in the complete output set of an enclosing agent type or agent set.

∀rpcb1, rpcb2 ∈ <remote procedure call body>:

let action1 = parentAS0ofKind(rpcb1, <action>) in

let action2 = parentAS0ofKind(rpcb2, <action>) in

parentAS0(rpcb1) ∉{<remote procedure call> ∪ <call statement>} // is <expression0>

∧ parentAS0(rpcb2) ∉{<remote procedure call> ∪ <call statement>} // is <expression0>

∧ rpcb1.s-<identifier> = rpcb2.s-<identifier> ⇒

action1 ≠ action2

∨ ¬isSameNode0(action1, action2)

∨ isSameNode0(rpcb1, rpcb2)

endlet

endlet

If a remote procedure is a value returning procedure, each action shall contain no more than one <remote procedure call body> used as an <expression0> for the same remote.

∀(cc∈<communication constraints>: ( |{d∈<destination>: d  in cc}| > 1)) ⇒

cc.parentAS0∉<remote procedure call body>∪<import expression>)

In Basic SDL-2010, a <communication constraints> shall contain at most one to<destination> clause. See clause 11.13.4 *Concrete grammar* of [ITU‑T Z.101].

Comprehensive SDL-2010 <communication constraints> of an <output body> of an <output area> is extended to allow more than one to<destination> clause. See clause 11.13.4 *Concrete grammar* of [ITU‑T Z.103].

NOTE – The rule in Z.101 is only extended in Z.103 for <output body>, therefore a <communication constraints> in a <remote procedure call body> or <import expression> shall contain no more than one <destination>.

∀tc∈<timer communication constraint>: tc.parentAS0.parentAS0∉<output body>

The <communication constraints> in an <output body> shall not contain a <timer communication constraint>. See clause 11.13.4 *Concrete grammar* of [ITU‑T Z.102].

∀(tc∈<timer communication constraint>: (|{tc  in tc.parentAS0}| ≤ 1)) ⇒

tc.parentAS0.parentAS0∉ (<remote procedure call body>∪<import expression>):

A <communication constraints> in a <remote procedure call body> shall contain no more than one <timer communication constraint>. A <communication constraints> in an <import expression> shall contain no more than one <timer communication constraint>. See clause 11.13.4 *Concrete grammar* of [ITU‑T Z.102].

∀v in <timer communication constraint>.**s**-<variable>-seq: ¬(isGlobalBlockVar0(v))

A <variable> of a <timer communication constraint> shall not be a global variable of a system (type) or block (type) except if the <timer communication constraint> is within the state machine actions of system (type) or block (type). See clause 10.5 *Model* of [ITU‑T Z.102].

Transformations

A remote procedure call body

– Proc(apar) to destination timer timerinfo via viapath

is modelled by an exchange of implicitly defined signals. If the to or via clauses are omitted from the remote procedure call, they are also omitted in the following transformations. The communication uses the channels where the remote procedure has been mentioned in the <signal list> (the outgoing for the importer and the incoming for the exporter) of at least one gate or channel connected to the importer or exporter. 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 (inparameters are excluded), and then interprets the transition for the remote procedure stimulus in the current state.

rpd.implicitName

provided

rpd ∈ <remote procedure definition>

∧ rpd.implicitName = undefined

=17=>

( implicitName\{( rpd, undefined )} ) ∪ {( rpd, newName)}

rpd=<remote procedure definition>(\*, sign)

provided

rpd.implicitName ≠ undefined

=17=>

let p = rpd.implicitName.s-Token

let sortlist = sign.s-<formal parameter>-seq in

< rpd, mk-<signal definition list>(undefined,

< mk-<signal definition>(

undefined, // type preamble

mk-<name >(p+"CALL"), // signal name

empty, // formal context parameters

undefined, // specialization

undefined, // virtuality constraint

< mk-<sort list>(

< sortlist[n].s-<sort> | n in 1..sortlist.length : sortlist[n].s-implicit ≠ **out** > ⁀

< predefinedId0("Integer") > )), // end mk sort list, end mk signal definition

< mk-<signal definition>(

undefined, // type preamble

mk-<name >(p+"REPLY"),// signal name

empty, // formal context parameters

undefined, // specialization

undefined, // virtuality constraint

< mk-<sort list>(

< sortlist[n].s-<sort> | n in 1..sortlist.length :

(sortlist[n].s-implicit ≠ **in** ∧ sortlist[n].s-implicit ≠ undefined)> ⁀

if sign.s-<result> ≠ undefined then < sign.s-<result> >

else empty

endif ⁀

< predefinedId0("Integer") >)) // end mk sort list, end mk signal definition

>) > // end mk signal definition list, end of replacement list

endlet // sortlist

endlet // p

There are two anonymously named implicit <signal definition list> items for each <remote procedure definition> in a system definition. The <signal name> items in these <signal definition> items are denoted by pCALL and pREPLY respectively, where p is uniquely determined. The signals are defined in the same scope unit as the <remote procedure definition>. Both pCALL and pREPLY have a last parameter of the predefined Integer sort.

< <channel definition>(n, er, delay,

<channel path>(ep1, ep2,

sigs1 ⁀ <signal list item>( i=<identifier>(q, \*)) ⁀ sigs2),

path2) >

provided stimulusKind(i) = **remote** **procedure** ∧ i.refersto0.implicitName ≠ undefined

=17=>

let p = i.refersto0.implicitName.s-Token in

<channel definition>(n, er, delay,

<channel path>(ep1, ep2,

sigs1⁀ < mk-<signal list item>(<identifier>(q, mk-<name>(p + "CALL"))) > ⁀ sigs2),

<channel path>(ep2, ep1,

if path2 = undefined then empty else path2.s-<signal list item>-seq endif ⁀

< mk-<signal list item>(<identifier>(q, mk-<name>(p + "REPLY")) > )))

endlet

On each channel mentioning the remote procedure, the remote procedure is replaced by pCALL. For each such channel, if it is unidirectional the channel is made bidirectional. In the opposite direction this channel carries the signal pREPLY. The new channel has the same delaying property as the original one.

The transform below replaces an exported procedure definition without an **as** clause by an exported procedure definition with an **as** clause that identifies the remote procedure with the same name as the exported procedure definition.

ipd = <internal procedure definition>

provided

ipd.s-<procedure heading>.s-<procedure preamble>.s-<exported> ≠ undefined

∧ ipd.s-<procedure heading>.s-<procedure preamble>.s-<exported>.s-<identifier> = undefined

// exported procedure without an **as** clause – change to identify remote procedure

=17=>

mk-<internal procedure definition>(

ipd.s-<package use clause>-seq,

mk-<procedure heading>(

mk-<procedure preamble>(

ipd.s-<procedure heading>.s-<procedure preamble>.s-implicit // virt/abstract,

mk-<exported>(

resolveByContainer0(ipd.surroundingScopeUnit0,

mk-<identifier>(empty, ipd.s-<procedure heading>.s-<name>),

remote procedure).identifier0 // id for rpd

) // end mk exported

), // end of procedure preamble

ipd.s-<procedure heading>.s-<qualifier>,

ipd.s-<procedure heading>.s-<name>,

ipd.s-<procedure heading>.s-<formal context parameters>,

ipd.s-<procedure heading>.s-<virtuality constraint>,

ipd.s-<procedure heading>.s-<specialization>,

ipd.s-<procedure heading>.s-<procedure formal parameters>,

ipd.s-<procedure heading>.s-<procedure result>

), // end of procedure heading

ipd.s-<entity in procedure>-seq,

ipd.s-implicit // body, statements or compound statement

) // end mk internal procedure definition

The transform below inserts the implicit variable definitions needed for an exported procedure after an exported procedure definition that has an **as** clause that identifies the remote procedure definition.

< ipd = <internal procedure definition> >

provided

ipd.s-<procedure heading>.s-<procedure preamble>.s-<exported> ≠ undefined

∧ ipd.s-<procedure heading>.s-<procedure preamble>.s-<exported>.s-<identifier> ≠ undefined

// exported procedure without an **as** clause – change to identify remote procedure

∧ ¬(∃ v ∈ <variables of sort> :

v.s-<name> =

mk-<name>(ipd.s-<procedure heading>.s-<procedure preamble>

.s-<exported>.s-<identifier>.refersTo0.implicitName.s-Token+"n")

∧ (fullQualifier0(v) = fullQualifier0(ipd)))

// There is not already a variable with the name p+"n" in the scope ipd defined,

// where p is the unique name associated with remote procedure definition,

// therefore the variables need to be defined.

=17=>

let rpd = ipd.s-<procedure heading>.s-<procedure preamble>.s-<exported>.s-<identifier> in

// remote procedure definition id

let p = rpd.refersto0.implicitName.s-Token in

// p is the unique Token associated with remote procedure definition

< ipd > ⁀

< variableDefinition0(p + "n", predefinedId0("Integer"), undefined) > ⁀

< variableDefinition0(p + "ivar", predefinedId0("Pid"), undefined) > ⁀

if rpd.s-<procedure signature>.s-<result> = undefined then

empty

else

< variableDefinition0(p + "res", rpd.s-<procedure signature>.s-<result>, undefined) >

endif ,

remoteProcParamsDef(ipd.s-<procedure heading>.s-<procedure formal parameters>, p)

> // end of replacement

endlet // p

endlet // rpd

a) Requesting agent

The remote procedure in a requesting agent has seven transforms. The first transform handles the variable definitions for implicit variables and is invoked only once. The subsequent six transforms insert actions before the action containing the remote procedure call body for the signal exchange. Depending on where the remote procedure call was, the pattern matches one of the six transformations for the enclosing body: an agent body with the remote procedure call in a start, an agent body with the remote procedure call in a state or free action, a composite state body with the remote procedure call in a start, a composite state body with the remote procedure call in a state or free action, a procedure body with the remote procedure call in a start, or a procedure body with the remote procedure call in a state or free action. These are invoked only if the implicit variables are defined. If the procedure is in an expression, the remote procedure call is replaced with an access to the implicit variable for the returned value - otherwise the remote procedure call is removed.

items= getEntities0(//entities defined in agent type definition where remote procedure is called

atd = parentAS0ofKind( // finds agent type definition with import expression

rpcb = <remote procedure call body>,

<system type definition> ∪ <block type definition> ∪ <process type definition>))

)) //end parentAS0ofKind, end of getEntities0

provided

¬ callerVariablesDefined(rpcb)

=17=>

< variableDefinition0(rpcb.s-<identifier>.refersto0.implicitName.s-Token + "n",

predefinedId0("Integer"),

mk-<operand5>( mk-<identifier>( predefinedQual0("Integer"), mk-<name>("1")))) >

⁀

< variableDefinition0( rpcb.s-<identifier>.refersto0.implicitName.s-Token + "newn",

predefinedId0("Integer"), undefined) >

⁀ if rpd.s-<procedure signature>.s-<result> = undefined then empty

else

< variableDefinition0( rpcb.s-<identifier>.refersto0.implicitName.s-Token + "res",

rpd.s-<procedure signature>.s-<result>,undefined) >

endif // rpcb is in an expression

⁀ items // variables are in scope and can be used in items

– For each imported procedure, two implicit anonymous Integer variables (in this description called n and newn) are defined in the enclosing scope unit of the <remote procedure call body>, and n is initialized to 0.

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

– If remote procedure is a value returning procedure, an implicit anonymous variable (in this description called res) is defined in the enclosing scope unit of the <remote procedure call body> with the sort returned by the procedure.

body = <agent body>( // agent body with remote procedure call in start

start = <start>(

virt,

entry,

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(

rpcb = <remote procedure call body>,

<action>), // parentAS0ofKind for action

<action>-seq), // parentAS0ofKind for actionItems,

tr) // tr is the terminator for <transition action items>

), // <start>

bodyitems

) // agent body with remote procedure call in start

provided

callerVariablesDefined(rpcb)

=17=>

let actions1 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ n < i > in

let actions2 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ i < n > in

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

// oldLabel – where to go for joins previously to action

let newLabel =

if rpcb.parentAS0 ∉ <remote procedure call> ∨ actions2.head.s-<label> = undefined

then if actions2≠ empty then newName else undefined endif

else actions2.head.s-<label> endif

in // newLabel – where to go after the replacement for action

mk-<agent body>(

mk-<start>(virt, *entry*, newActions1AS0(actions1, oldlabel)),

transformItems0(oldlabel, newlabel, rpcb)⁀ newActions2AS0(actions2, newlabel, rpcb, tr)⁀ bodyitems

) // end agent body with remote procedure call in start

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

body = <agent body>( // agent body with remote procedure call in state or free action

start,

bodyitems = parentAS0ofKind(

bodyitem = parentAS0ofKind(

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(

rpcb = <remote procedure call body>,

<action>), // parentAS0ofKind for action

<action>-seq), // parentAS0ofKind for actionItems,

tr), // tr is the terminator for <transition action items>

(<state> ∪ <free action>)) // parentAS0ofKind for bodyitem

(<state> ∪ <free action>)-seq) // parentAS0ofKind for bodyitems

) // agent body with remote procedure call in state or free action

provided

callerVariablesDefined(rpcb)

=17=>

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

// oldLabel – where to go for joins previously to action

let newLabel =

if rpcb.parentAS0 ∉ <remote procedure call> ∨ actions2.head.s-<label> = undefined

then if actions2≠ empty then newName else undefined endif

else actions2.head.s-<label> endif

in // newLabel – where to go after the replacement for action

mk-<agent body>(

start,

bilistpre0(bodyitems, bodyitem)⁀

if body ∈ <state>

then < revisedState0(bodyitem, action, newActions1AS0(actions1, oldlabel)) >

else < mk-<free action>(newactions1AS0(actions1, oldlabel)) >

endif⁀

transformItems0(oldlabel, newlabel, rpcb)⁀

newActions2AS0(actions2, newlabel, rpcb, tr)⁀

bilistpost0(bodyitems,bodyitem)

) // agent body with remote procedure call in state or free action

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

body = <composite state body>( // composite state body with remote procedure call in start

startlist1⁀

< start = <start>(

virt,

entry,

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(

rpcb = <remote procedure call body>,

<action>), // parentAS0ofKind for action

<action>-seq), // parentAS0ofKind for actionItems,

tr) // tr is the terminator for <transition action items>

) > // start list item containing remote procedure call

⁀startlist2,

bodyitems

) // composite state body with remote procedure call in start

provided

callerVariablesDefined(rpcb)

=17=>

let actions1 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ n < i > in

let actions2 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ i < n > in

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

// oldLabel – where to go for joins previously to action

let newLabel =

if rpcb.parentAS0 ∉ <remote procedure call> ∨ actions2.head.s-<label> = undefined

then if actions2≠ empty then newName else undefined endif

else actions2.head.s-<label> endif

in // newLabel – where to go after the replacement for action

<composite state body>(

startlist1⁀

< mk-<start>(virt, *entry*, newActions1AS0(actions1, oldlabel)) >⁀ // revised start as list item

startlist2,

transformItems0(oldlabel, newlabel, rpcb)⁀ newActions2AS0(actions2, newlabel, rpcb, tr)⁀ bodyitems

) // composite state body with remote procedure call in start

body = <composite state body>( // agent body with remote procedure call in state or free action

startlist,

bodyitems = parentAS0ofKind(

bodyitem = parentAS0ofKind(

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(

rpcb = <remote procedure call body>,

<action>), // parentAS0ofKind for action

<action>-seq), // parentAS0ofKind for actionItems,

tr), // tr is the terminator for <transition action items>

(<state> ∪ <free action>)) // parentAS0ofKind for bodyitem

(<state> ∪ <free action>)-seq) // parentAS0ofKind for bodyitems

) // composite state body with remote procedure call in state or free action

provided

callerVariablesDefined(rpcb)

=17=>

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

// oldLabel – where to go for joins previously to action

let newLabel =

if rpcb.parentAS0 ∉ <remote procedure call> ∨ actions2.head.s-<label> = undefined

then if actions2≠ empty then newName else undefined endif

else actions2.head.s-<label> endif

in // newLabel – where to go after the replacement for action

mk-<composite state body>(

startlist,

bilistpre0(bodyitems, bodyitem)⁀

if body ∈ <state>

then < revisedState0(bodyitem, action, newActions1AS0(actions1, oldlabel)) >

else < mk-<free action>(newActions1AS0(actions1, oldlabel)) >

endif⁀

transformItems0(oldlabel, newlabel, rpcb)⁀

newActions2AS0(actions2, newlabel, rpcb, tr)⁀

bilistpost0(bodyitems,bodyitem)

) // composite state body with remote procedure call in state or free action

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

body = <procedure body>( // agent body with remote procedure call in start

start = <start>(

virt,

entry,

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(

rpcb = <remote procedure call body>,

<action>), // parentAS0ofKind for action

<action>-seq), // parentAS0ofKind for actionItems,

tr) // tr is the terminator for <transition action items>

), // <start>

bodyitems

) // procedure body with remote procedure call in start

provided

callerVariablesDefined(rpcb)

=17=>

let actions1 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ n < i > in

let actions2 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ i < n > in

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

// oldLabel – where to go for joins previously to action

let newLabel =

if rpcb.parentAS0 ∉ <remote procedure call> ∨ actions2.head.s-<label> = undefined

then if actions2≠ empty then newName else undefined endif

else actions2.head.s-<label> endif

in // newLabel – where to go after the replacement for action

mk-<procedure body>(

mk-<start>(virt, *entry*, newActions1AS0(actions1, oldlabel)),

transformItems0(oldlabel, newlabel, rpcb)⁀ newActions2AS0(actions2, newlabel, rpcb, tr)⁀ bodyitems

) // end procedure body with remote procedure call in start

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

body = <procedure body>( // procedure body with remote procedure call in state or free action

start,

bodyitems = parentAS0ofKind(

bodyitem = parentAS0ofKind(

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(

rpcb = <remote procedure call body>,

<action>), // parentAS0ofKind for action

<action>-seq), // parentAS0ofKind for actionItems,

tr), // tr is the terminator for <transition action items>

(<state> ∪ <free action>)) // parentAS0ofKind for bodyitem

(<state> ∪ <free action>)-seq) // parentAS0ofKind for bodyitems

) // procedure body with remote procedure call in state or free action

provided

callerVariablesDefined(rpcb)

=17=>

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

// oldLabel – where to go for joins previously to action

let newLabel =

if rpcb.parentAS0 ∉ <remote procedure call> ∨ actions2.head.s-<label> = undefined

then if actions2≠ empty then newName else undefined endif

else actions2.head.s-<label> endif

in // newLabel – where to go after the replacement for action

mk-<procedure body>(

start,

bilistpre0(bodyitems, bodyitem)⁀

if body ∈ <state>

then < revisedState0(bodyitem, action, newActions1AS0(actions1, oldlabel)) >

else < mk-<free action>(newActions1AS0(actions1, oldlabel)) >

endif⁀

transformItems0(oldlabel, newlabel, rpcb)⁀

newActions2AS0(actions2, newlabel, rpcb, tr)⁀

bilistpost0(bodyitems,bodyitem)

) // procedure body with remote procedure call in state or free action

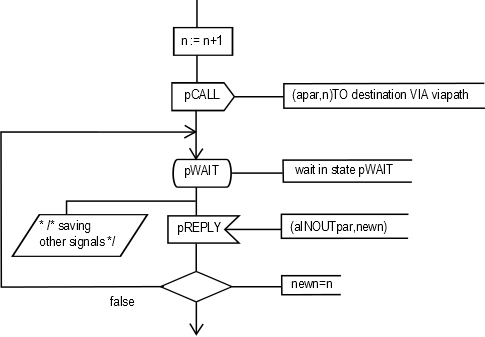
endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

– The <remote procedure call body> is transformed as below, so that the following is inserted before the action that contained the <remote procedure call body>, where in the output 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:

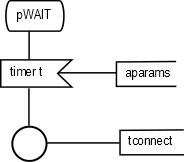


– where apar is the list of actual parameters except actual parameters corresponding to **out**-parameters, and aINOUTpar is the modified list of actual **in/out**-parameters and **out**‑parameters, including the implicit variable res as an additional parameter if a value returning remote procedure call is transformed.

– The transform is labelled with the label on the action containing the remote procedure call or a new label if this action is not labelled, and the preceding path is changed to join this label.

– If a value returning remote procedure call is transformed, the true path above is terminated with a join to the action that contained the remote procedure call with a new label, and the remote procedure call is replaced by an access of the implicit variable res used to receive the returned value. Otherwise the remote procedure call action is removed and the true path above is joined to the action following the remote procedure call action.

– Additionally, the following will be inserted if a <timer communication constraint> is included in <communication constraints>



where

– t is the <timer identifier> in the <timer communication constraint>;

– aparams is the optional list of optional <variable> items given after the <timer identifier> in the <timer communication constraint>;

– tconnect is the <connector name> if one is given in the <timer communication constraint>; otherwise tconnect is the name of the timer.

– In all other states, pREPLY is discarded. This is not explicitly modelled: instead the handling of pREPLY is left unspecified in the transformed concrete syntax except for the pWAIT states with the consequence that there is an implicit transition (see clause 11.8 of [ITU-T Z.103]) for other states that discards the signal.

– If the <remote procedure call body> was directly enclosed by a <remote procedure call>, it is an <action> that is the <remote procedure call> and the transform replaces the <remote procedure call body>. Otherwise the <remote procedure call body> is a <value returning procedure call> as an <expression0>, and transform is inserted before the action that contained the <value returning procedure call>, and the <value returning procedure call> is replaced in this action by an access of the implicit variable res used to receive the returned value.

b) Server agent

The ASM for transformations for server agents is given in 3 parts: <input part> where the remote procedure is given explicitly; any <save> where a remote procedure is given explicitly; states in agents that can receive the remote procedure call where the remote procedure is not given explicitly either in an <input part> or a <save part>.

ip = <input part>

provided

ip.s-<input list>.s-<asterisk input list> = undefined // not asterisk input list

∧ ip.s-<input list>.s-<encoded input > = undefined // not encoded input

∧ ip.s-<input list>.s-implicit-seq.length = 1 // assume all input lists one stimulus/list

∧ ip.s-<input list>.s-implicit-seq[1].s-<stimulus>.s-<signal list item>.refersto0

∈ <remote procedure definition> // the stimulus refers to a remote procedure definition

∧ ip.s-<input list>.s-implicit-seq[1].s-<signal list item>.refersto0.implicitName ≠ undefined

// the <remote procedure definition> transformed

=17=>

//replaces the <input part> for rpc

let rpc = ip.s-<input list>.s-implicit-seq[1].s-<stimulus>.s-<signal list item> in // remote proc call id

let rpd = rpc.refersto0 in // remote procedure definition

let fpl = rpd.s-<procedure signature>.s-<formal parameter>-**seq** in // formal params list

let p = rpd.implicitName.s-Token in // unique Token associated with remote procedure definition

let pCALL = mk-<identifier>( rpd.fullQualifier0, mk-<name>(p + "CALL")) in // identifier for pCALL

let pREPLY = mk-<identifier>( rpd.fullQualifier0, mk-<name>(p + "REPLY")) in // identifier for pREPLY

let exportedProc = take({proc ∈<internal procedure definition> :

isExported0(proc)

∧ proc.s-<procedure heading>.s-<procedure preamble>.s-<exported> = rpc

∧ parentAS0ofKind(proc, ScopeUnit0) =

parentAS0ofKind(ip,

<system type definition> ∪ <block type definition> ∪ <process type definition>)

})

in

let fpar = exportedProcParams(fpl, remoteProcParamsDef(fpl, p)) in

let sq = parentAS0ofKind(exportedProc, ScopeUnit0).fullQualifierWithin0 in // qualifier for vars

let n = mk-<identifier>( sq, mk-<name>( p + "n")) in // var for the integer to be returned in reply

let ivar = mk-<identifier>( sq, mk-<name>(p + "ivar")) in // var for the sender Pid

let res = mk-<identifier>( sq, mk-<name>(p + "res")) in // var for the result for a value returning procedure

<input part>(

ip.s-<virtuality>,

< // input list

mk-<stimulus>(

mk-<signal list item>( pCALL),

remoteProcParams(fpl, remoteProcParamsDef(fpl, p)) ⁀ < n >,

ip.s-<input list>.s-implicit-seq[1].s-<stimulus>.s-<via path>), // via path, end of stimulus

ip.s-<input list>.s-implicit-seq[1].s-<in choice>, // in choice

ip.s-<input list>.s-implicit-seq[1].s-<priority clause> // priority

>, // end of input list

ip.s-<enabling condition list>, // enabling condition

mk-<transition action items>( < // <action> list

assignmentTaskAction0(ivar, mk-<operand5>(undefined, mk-<sender expression>())) ⁀

if rpd.s-<procedure signature>.s-<result> = undefined then

mk-<action>(

undefined, // omitted <label> of <action>

mk-<procedure call>( mk-<procedure call body>(undefined, exportedProc, fpar)))

else

assignmentTaskAction0(res,

mk-<value returning procedure call>(

mk-<procedure call body>(undefined, exportedProc, fpar)))

endif ⁀

mk-<output>( pREPLY, aINOUTremoteProcParams(rpc)⁀< n >, ivar) ⁀

ip.s-<transition>.s-<action>-seq >, // end of <action> list

ip.s-<transition>.s-<terminator> ) // <terminator> of <transition action items>

) // end of <input part> replacement

endlet // res

endlet // ivar

endlet // n

endlet // sq

endlet // fpar

endlet // exportedProc

endlet // pREPLY

endlet // pCALL

endlet // p

endlet // fpl

endlet // rpd

endlet // rpc

save = <save part> ( // that contains a save using rpc

virt, // virtuality of <save part>

< // Save list

rpc = take({id ∈<identifier> : id.refersto0 ∈ <remote procedure definition>}),

vp // via path, end of stimulus

>, // End of Save list

),// end of <save part>

provided

rpc.refersto0.implicitName ≠ undefined // the <remote procedure definition> transformed

∧ save ≠ undefined // there is a <save part> as defined by save

=17=>

mk-<save part>(virt, < rpc.refersto0.implicitName.s-Token + "CALL", vp >)

s = <state>

provided

parentAS0ofKind(s,<system type definition> ∪ <block type definition> ∪ <process type definition>) = def

// def that channel endpoint with remote procedure call is connected to

∧ rpd ∈ <remote procedure definition>

∧ rpd.implicitName ≠ undefined // the <remote procedure definition> transformed

∧ def ∈{cp.s2-<channel endpoint>.refersto0.s-<type expression>.s-<base type>.refersto0:

cp ∈ <channel path> ∧ sli in cp.s-<signal list>

∧ ( (sli.s-<signal list item> =

<identifier>( rpd.fullQualifier0, <name>(rpd.implicitName.s-Token + "CALL")))

∨ (sli.s-<signal list item> = rpd.fullIdentifier0)) // signal list item pCALL or remote proc

∧ cp.s2-<channel endpoint>.s- implicit ∈ <identifier> // to <channel endpoint>

} // set for def

∧ ¬handled(rpd.fullIdentifier0, s.s-<state list>.head.name0)) // rpc is not handled in s.

=17=>

let rpc = rpd.fullIdentifier0 in // remote procedure id

let p = rpd.implicitName in // unique name associated with remote procedure definition

let pCALL = mk-<identifier>( rpd.fullQualifier0, mk-<name>(p + "CALL")) in // identifier for pCALL

let pREPLY = mk-<identifier>( rpd.fullQualifier0, mk-<name>(p + "REPLY")) in // identifier for pREPLY

let sq = def.fullQualifierWithin0 in // qualifier for items defined in def – the scope for vars

let n = mk-<identifier>( sq, mk-<name>(p + "n")) in // var for the integer to be returned in reply

let ivar = mk-<identifier>( sq, mk-<name>(p + "ivar")) in // var for the sender Pid

let res = mk-<identifier>( sq, mk-<name>(p + "res")) in // var for the result for a value returning procedure

let fpl = rpd.s-<procedure signature>.s-<formal parameter>-**seq** in

let remoteProcParamVars = remoteProcParamsDef(fpl, p) in

let exportedProc = take({proc ∈<internal procedure definition> :

isExported0(proc)

∧ proc.s-<procedure heading>.s-<procedure preamble>.s-<exported> = rpd.fullIdentifier0

∧ parentAS0ofKind(proc, ScopeUnit0) = def })

in

let fpar = exportedProcParams(fpl, remoteProcParamVars) in

let actionItems =

assignmentTaskAction0(ivar, mk-<operand5>(undefined, mk-<sender expression>())) ⁀

if rpd.s-<procedure signature>.s-<result> = undefined then

mk-<action>(

undefined, // omitted <label> of <action>

mk-<procedure call>( mk-<procedure call body>(undefined, exportedProc, fpar)))

else

assignmentTaskAction0(res,

mk-<value returning procedure call>(

mk-<procedure call body>(undefined, exportedProc, fpar)))

endif ⁀

mk-<output>( pREPLY, aINOUTremoteProcParams(rpc)⁀< n >, ivar)

in

mk-<state>(s.s-<state list>, s.s-implicit ⁀

mk-<input part>(virt,

< // input list

mk-<stimulus>( pCALL,

remoteProcParams(fpl, remoteProcParamVars) ⁀ < n >,

vp ), // via path, end of stimulus

undefined, // in choice

undefined // priority

>, // end of input list

ec, // enabling condition

mk-<transition action items>( < // <action> list

actionItems ⁀

mk-<terminator>(undefined, mk-<dash nextstate>())

>) // end of <action> list

> // end of input list

) // ) for <input part>

) // ) for <state>

endlet // actionItems

endlet // fpar

endlet // exportedProc

endlet // remoteProcParamsVars

endlet // fpl

endlet // res

endlet // ivar

endlet // n

endlet // sq

endlet // pREPLY

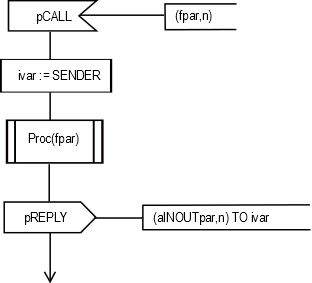
endlet // pCALL

endlet // p

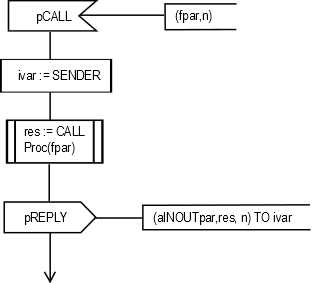
endlet // rpc

In the server agent, an implicit anonymous Integer variable (in this description called n) is defined for each <input> that is a remote-procedure input. Furthermore, there is an implicit anonymous Pid variable (in this description called ivar) for each such <input> defined in the scope where the remote procedure input occurs. If a value returning remote procedure call is transformed, an implicit anonymous variable (in this description called res) with the same sort as <sort> in <procedure result> is defined.

– To all <state> items with a remote procedure input transition, the following <input> replaces the remote procedure input and leads to the transition for the remote procedure:



– or,



– if a value returning remote procedure call was transformed.

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



– To all other <state> items (excluding implicit states derived from input with pCALL) where the remote procedure is not shown, the <input> described above is added and terminates in a next state that returns to the same state.

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

Auxiliary functions

The function implicitName is used to store the implicitly generated name for a remote entity definition.

controlled implicitName: <remote procedure definition> ∪ <remote variable definition> → <name>

initially ∀ r ∈ <remote procedure definition> ∪ <remote variable definition>: r.implicitName = undefined

The function callerVariablesDefined checks if AS0 has already been modified to include variable definitions in the agent type surrounding the remote procedure call or remote variable import.

callerVariablesDefined(rem: <remote procedure call body> ∪ <import expression>): Boolean =def

rem.s-<identifier>.refersto0.implicitName ≠ undefined

∧ (∃ v ∈ <variables of sort> :

v.s-<name>.s-Token =

(rem.s-<identifier>.refersto0.implicitName.s-Token + "n")

∧ (fullQualifier0(v) = fullQualifierWithin0( parentAS0ofKind( rem,

<system type definition> ∪ <block type definition> ∪ <process type definition>))))

// There is already already a variable with the name z +"n" in the type definition surrounding rem

// where z is the Token) for the unique name associated with the remote definition

// (that is, z = rem.s-<identifier>.refersto0.implicitName.s-Token).

The function variableDefinition0 makes an AS0 <variable definition> for a variable with a <name> that has the given Token value, the given <sort> and the given <operand5> constant expression, which should be undefined if the variable is not initialized.

variableDefinition0(token: Token, sort: <sort>, const: <operand5>): <variable definition> =def

mk-<variable definition>( < mk-<variables of sort>( PART, < mk-<name>(token) >, sort, const ) > )

The function newActions1AS0 makes an AS0 <transition action items> or <terminator> to replace the actions before the action containing the remote procedure call.

newActions1AS0(actions1: <action>\*, oldlabel: <name>): <transition action items> ∪ <terminator> =def

if actions1= empty

then mk-<terminator>( mk-<join>(oldLabel))

else mk-<transition action items>(actions1, mk-<terminator>( mk-<join>(oldLabel)))

endif

The function transformItems0 is used to define a list with a <free action> and a <state> to insert before the action that had the remote procedure call body in the body of an agent, state or procedure.

transformItems0(oldlabel: <name>, newlabel: <name>, rpcb: <remote procedure call body>):

(<state>∪<free action>)\* =def

let sq = fullQualifierWithin0( parentAS0ofKind( rpcb,

<system type definition> ∪ <block type definition> ∪ <process type definition>)) in

let rpd = rpcb.s-<identifier>.refersto0 in

let nid = mk-<identifier>(sq, mk-<name>( rpcb.s-<identifier>.refersto0.implicitName.s-Token + "n")) in

let n = mk-<operand5>(<variable access>(nid) in

let resid = mk-<identifier>(sq, mk-<name>( rpcb.s-<identifier>.refersto0.implicitName.s-Token + "res")) in

let p = rpd.implicitName.Token in

let pCALL = mk-<identifier>( rpcb.s-<identifier>.s-<qualifier>, mk-<name>(p + "CALL")) in

let pREPLY = mk-<identifier>( rpcb.s-<identifier>.s-<qualifier>, mk-<name>(p + "REPLY")) in

let pWAIT = mk-<name>(p + "WAIT") in

< mk-<free action>(

mk-<transition action items>(<

mk-<action>( oldLabel, mk-<task>(<assignment>( nid,

mk-<operator application>("+", < n, predefinedItem0("Integer","1") >)))),

mk-<action>(undefined,<output>(

mk-<output body>(

mk-<output body item>(pCALL,

apar(rpd.s-<procedure signature>.s-<formal parameter>-seq,

rpcb.s-<actual parameters>)⁀< n >),

commConstrNoTimer0(rpcb.s-<communication constraints>))),

mk-<terminator>(undefined, mk-<nextstate>( mk-<nextstate body name>(pWAIT)))

> ), undefined)

> ⁀

< mk-<state>(< pWAIT >,

timerInput0(rpcb.s-<communication constraints>.s-<timer communication constraint>) ⁀

< mk-<save part>(undefined, <asterisk save list>()),

mk-<input part>(undefined, //virtuality

< mk-<stimulus>(mk-<signal list item>(pREPLY),

aINOUTpar(rpd.s-<procedure signature>.s-<formal parameter>-seq,

rpcb.s-<actual parameter>-seq,

if rpd.s-<procedure signature>.s-<result> ≠ undefined

then mk-<operand5>( mk <variable access>(resid))

else undefined

endif) // aINOUTpar

⁀ < mk-<identifier>(sq, varNewN) >, undefined) // stimulus,

undefined, // in choice

undefined // priority clause

>,

undefined, // enabling condition

mk-<transition action items>(

mk-<decision>(<operand5>( mk-<operator application>(mk-<name>("="), < newn, n >)),

mk-<textual answer part>(predefinedItem0("Boolean","true"),

mk-<terminator>( mk-<join>(newLabel)))

mk-< textual answer part>(predefinedItem0("Boolean","false"),

mk-<terminator>(<nextstate>(pWAIT)))

) // transition

>, // state parts

undefined ) // state timer part

> // for other states nothing is specified for p + "REPLY" so it is treated as an implicit transition

endlet // pWAIT

endlet // pREPLY

endlet // pCALL

endlet // p

endlet // resid

endlet // n

endlet // nid

endlet // rpd

endlet // sq

The function newActions2AS0 makes an AS0 <transition action items> or <terminator> to replace the actions after the action containing the remote procedure call. If the remote procedure call is in a expression, the action containing the remote procedure call has the call changed to access the implicit variable to the result of the call. Otherwise, the action is the remote procedure call and is removed.

newActions2AS0(actions2: <action>\*, newlabel: <name>, rpcb: <remote procedure call body>,

tr: <terminator>): (<state>∪<free action>)\* =def

let sq = fullQualifierWithin0( parentAS0ofKind( rpcb,

<system type definition> ∪ <block type definition> ∪ <process type definition>)) in

< mk-<free action>(

mk-<transition action items>(

if rpcb.parentAS0 ∉ <remote procedure call>

then // <remote procedure call body> in <expression>

< mk-<action>(newLabel,

replaceOnceInSyntaxTree0(rpcb,

mk-<operand5>(<variable access>(<identifier>(

sq, mk-<name>( rpcb.s-<identifier>.refersto0.implicitName.s-Token + "res")))),

parentAS0ofKind(rpcb, <action>).s-implicit) // replace rpcb by res access in action

> ⁀ actions2

else // <remote procedure call body> in <remote procedure call> -

if actions2 ≠ empty

then < mk-<action>(newLabel, actions2.head.s-implicit) >⁀ actions2.tail

else // dummy action n := n+0 so action can be labelled!

< mk-<action>( newLabel, mk-<task>( mk-<assignment>(

mk-<identifier>(sq,rpcb.s-<identifier>.refersto0.implicitName.s-Token + "n"),

mk-<operand5>(<variable access>( <identifier>(sq,

mk-<name>(rpcb.s-<identifier>.refersto0.implicitName.s-Token+ "n")))),

))) // assignment, task, action

> // action list

endif,

endif,

tr) //transition action items

) > // free action, sequence

endlet // sq

The function bilistpre0 takes a body list of <state> and <free action> items, and produces a new list where the given item (<state> or <free action>) and all subsequent items are deleted, so that the new list can be used to replace the first part of the body list.

bilistpre0(bl:(<state>∪<free action>)\*, bsf:<state>∪<free action>):(<state>∪<free action>)\* =def

if bl.length = 1 ∨ bl.tail.head = bsf

then empty

else < bl.head > ⁀ bilistpre0(bl.tail,bsf)

endif

The function revisedState0 takes <state> item, and returns a new <state> item, where each part of the <state> (<input part>, <priority input>, <save>, <state timer input>) is the result of revisedStatePart0 that returns a modified part only for the part that contains the given action.

revisedState0(st(statenames, stateparts, statetimer): <state>, action: <action>,

newactions1: <transition action items> ∪ <terminator>):<state> =def

st( statenames,

< revisedStatePart0(stateparts[i], action, newactions1) | i in 1..stateparts.length >,

revisedStatePart0(statetimer, action, newactions1))

The function revisedStatePart0 takes state part, and returns the same state part if it does not contain (is not a parent of) the given action, otherwise it returns a revised part where the transition containing the action is replaced the given transition parameter.

NOTE – Assuming the original transition containing action is of the form actions1⁀action⁀actions2 then the replacement newactions1 is a version of actions1 that ends with a join to the new actions inserted before the old action. The replacement for action and the subsequent actions2 are placed in <state> and <free action> items in the body item list of the body. The order of these items does not really matter as they connected by state names and labels that also ensure that no change is needed to the rest of the revised state and any other state. If action was labelled, this label is moved to the start actions inserted before action, therefore any join to action becomes a join to this point.

revisedStatePart0(part: StatePart0, action: <action>,

newactions1: <transition action items> ∪ <terminator>): StatePart0 =def

let anspart = parentAS0ofKind(action, <textual answer part>) in

let elsepart = parentAS0ofKind(action, <textual else part>) in

if parentAS0ofKind(action, StatePart0) ≠ part then part

elseif parentAS0ofKind(anspart, StatePart0) = part

then

let d = anspart.parentAS0.parentAS0 in

replaceInSyntaxTree0(d,

mk-<decision>(d.s-implicit, // the question

mk-<textual decision body>(

revisedAnswers0(

d.s-<textual decision body>.**s**-<textual answer part>-seq,action,newactions1),

d.s-<textual decision body>.**s**-<textual else part>)),

part)

endlet

elseif parentAS0ofKind(elsepart, StatePart0) = part

let d = elsepart.parentAS0.parentAS0 in

replaceInSyntaxTree0(d,

mk-<decision>(d.s-implicit, // the question

mk-<textual decision body>(d.s-<textual decision body>.**s**-<textual answer part>-seq,

mk-<textual else part>(newactions1))),

part)

endlet

then

else

case part of

| <connect part>(v,\*,\*) then mk-<connect part>(v, part.s-implicit, newactions1)

| <continuous signal>(v,ce,pri,\*) then mk-<continuous signal>(v, ce, pri, newactions1)

| <input part>(v,\*,ec,\*) then mk-<input part>(v, part.s-implicit,ec, newactions1)

| <priority input>(v,\*,\*) then mk-<priority input>(v, part.s-implicit, newactions1)

| <save part> then part

| <spontaneous transition>(v,ec,\*) then mk-<spontaneous transition>(v,ec,newactions1)

| <state timer part>(v,\*,\*) then mk-<state timer part>(v, part.s-implicit, newactions1)

endcase

endif

endlet

endlet

The function revisedAnswers0 takes <textual answer part> list of a decision, and produces a new list where the answer that contains the given <action> is replaced by an answer with the newactions1 list of <actions> as the answer transition.

revisedAnswers0(answers:<textual answer part>\*, action: <action>,

newactions1: <transition action items> ∪ <terminator>):<textual answer part>\* =def

if answers = empty then empty

elseif answers.head.s-<transition action items> =

parentAS0ofKind(action, <transition action items>)

then < mk-<textual answer part>( answers.head.s-implicit, newactions1) > ⁀ answers.tail

else < answers.head > ⁀ revisedAnswers0(answers.tail,action,newactions1)

The function bilistpost0 takes a body list of <state> and <free action> items, and produces a new list where the given item (<state> or <free action>) and all preceding items are deleted, so that the new list can be used to replace the last part of the body list.

bilistpost0(bilist:{<state>∪<free action>}\*,bodyitem: <state>∪<free action>):{<state>∪<free action>}\* =def

if bilist.length = 1 then empty

elseif bilist.head = bodyitem then bilist.tail

else bilistpost0(bilist.tail,bodyitem) endif

The function handled checks whether a remote procedure call is explicitly handled within a state. The function returns true if (and only if) there exists a state s that is in the given states list, such that the given state name is one of the names in the <state list> of s, and for the given identifier rpc: either rpc (or its pCALL replacement) is one of the signal list item identifiers of one of the input list items of an <input part> of s, or rpc (or its pCALL replacement) is one of the save item identifiers of one of the save items of a <save part> of s.

handled(rpc: <identifier>, state: <name>): Boolean =def

let pCALL =

mk-<identifier>(rpc.s-<qualifier>, mk-<name>(rpc.refersto0.implicitName.s-Token + "CALL"))

in // id for pCALL

∃ s: state ∈

{ if si.s-<composite state list item> ≠ undefined

then si.s-<composite state list item>.s-<name>

elseif si.s-<typebased composite state> ≠ undefined

then si.s-<typebased composite state>.s-<name>

else si.s-<name> // <basic state name>

endif

| si ∈ s.s-<state list>.toSet }

∧ ( rpc ∈ **U**{ { ili.s-<signal list item>.s-<identifier> | ili ∈ input.s-<input list>.toSet }

| input ∈ s.s-implicit: input ∈ <input part> } ∨

pCALL ∈ **U**{ { ili.s-<signal list item>.s-<identifier> | ili ∈ input.s-<input list>.toSet }

| input ∈ s.s-implicit: input ∈ <input part> } ∨

rpc ∈ **U**{ { svi.s-<identifier> | svi ∈ save.s-<save list>.toSet }

| save ∈ s.s-implicit: save ∈ <save part> } ∨

pCALL ∈ **U**{ { svi.s-<identifier> | svi ∈ save.s-<save list>.toSet }

| save ∈ s.s-implicit: save ∈ <save part> }

)

endlet //pCALL

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

isSameProcedureAndSignature0(pd:<internal procedure definition>, ps: <procedure signature>):Boolean =def

**let** fpl = ps.procedureSignatureParameterList0 in

let fpl1 = pd.procedureFormalParameterList0 **in**

(fpl.length = fpl1.length) ∧

(∀i∈1..fpl.length:

(fpl[i].s-<parameter kind> = fpl1[i].parentAS0.parentAS0.s-<parameter kind>) ∧

isSameSort0(fpl[i].s-<sort>, fpl1[i].parentAS0.s-<sort>))∧

isSameResult0(ps.s-<result>, pd.s-<procedure heading>.s-<procedure result>))

endlet

Determine if two results are matching.

isSameResult0(r: <result>∪<procedure result>∪<operation result>,

r1: <result>∪<procedure result>∪<operation result>): Boolean =def

isSameSort0(r.s-<sort>,r1.s-<sort>)

The function timerInput0 produces an <input part> list (empty or one element) for <communication constraints> with <timer communication constraint> for a remote procedure call or remote variable import.

timerInput0(tcc: <timer communication constraint>): <input part>\* =def

if tcc = undefined then empty else

< mk-<input part>(undefined,

< mk-<stimulus>( mk-<signal list item>(tcc.s-<identifier>), tcc.s-<variable>-seq,undefined),

undefined,

undefined

>,

undefined,

mk-<terminator>(undefined,

mk-<join>(

if tcc.s-<name> ≠ undefined then tcc.s-<name> else tcc.s-<identifier>.s-<name> endif )))

>

endif

The function apar produces <actual parameters> for the values for the pCALL signal of a remote procedure call in the requesting agent.

apar(fpl: <formal parameter>\*, params: [ <actual parameter> ]+): [ <actual parameter> ]+=def

if params = undefined then undefined else

if fpl.head.s-implicit ≠ **out** then < params.head > **else** empty endif ⁀

if fpl.tail.length > 0 then apar(fpl.tail, params.tail) **else** empty endif

endif

The function commConstrNoTimer0 a list of communication constraints and returns the same list excluding any timer communication constraint in the list.

commConstrNoTimer0(ccl: <communication constraints>): <communication constraints> =def

if ccl.head ∈ <timer communication constraint> then empty **else** ccl.head endif ⁀

if ccl.tail.length > 0 then commConstrNoTimer0(ccl.tail) **else** empty endif

The function aINOUTpar produces the variable list for the pReply signal stimulus of a remote procedure call in the requesting agent.

aINOUTpar(fpl: <formal parameter>\*, params: [ <actual parameter> ]+, res: <variable>): <variable>\*=def

if params = undefined then undefined else

if fpl.head.s-implicit ≠ **in** ∧ fpl.head.s-implicit ≠ undefined then < params.head > **else** empty endif ⁀

if fpl.tail.length > 0 then aINOUTpar(fpl.tail, params.tail)

**else** if res ≠ undefined then res **else** empty endif

endif

endif

The function remoteProcParamsDef produces a list of variable definitions for the variables to hold the values passed via the pCALL and pREPLY signals for a remote procedure call. The function is passed a formal parameter list (of the remote procedure) and a Token (which should be the Token for the unique implicit name associated with the remote procedure definition). Each variable is given a new unique name formed from the Token parameter and the list position of the parameter (counting backwards from the end of the list).

remoteProcParamsDef(fpl:<formal parameter>\*, p: Token ):<variable definition>\*=def

if fpl.length = 0 then empty

else

< mk-<variable definition>(

< <variables of sort>(

PART,< <name>(p + natToIntToken(fpl.length)) >, fpl.head.s-<sort>, undefined) >

) >

⁀remoteProcParamsDef(fpl.tail, p)

endif

The function remoteProcParams produces a list of variable identifiers to hold the values passed via the pCALL signal for a remote procedure call before calling the exported procedure. The fpl parameter is the list of formal parameters for the remote procedure and the vars parameter is the list of variable definitions in server agent created from the list of formal parameters.

remoteProcParams(fpl: <formal parameter>\*, vars: <variable definition>\*):<identifier>\*=def

if fpl.length = 0 then empty

else

if fpl.head.s-implicit ≠ **out** then

< mk-<identifier>(vars.head.fullQualifier0, vars.head.s-<variables of sort>.s-<name>) >

**else** empty

endif

⁀ remoteProcParams(fpl.tail, vars.tail)

endif

The function exportedProcParams produces a list of variable identifiers to use in the call of the exported procedure after the input of pCALL. The fpl parameter is the list of formal parameters for the remote procedure and the vars parameter is the list of variable definitions in server agent created from the list of formal parameters.

exportedProcParams(fpl: <formal parameter>\*, vars: <variable definition>\*):<identifier>\*=def

if fpl.length = 0 then empty

else

< mk-<identifier>(vars.head.fullQualifier0, vars.head.s-<variables of sort>.s-<name>) >

⁀ exportedProcParams(fpl.tail, vars.tail)

endif

The function aINOUTremoteProcParams produces variable identifier list for the values passed via the pREPLY signal of a remote procedure call with the list being in the order of each IN/OUT or OUT parameter.

aINOUTremoteProcParams(fpl: <formal parameter>\*, vars: <variable definition>\*):<identifier>\*=def

if fpl.length = 0 then empty

else

if fpl.head.s-implicit ≠ **in** ∧ fpl.head.s-implicit ≠ undefined then

< mk-<identifier>(vars.head.fullQualifier0, vars.head.s-<variables of sort>.s-<name>) >

**else** empty

endif

⁀aINOUTremoteProcParams(fpl.tail, vars.tail)

endif

#### F2.2.7.6 Remote variable

NOTE – A remote variable enables an exported value of a variable owned by one agent (the exporter) to be imported into another agent (the importer) by using an <import expression>, and because a remote variable is allowed to be used by more than one exporter, it is allowed to direct the request for the exported value to be to a particular agent (or agent instance set) by the <destination> in the <communications constraint> of the <import expression>. In the transform (described below) the <output> request includes the <destination> of the <import expression>, and output compatibility (that the signal is receivable by the destination) applies for the request. If the <destination> is a <pid<expression> with a pid sort (the sort of an Interface-definition) other than Pid, AS1 is statically checked for output compatibility. If the sort is Pid, output compatibility needs to be checked dynamically.

Concrete syntax

<remote variable definition> :: <remote variables of sort>+

<remote variables of sort> ::

<remote variable<name>+ <sort> [ **nodelay** ]

<export statement> :: <export body>

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

Conditions on concrete syntax

See clause 10.6 *Model* of [ITU‑T Z.102] unless another reference is given.

∀v ∈ <exported variables of sort>: ∀ev in v.**s**-<exported variable>-seq:

ev.**s**-<identifier> ≠ undefined⇒

isSameSort0(ev.parentAS0.s-<sort>,getEntityDefinition0(ev.**s**-<identifier>,remote variable).s-<sort>)

The <remote variable identifier> following **as** in an exported variable definition shall denote a <remote variable definition> of the same sort as the exported variable definition.

∀v ∈ <exported variables of sort>:∀ev in v.**s**-<exported variable>-seq:

let rvd = resolveByContainer0(ev.surroundingScopeUnit0,

mk-<identifier>(empty, ev.s-<name>),

remote variable)

in

rvd ≠ undefined ∧ ev.s-<identifier> = undefined⇒

isSameSort0(ev.parentAS0.s-<sort>, rvd.s-<sort>)

endlet

In the 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.

∀vid ∈ <identifier>: vid.parentAS0∈<export body>⇒getEntityDefinition0(vid,variable).isExported0

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

∀dest∈<identifier> ∪ this: ∀importExp∈<import expression>:

dest. parentAS0 = importExp ⇒

( let def =

if dest ∈ <identifier>

then getEntityDefinition0(dest, sort)

else getEntityDefinition0(

parentAS0ofKind(dest, <agent definition> ∪ <agent type definition>).s-<identifier>, sort)

endif

in

let rv = getEntityDefinition0(importExp.s-<identifier>, remote variable) in

def ∈ <interface definition> ∧ isDefinedIn0(rv, def)

endlet

endlet)

If <destination> in <communication constraints> of an <import expression> is an <agent<identifier> or this, the <remote variable identifier> shall represent a remote variable contained in the interface of the agent type.

∀v ∈ <exported variables of sort>:

parentAS0ofKind(v, <internal procedure definition>) = undefined

∀ie1, ie2 ∈ <import expression>:

let action1 = parentAS0ofKind(ie1, <action>) in

let action2 = parentAS0ofKind(ie2, <action>) in

ie1.s-<identifier> = ie2.s-<identifier> ⇒

action1 ≠ action2

∨ ¬isSameNode0(action1, action2)

∨ isSameNode0(ie1, ie2)

endlet

endlet

Each action shall contain no more than one <import expression> for the same remote variable.

A procedure variable is not allowed to be exported (See clause 9.4 *Semantics* of [ITU‑T Z.101]). The definition of **exported** variables is not allowed in a <variable definition> in a <procedure definition> (See clause 9.4 *Concrete grammar* of [ITU‑T Z.103]).

Transformations

The use of <import expression> in an expression is shorthand for inserting a task just before the action, where the expression occurs which assigns to 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. See clause 12.3.4.5 *Model* of [ITU‑T Z.104].

An import access is modelled by exchange of implicitly defined signals. 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. See clause 10.6 *Model* of [ITU‑T Z.102].

rvd.implicitName

provided

rvd ∈ <remote variable definition>

∧ rvd.implicitName = undefined

=17=>

( implicitName\{( rvd, undefined )} ) ∪ {( rvd, newName)}

rvd=<remote variable definition>(< <remote variables of sort>(< n >, sort, delay) >)

provided

rvd.implicitName ≠ undefined // implicit name defined

∧ ¬ (∃ sigdef ∈ <signal definition>: sigdef.s-<name>.s-Token = rvd.implicitName.s-Token +"QUERY")

// signal xQUERY already defined for this remote variable

=17=>

let x = rvd.implicitName.s-Token in

rvd ⁀ mk-<signal definition list>(

< mk-<signal definition> (undefined,

mk-<name>(x + "QUERY"), empty, undefined, undefined, < predefinedId0("Integer") >) ,

mk-<signal definition> (undefined,

mk-<name>(x + "REPLY"), empty, undefined, undefined, < sort, predefinedId0("Integer") >)

> )

endlet // x

There are two implicit signal definitions for each variable of a <remote variable definition> in a system definition. A <remote variable definition> that defines multiple variables is expanded to a <remote variable definition> list with one variable per <remote variable definition>. The <signal name>s in the implicit signal definitions are denoted by xQUERY and xREPLY respectively, where x denotes an implicit <name> associated with 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. See clause 10.6 *Model* of [ITU‑T Z.102].

<channel definition>(n, er, delay,

<channel path>(ep1, ep2,

sigs1 ⁀ <signal list item>(i=<identifier>(q, \*)) ⁀ sigs2),

path2)

provided

stimulusKind(i) = **remote** **variable**

∧ i.refersto0.implicitName ≠ undefined

=17=>

let x = i.refersto0.implicitName.s-Token in

mk-<channel definition>(n, er, delay,

mk-<channel path>(ep1, ep2,

sigs1 ⁀ < mk-<identifier>(q, mk-<name>(x + "QUERY")) > ⁀ sigs2),

mk-<channel path>(ep2, ep1,

if path2 = undefined then empty else path2.s-<signal list item>-seq endif ⁀

< mk-<identifier>(q, mk-<name>x + "REPLY")) >))

endlet

On each channel mentioning the remote variable, the remote variable is replaced by xQUERY. For each such channel, if it is unidirectional the channel is made bidirectional. In the opposite direction this channel carries the signal xREPLY. See clause 10.6 *Model* of [ITU‑T Z.102].

a) Importer

See clause 10.6 *Model* (a) Importer of [ITU‑T Z.102].

The first transformation below inserts variable definitions for the implicit variables needed for an import expression provided a unique name associated with the remote variable has been introduced and the variables have not already been defined. The second transformation updates the access of the remote variable to be an access of the implicit variable used to store the import value from the signal exchange for the import, and is followed by dependent transformations one of which inserts the items for the signal exchange before the variable access. Depending on where the import expression was, the pattern matches one of the six dependent transformations for the enclosing body: an agent body with the import expression in a start, an agent body with the import expression in a state or free action, a composite state body with the import expression in a start, a composite state body with the import expression in a state or free action, a procedure body with the import expression in a start, or a procedure body with the import expression in a state or free action.

items=getEntities0(//entities defined in agent type definition where import expression is used

atd = parentAS0ofKind( // finds agent type definition with import expression

r = <import expression>(rv, \*),

<system type definition> ∪ <block type definition> ∪ <process type definition>))

)) //end parentAS0ofKind, end of getEntities0

provided

¬ callerVariablesDefined(r)

=17=>

< variableDefinition0(rv.refersto0.implicitName.s-Token + "xn", rv.refersto0.s-<sort>, undefined),

variableDefinition0(rv.refersto0.implicitName.s-Token + "n", predefinedId0("Integer"),

<operand5>( <identifier>( predefinedQual0("Integer"), <name>("1")))),

variableDefinition0(rv.refersto0.implicitName.s-Token + "newn", predefinedId0("Integer"), undefined)

> // end of variable definitions

⁀ items // variables are in scope and can be used in items

<variable access>(r = <import expression>(rv=<identifier>(q, \*), constr))

provided

callerVariablesDefined(r)

=17=>

mk-<variable access>( mk-<identifier>(

fullQualifierWithin0( parentAS0ofKind( r,

<system type definition> ∪ <block type definition> ∪ <process type definition>)),

mk-<name>(rv.refersto0.implicitName.s-Token + "xn")))

and

body = <agent body>( // agent body with import expression in start

start = <start>(virt,

entry,

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(r,

<action>), // parentAS0ofKind for action

<action>-seq), // parentAS0ofKind for actionItems,

tr) // tr is the terminator for <transition action items>

), // <start>

bodyitems

) // agent body with import expression in start

=>

let actions1 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ n < i > in

let actions2 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ i < n > in

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

let newLabel = newName in

let newAction = mk-<action>(newLabel, action.s-implicit) in

let newactions1 = if actions1= empty then mk-<terminator>( mk-<join>(oldLabel))

else mk-<transition action items>(actions1, mk-<terminator>( mk-<join>(oldLabel))) endif

in

let newactions2 = < mk-<free action>( mk-<transition action items>(< newAction >⁀actions2, tr)) >

in

mk-<agent body>( mk-<start>(virt, entry, newactions1),

newImportItems(r, oldlabel, newlabel) ⁀ newactions2⁀ bodyitems

)// end agent body with import expression in start

endlet // newactions2

endlet // newactions1

endlet // newAction

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

and

body = <agent body>( // agent body with import expression in state or free action

start,

bodyitems = parentAS0ofKind(

bodyitem = parentAS0ofKind(

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(r, <action>),

<action>-seq), // parentAS0ofKind for actionItems,

tr), // tr is the terminator for <transition action items>

(<state> ∪ <free action>)) // parentAS0ofKind for bodyitem

(<state> ∪ <free action>)-seq) // parentAS0ofKind for bodyitems

) // agent body with import expression in state or free action

=>

let actions1 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ n < i > in

let actions2 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ i < n > in

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

let newLabel = newName in

let newAction = mk-<action>(newLabel, action.s-implicit) in

let newactions1 = if actions1= empty then mk-<terminator>( mk-<join>(oldLabel))

else mk-<transition action items>(actions1, mk-<terminator>( mk-<join>(oldLabel))) endif

in

let newactions2 = < mk-<free action>( mk-<transition action items>(< newAction >⁀actions2, tr)) >

in

mk-<agent body>( start,

if bodyitem ∈ <state>

then

bilistpre0(bodyitems,bodyitem)⁀< revisedState0(bodyitem,action,newactions1) >

⁀newbodyitems0⁀newactions2⁀bilistpost0(bodyitems,bodyitem)

else // free action

bilistpre0(bodyitems,bodyitem)⁀ < mk-<free action>(newactions1) >

⁀ newImportItems(r, oldlabel, newlabel)⁀newactions2⁀bilistpost0(bodyitems,bodyitem)

endif // state or free action

) // agent body with import expression in state or free action

endlet // newactions2

endlet // newactions1

endlet // newAction

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

and

body = <composite state body>( // composite state body with import expression in start

startlist1⁀

< start = <start>(virt,

entry,

<transition action items>(

actionItems = parentAS0ofKind( action =

parentAS0ofKind(r,

<action>), // parentAS0ofKind for action

<action>-seq), // parentAS0ofKind for actionItems,

tr) // tr is the terminator for <transition action items>

) > // start list item containing import expression r

⁀startlist2,

bodyitems

) // composite state body with import expression in start

=>

let actions1 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ n < i > in

let actions2 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ i < n > in

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

let newLabel = newName in

let newAction = mk-<action>(newLabel, action.s-implicit) in

let newactions1 = if actions1= empty then mk-<terminator>( mk-<join>(oldLabel))

else mk-<transition action items>(actions1, mk-<terminator>( mk-<join>(oldLabel))) endif

in

let newactions2 = < mk-<free action>( mk-<transition action items>(< newAction >⁀actions2, tr)) >

in

mk-<composite state body>( startlist1⁀< mk-<start>(virt, entry, newactions1) >⁀startlist2,

newImportItems(r, oldlabel, newlabel) ⁀ newactions2⁀ bodyitems

) // composite state body with import expression in start

endlet // newactions2

endlet // newactions1

endlet // newAction

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

and

body = <composite state body>( // composite state body with import expression in state or free action

startlist,

bodyitems = parentAS0ofKind(

bodyitem = parentAS0ofKind(

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(r, <action>),

<action>-seq), // parentAS0ofKind for actionItems,

tr), // tr is the terminator for <transition action items>

(<state> ∪ <free action>)) // parentAS0ofKind for bodyitem

(<state> ∪ <free action>)-seq) // parentAS0ofKind for bodyitems

) // composite state body with import expression in state or free action

=>

let actions1 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ n < i > in

let actions2 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ i < n > in

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

let newLabel = newName in

let newAction = mk-<action>(newLabel, action.s-implicit) in

let newactions1 = if actions1= empty then mk-<terminator>( mk-<join>(oldLabel))

else mk-<transition action items>(actions1, mk-<terminator>( mk-<join>(oldLabel))) endif

in

let newactions2 = < mk-<free action>( mk-<transition action items>(< newAction >⁀actions2, tr)) >

in

mk-<composite state body>( startlist,

if bodyitem ∈ <state>

then

bilistpre0(bodyitems,bodyitem)⁀< revisedState0(bodyitem,action,newactions1) >

⁀newbodyitems0⁀newactions2⁀bilistpost0(bodyitems,bodyitem)

else // free action

bilistpre0(bodyitems,bodyitem)⁀ < mk-<free action>(newactions1) >

⁀ newImportItems(r, oldlabel, newlabel)⁀newactions2⁀bilistpost0(bodyitems,bodyitem)

endif // state or free action

) // composite state body with import expression in state or free action

endlet // newactions2

endlet // newactions1

endlet // newAction

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

and

body = <procedure body>( // procedure body with import expression in start

start = <start>(virt,

entry,

<transition action items>(

actionItems = parentAS0ofKind( action =

parentAS0ofKind(r,

<action>), // parentAS0ofKind for action

<action>-seq), // parentAS0ofKind for actionItems,

tr) // tr is the terminator for <transition action items>

), // <start>

bodyitems

) // procedure body with import expression in start

=>

let actions1 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ n < i > in

let actions2 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ i < n > in

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

let newLabel = newName in

let newAction = mk-<action>(newLabel, action.s-implicit) in

let newactions1 = if actions1= empty then mk-<terminator>( mk-<join>(oldLabel))

else mk-<transition action items>(actions1, mk-<terminator>( mk-<join>(oldLabel))) endif

in

let newactions2 = < mk-<free action>( mk-<transition action items>(< newAction >⁀actions2, tr)) >

in

mk-<procedure body>( mk-<start>(virt, entry, newactions1),

newImportItems(r, oldlabel, newlabel) ⁀ newactions2⁀ bodyitems

)// end procedure body with import expression in start

endlet // newactions2

endlet // newactions1

endlet // newAction

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

and

body = <procedure body>( // procedure body with import expression in state or free action

start,

bodyitems = parentAS0ofKind(

bodyitem = parentAS0ofKind(

<transition action items>(

actionItems = parentAS0ofKind(

action = parentAS0ofKind(r, <action>),

<action>-seq), // parentAS0ofKind for actionItems,

tr), // tr is the terminator for <transition action items>

(<state> ∪ <free action>)) // parentAS0ofKind for bodyitem

(<state> ∪ <free action>)-seq) // parentAS0ofKind for bodyitems

) // procedure body with import expression in state or free action

=>

let actions1 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ n < i > in

let actions2 = < a[n] in actionItems.s-<action>-seq : action = actionItems.s-<action>-seq[i] ∧ i < n > in

let oldLabel = if action.s-<label> = undefined then newName else action.s-<label> endif in

let newLabel = newName in

let newAction = mk-<action>(newLabel, action.s-implicit) in

let newactions1 = if actions1= empty then mk-<terminator>( mk-<join>(oldLabel))

else mk-<transition action items>(actions1, mk-<terminator>( mk-<join>(oldLabel))) endif

in

let newactions2 = < mk-<free action>( mk-<transition action items>(< newAction >⁀actions2, tr)) >

in

mk-<procedure body>( start,

if bodyitem ∈ <state>

then

bilistpre0(bodyitems,bodyitem)⁀< revisedState0(bodyitem,action,newactions1) >

⁀newbodyitems0⁀newactions2⁀bilistpost0(bodyitems,bodyitem)

else // free action

bilistpre0(bodyitems,bodyitem)⁀ < mk-<free action>(newactions1) >

⁀ newImportItems(r, oldlabel, newlabel)⁀newactions2⁀bilistpost0(bodyitems,bodyitem)

endif // state or free action

) // procedure body with import expression in state or free action

endlet // newactions2

endlet // newactions1

endlet // newAction

endlet // newLabel

endlet // oldLabel

endlet // actions2

endlet // actions1

– For each imported variable, two implicit Integer variables (in this description called n and newn) are defined in the enclosing scope unit of the <import expression>, and n is initialized to 0. In addition, an implicit anonymous variable (in this description called xn) of the sort of the remote variable is defined.

– The <import expression>

import (x to destination via via-path)

is transformed so that the following is inserted before the action that contained the <import expression>, where in the output 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:

Rick's MacBook Pro:Users:rickreed:Documents:ITU:wpx 2017-2020:170829 Q12 Geneva:T17-SG17-170829-TD-PLEN-0625 Annex F2:Document3.pdf

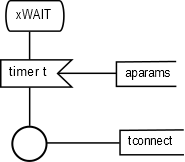
– The insertion is labelled with the label on the action containing the import expression or a new label if this action is not labelled, and the preceding path is changed to join this label.

– The true path above is terminated with a join to the action that contained the import expression with a new label.

– The import expression is changed to an access of the variable xn.

– In all other states, xREPLY is discarded. This is not explicitly modelled: instead the handling of xREPLY is left unspecified in the transformed concrete syntax except for the xWAIT states with the consequence that there is an implicit transition (see clause 11.8 of [ITU-T Z.103]) for other states that discards the signal.

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



where

t is the <timer identifier> in the <timer communication constraint>;

aparams is the optional list of optional <variable> items given after the <timer identifier> in the <timer communication constraint>;

tconnect is the <connector name> if one is given in the <timer communication constraint>; otherwise tconnect is the name of the timer.

b) Exporter

See clause 10.6 *Model* (b) Exporter of [ITU‑T Z.102].

The first transform below handles the variable definitions for implicit anonymous variable to store the exported value and the implicit anonymous variable to store the integer passed received in xQUERY and returned in xREPLY. The subsequent three transforms modify the bodies of agents, composite states and procedures (respectively) where an exported variable is in scope to include the xQUERY handler on all states.

v = <variable definition>( < <exported variables of sort>(ak, < <exported variable>(\*, rv) >, sort), const) >)

provided

rv.refersto0.implicitName ≠ undefined // implicit name defined

∧¬(∃ vos ∈ <variables of sort>: vos.s-<name>-seq[1].s-Token = rv.refersto0.implicitName.s-Token+"imcx"

∧ surroundingQualifier0(vos) = surroundingQualifier0(v)) // imcx not defined

=17=>

let imcxName = mk-<name>(rv.refersto0.implicitName.s-Token+"imcx") in

v ⁀ mk-<variable definition>( < mk-<variables of sort>( ak, < imcxName >, sort, const) >)

⁀ variableDefinition0(rv.refersto0.implicitName.s-Token+"n", predefinedId0("Integer"), undefined) >)

endlet

For each exported variable, an implicit anonymous variable (in this description denoted by imcx) is defined to hold the exported value of the exported variable, and an implicit anonymous variable of type Integer (in this description denoted by n) is defined.

If a default initialization is attached to the export variable or if the export variable imcx is initialized when it is defined, then the implicit copy is also initialized with the same result as the export variable.

body = <agent body>(start,bodyitems)

provided

( ∃ ev ∈ <exported variable>: exportedVarScope0(ev, body))

//There exists an exported variable ev whose scope encloses the body, with no xQUERY input parts

=17=>

mk-<agent body>(start, newBodyitems0(body))

body = <composite state body>(startlist,bodyitems)

provided

( ∃ ev ∈ <exported variable>: exportedVarScope0(ev, body))

//There exists an exported variable ev whose scope encloses the body, with no xQUERY input parts

=17=>

mk-<composite state body>(startlist, newBodyitems0(body))

body = <procedure body>(start,bodyitems)

provided

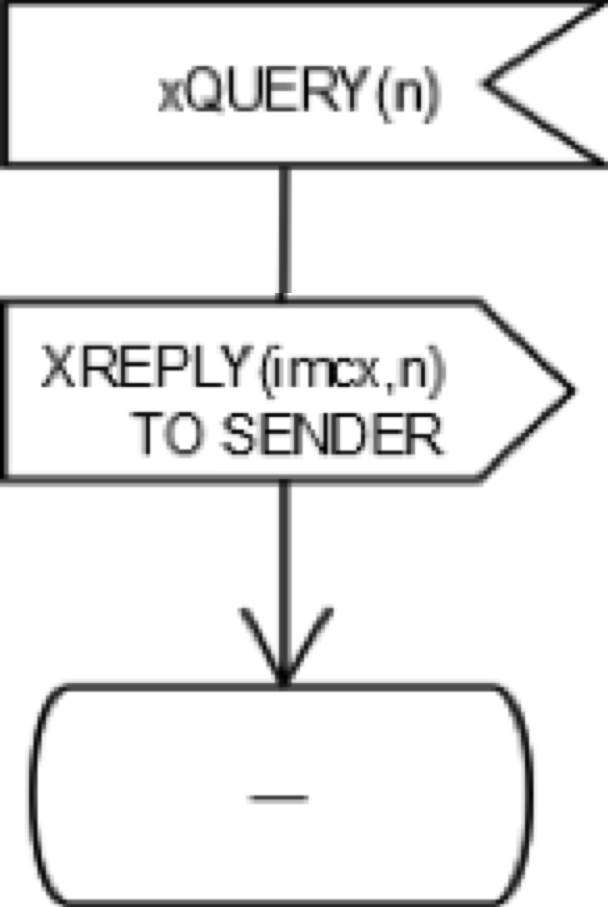
( ∃ ev ∈ <exported variable>: exportedVarScope0(ev, body))

//There exists an exported variable ev whose scope encloses the body, with no xQUERY input parts

=17=>

mk-<procedure body>(start, newBodyitems0(body))

– To all states of the exporter, the following input is added:



<export statement>(<export body>(< id >))

=17=>

<task>(<assignment>(id.refersto0.implicitName, id))

The <export statement>

export x

is transformed to the following:

task imcx:= x;

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

The keyword **nodelay** has no SDL‑2010 meaning, though to be compatible with SDL‑92 the channel conveying the signals for the remote variable should be a channel without delay.

Auxiliary functions

The function newImportItems is used to define a list of <free action> and <state> items to insert into the body of an agent, state or procedure that has an import expression.

newImportItems(r: <import expression>, oldlabel: <label>, newlabel: <label>):

(<free action> ∪ <state>)\* =def

let sq = fullQualifierWithin0( parentAS0ofKind( r,

<system type definition> ∪ <block type definition> ∪ <process type definition>))

in

let rv = r.s-<identifier> in

let constr= r.s-<communication constraints> in

let xnName = mk-<name>(rv.refersto0.implicitName.s-Token + "xn") in

let nName = mk-<name>(rv.refersto0.implicitName.s-Token + "n") in

let n = mk-<operand5>( mk-<variable access>( mk-<identifier>( sq, nName))) in

let newnName = mk-<name>(rv.refersto0.implicitName.s-Token + "newn") in

let newn= mk-<operand5>( mk-<variable access>( mk-<identifier>( sq, newnName ))) in

let xWAIT = newName in

let xQUERY = mk-<name>(rv.refersto0.implicitName.s-Token + "QUERY")in

let xREPLY = mk-<name>(rv.refersto0.implicitName.s-Token + "REPLY")in

< mk-<free action>(

mk-<transition action items>(<

mk-<action>( oldLabel, mk-<task>( mk-<assignment>( mk-<identifier>(sq,nName),

mk-<operator application>("+", < n, predefinedItem0("Integer","1") >)))),

mk-<action>(undefined, mk-<output>( mk-<output body>(

mk-<output body item>( mk-<identifier>(q, xQUERY), < n >),

commConstrNoTimer0(constr))),

mk-<terminator>(undefined, mk-<nextstate>( mk-<nextstate body name>(xWAIT)))

> ), undefined)

> ⁀

< mk-<state>(< xWAIT >,

timerInput0(constr.s-<timer communication constraint>) ⁀

< mk-<save part>(undefined, <asterisk save list>()),

mk-<input part>(undefined, //virtuality

< mk-<stimulus>( mk-<signal list item>( mk-<identifier>(q, xREPLY)),

< mk-<identifier>(sq, xnName),

mk-<identifier>(sq, newnName ) >, undefined) // stimulus,

undefined, // in choice

undefined // priority clause

>,

undefined, // enabling condition

mk-<transition action items>(

mk-<decision>( mk-<operand5>( mk-<operator application>("=", < newn, n >)),

mk-<textual answer part>(predefinedItem0("Boolean","true"),

mk-<terminator>( mk-<join>(newLabel)))

mk-< textual answer part>(predefinedItem0("Boolean","false"),

mk-<terminator>( mk-<nextstate>(xWAIT)))

) // transition

>, // state parts

undefined ) // state timer part

> // for other states nothing is specified for xREPLY so it is treated as an implicit transition

endlet // xREPLY

endlet // xQUERY

endlet // xWAIT

endlet // newn

endlet // newnName

endlet // n

endlet // nName

endlet // xnName

endlet // constr

endlet // sq

The function exportedVarScope0 determines if there is an agent type definition that has a variable definition where the given exported variable ev is defined, and encloses the given body in which there is no input part that handles the xQUERY signal for the exported variable.

exportedVarScope0(ev: <exported variable>,

body: <agent body> ∪ <composite state body> ∪ <procedure body>): Boolean =def

// ev.s-<identifier> is remote variable identifier

// <exported variable> is only in <variable definition> which at this point has one variable per definition

ev.s-<identifier>.refersto0.implicitName ≠ undefined // implicit name defined

∧ isSameNode0( // nearest enclosing agent type def of body and ev the same agent type definition

parentAS0ofKind(body,<system type definition> ∪ <block type definition> ∪ <process type definition>),

parentAS0ofKind(ev,<system type definition> ∪ <block type definition> ∪ <process type definition>))

∧ ¬ (∃ input ∈ <input part>:

parentAS0ofKind(input, <agent body> ∪ <composite state body> ∪ <procedure body>) = body

∧ parentAS0ofKind(rv.refersto0.implicitName.s-Token +"QUERY", <input part>) = input)

// does not exist an input part in body for xQUERY

The function newBodyItems0 takes a body (an agent body, composite state body or procedure body), and provides a replacement list for the body items of the body where every state has a handler for xQUERY added.

newBodyItems0(body: <agent body> ∪ <composite state body> ∪ <procedure body>):

(<state> ∪ <free action>)-seq =def

let bodyitems = body.s-(<state> ∪ <free action>)-seq in

let evar = take({ev ∈ <exported variable> : exportedVarScope0(ev: <exported variable>, body }) in

let rv = evar.s-<identifier> in

let x = rv.refersto0.implicitName.s-Token in

let imcx = <operand5>(

mk-<variable access>(

mk-<identifier>(surroundingQualifier0(evar), mk-<name>(x+"imcx"))

)) // variable access, operand5

in

let nid = mk-<identifier>(surroundingQualifier0(evar), mk-<name>(x+"n")) in

let n = mk-<operand5>(mk-<variable access>(nid)) in

let xQUERY = mk-<name>(rv.refersto0.implicitName.s-Token + "QUERY")in

let xREPLY = mk-<name>(rv.refersto0.implicitName.s-Token + "REPLY")in

< if bodyitems[i] ∈ <state> // bodyitems is {<state> | <free action>}\*

then // body item is a state, add xQUERY input

// note – expanded to one state per <state> at ths point

mk-<state>(

bodyitems[i].s1-implicit, // state name, typebased comp state …

bodyitems[i].s2-implicit // old list of connect part, input part …

⁀< mk-<input part>( // add input part for xQUERY

undefined, // virtuality

< mk-<stimulus>(

mk-<signal list item>(

mk-<identifier>(rv.s-<qualifier>,xQUERY)), < nid >,undefined), // stimulus

undefined, // in choice

undefined // priority clause

>, // simulus list

undefined, // enabling condition

mk-<transition action items>( <

mk-<action>(undefined, mk-<output>( mk-<output body>(

mk-<output body item>(<identifier>(rv.s-<qualifier> xREPLY),

< imcx, n>), // output body item

undefined)), output body, output

mk-<terminator>(undefined, mk-<dash nextstate>) // action

>) // action list, transition action items

) >, // input part, parts list containing input part

bodyitems[i].s-<state timer part> // state timer for xQUERY input part

) // state

else // body item is a free action – not changed

bodyitems[i]

endif

| i ∈ 1.. length(bodyitems) > // each item of bodyitems

endlet // xREPLY

endlet // xQUERY

endlet // n

endlet // nid

endlet // imcx

endlet // x

endlet //rv

endlet // evar

endlet // bodyitems

#### F2.2.7.7 Communication path encoding rules, encode and decode

See clause 10.7 of [ITU‑T Z.104].

Abstract syntax

Encoding-rules :: Rules-identifier

Encode-procedure-identifier

Decode-procedure-identifier

Encoding-expression :: Signal-identifier

[ Expression]\*

Encoding-path

Encoding-path :: Gate-identifier

| Data-type-identifier Encoding-rules

Decoding-expression :: Expression

Encoding-path

Rules-identifier = Literal-identifier

Encode-procedure-identifier = Procedure-identifier

Decode-procedure-identifier = Procedure-identifier

Conditions on abstract syntax

The Rules-identifier shall be one of the literal identifiers of the data type Encoding. The data type Encoding shall be the Predefined data type Encoding or a data type with the name Encoding that is a specialization (direct or indirect) of the Predefined data type Encoding. The specialization shall only add literal names to the Predefined data type Encoding and shall not change any other properties.

If the Rules-identifier corresponds (by Name) to an Encoding literal defined in the package Predefined, built-in procedures implied by the standardized sets of encoding rules are invoked (and other procedures – even if visible with names corresponding as below – are ignored). These built-in encode and decode procedures are implicit parts of package Predefined.

If the Rules-identifier corresponds to an additional literal added to a specialization of the predefined data type Encoding, there shall be a visible procedure with the appropriate signature for each invocation (implicit or explicit) of the Encoding-expression or Decoding-expression.

The name of the procedure for encode is the name encode concatenated with the name of an Encoding literal. This procedure shall have one parameter of the implicit choice type for the relevant path for the invocation. The procedure shall return a Charstring, Bitstring or Octetstring.

The name of the procedure for decode is the name decode concatenated with the name of an Encoding literal. This procedure shall have one parameter of the same type (Charstring, Bitstring or Octetstring) as the corresponding procedure for encode, and shall return the choice type for the relevant path for the invocation.

Each encode or decode procedure shall be functional (that is, it shall not contain states and shall not change the value of any variable external to the procedure when it is interpreted).

The length of the optional Expression list of an Encoding-expression shall be the same as the number of Sort-reference-identifiers in the Signal-definition denoted by the Signal-identifier.

Each Expression in an Encoding-expression shall be sort compatible to the corresponding (by position) Sort-reference-identifier in the Signal-definition denoted by the Signal-identifier.

For a Gate-identifier of an Encoding-path of an agent's Encoding-expression, the gate shall be a gate of the agent or reachable via a channel with the Signal-identifier of the Encoding-expression from the agent, and the Out-signal-identifier-set of the gate shall include the Signal-identifier.

The Data-type-identifier of an Encoding-path shall identify an Interface-definition.

For an Encoding-path of an Encoding-expression with a Data-type-identifier that identifies an Interface-definition, the Signal-identifier shall identify one of the signals defined or inherited by the interface identified.

The Expression in a Decoding-expression shall be compatible with the sort generated by an Encoding-expression using the same Encoding-path.

Concrete syntax

<encoding rules> :: <rules identifier>

<rules identifier> :: <literal>

<encoding expression> :: { <signal<identifier> [ <actual parameters> ] | <expression> }[ <encoding path> ]

<encoding path> :: { <channel<identifier> | <gate<identifier> | <interface<identifier> <rules<identifier> }

<decoding expression> :: <expression> [ <encoding path> ]

Conditions on concrete syntax

Further study required for handling <encoding rules>.

In an <encoding expression> with an <expression>, the sort of the <expression> shall be the sort of the choice data type corresponding to the set of signals for the <encoding path>: normally the choice data type defined for the channel, gate or interface.

The <encoding path> shall only be omitted from <encoding expression> if there is exactly one path (channel or gate) with encoding for output of the signal in the context of the <encoding expression> and, in this case, the encoding for that path is used.

The <encoding path> shall only be omitted from <decoding expression> if there is exactly one path (channel or gate) with encoding for input in the context of the decoding expression and, in this case, the encoding for that path is used.

The encode and decode procedures associated with a <rules identifier> shall be visible where the <rules identifier> is used.

Transformations

If an <encoding expression> contains an <expression>, the choice present is determined from the expression, and the signal with the same name as the choice is output with the values of the signal parameters given by the expression.

If an <encoding path> contains a <channel<identifier>, this is transformed to the (possibly anonymous) <gate<identifier> for the gate of the identified channel nearest to the agent containing the <encoding path>.

Mapping to abstract syntax

Further study required.

### F2.2.8 Behaviour

#### F2.2.8.1 Start

Abstract syntax

State-start-node :: Transition

Procedure-start-node :: Transition

Concrete syntax

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

Conditions on concrete syntax

∀s∈<start>:

(s.**s**-<name> ≠ undefined) ⇒

(s.surroundingScopeUnit0∈<composite state definition> )∨

(s.surroundingScopeUnit0∈<composite state type definition>)

If <state entry point name> is given in a <start>, the <start> shall be the <start> of a composite state. See clause 11.1 *Concrete grammar* of [ITU‑T Z.102].

NOTE – A <composite state definition> is transformed into a typebased composite state and a <composite state type definition>.

Mapping to abstract syntax

| s=<start>(\*,entry,trans) then

if s.parentAS0 ∈ <procedure body>

then mk-Procedure-start-node(Mapping(trans))

elseif entry = undefined

then mk-State-start-node(Mapping(trans) )

else mk-Named-start-node(Mapping(entry), Mapping(trans))

endif

#### F2.2.8.2 State

Abstract syntax

State-node :: State-name

Save-signalset

Input-node-set

{Spontaneous-transition-set Continuous-signal-set

| Composite-state-type-identifier Connect-node-set }

[State-timer]

*State-timer* :: *Time-expression*

*Timer-identifier*

*Expression*\*

*Transition*

Conditions on abstract syntax

∀sn, sn1 ∈ State-node:(sn≠sn1)∧(sn.parentAS1=sn1. parentAS1)⇒(sn.**s**-Name≠sn1.**s**-Name)

Each State-node within a graph (State-transition-graph or Procedure-graph) shall have a State-name different from any other State-name in the same graph. See clause 11.2 of [ITU‑T Z.101].

NOTE – A State-name is defined to be the same as a Name.

∀ sn ∈ State-node: ∀svi,svi1∈Save-signalset: ∀in,in1∈Input-node:

(svi≠svi1)∧(svi.parentAS1=sn)∧(svi1.parentAS1=sn) // svi and svi1 in same state node

∧ (in≠in1)∧(in.parentAS1=sn)∧(in1.parentAS1=sn) // in and in1 in same state node

⇒

( in.s2-Identifier = undefined⇒ // input has no gate implies

( ((in.s1-Identifier ≠ in1.s1-Identifier) ∨ (in1.s2-Identifier ≠ undefined))//in1 different signal or has gate

∧ ((in.s1-Identifier ≠ svi.s1-Identifier) ∨ (svi.s2-Identifier ≠ undefined))//svi diff. signal or has gate

∧ ((in.s1-Identifier ≠ svi1.s1-Identifier) ∨ (svi1.s2-Identifier ≠ undefined))//svi1 diff. signal or has gate

) )

∧

( in.s2-Identifier≠undefined⇒ // input has a gate implies

( (in.s1-Identifier ≠ in1.s1-Identifier) //different signal in in1

∨ (in.s2-Identifier ≠ in1.s2-Identifier) //different gate in in1 (including undefined gate)

∨ (in.s1-Identifier ≠ svi.s1-Identifier) //different signal in svi

∨ (in.s2-Identifier ≠ svi.s2-Identifier) //different gate in svi (including undefined gate)

∨ (in.s1-Identifier ≠ svi1.s1-Identifier) //different signal in svi1

∨ (in.s2-Identifier ≠ svi1.s2-Identifier) //different gate in svi1 (including undefined gate)

) )

In the following, *Save-item* and *Input-node* refer, respectively, to a *Save-item* of the *Save-signalset* of a basic *State-node* and an *Input-node* of the *Input-node-set* of the same basic *State-node*. If an *Input-node* has no *Gate*-identifier, the *Signal-identifier* of that *Input-node* shall not appear without a *Gate-identifier* in another *Input-node* or a *Save-item*. If an *Input-node* has a *Gate-identifier*, the Signal-identifierof that *Input-node* shall not appear with the same *Gate-identifier* in another *Input-node* or a *Save-item*. See clause 11.2 of [ITU‑T Z.101].

NOTE – Both Signal-identifier and Gate-identifier are defined to be the same as an Identifier. In both Input-node and Save-item, the Signal-identifier is followed by an optional Gate-identifier and are therefore selected by s1-Identifier ands2-Identifier respectively.

Concrete syntax

<state> ::

<state list>

{ <connect part>

| <continuous signal>

| <input part>

| <priority input>

| <save part>

| <spontaneous transition>}\*

[ <state timer part> ]

NOTE – Although the concrete grammar allows <continuous signal> and <spontaneous transition> for a <state> with a <state list item> that is a <typebased composite state> or <composite state list item>, for a composite state in the abstract grammar Spontaneous-transition and Continuous-signal are not allowed, therefore they are only valid for basic state items.

<state list> ::

{<basic state name> | <typebased composite state> | <composite state list item>}+

| <asterisk state list>

<basic state name> = <basic state<name>

<composite state list item> :: <composite state name> <nextstate parameters>

<composite state name> = <composite state<name>

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

<state timer part> ::

[ <virtuality> ] <state timer> <transition>

<state timer> = <Time<expression> | <set clause>

Conditions on concrete syntax

∀bs∈<state>:∀sn∈<name>:

(sn.parentAS0 = bs.s-<state list>∧ bs.s-<state list> ∈ <asterisk state list>) ⇒

(∀sn1∈<name> (sn.parentAS0= sn1.parentAS0) ⇒ (sn ≠ sn1)) ∧

(sn∈bs.surroundingScopeUnit0.stateNameSet0)

The <state name>s in an <asterisk state list> shall be distinct and shall be contained in other <state list>s in the enclosing body or in the body of a supertype. See 11.2 *Concrete grammar* of [ITU‑T Z.103].

∀s1,s2∈<state>:

(∃ sn∈<name>: (sn∈s1.stateNameSet0 ∧ sn∈s2.stateNameSet0)) ⇒

( (s1.s-<state timer part> = undefined)

∨ (s2.s-<state timer part> = undefined)

∨ (s1.s-<state timer part> = s1.s-<state timer part>))

When two <state> items contain the same <state name>, a <state timer> shall not be specified for both <state> items or both <state> items shall specify the same <state timer>. See 11.2 *Concrete grammar* of [ITU‑T Z.102].

Transformations

< <state>(< s > ⁀ rest, triggers, timer) >

provided rest ≠ empty

=8=>

< mk-<state>(< s >, triggers, timer), mk-<state>(rest, triggers, timer) >

When the <state list> of a <state> contains more than one <state name>, a copy of that <state> is created for each such <state name>. Then the <state> is replaced by these copies. See 11.2 *Model* of [ITU‑T Z.103].

< <state>(< s >, triggers1, timer1) > ⁀rest ⁀ < <state>(< s >, triggers2, timer2) >

=15=>

rest⁀< mk-<state>(< s >, triggers1 ⁀ triggers2, if timer1 ≠ undefined then timer1 else timer2 endif) >

When two <state> items that each contain one <state name>, contain the same <state name>, these <state> items are concatenated into one <state> having that <state name> with the <state timer> specified for one (or both if it is the same) <state> items. See 11.2 *Model* of [ITU‑T Z.103].

<composite state list item>( name, nextstateparams)

provided mk-<identifier>(name.surroundingQualifier0, name).anonCompStateTypeName ≠ undefined

=12=>

mk-<typebased composite state>(

name,

nextstateparams,

<type expression>(

mk-<identifier>(name.surroundingQualifier0, // qualifier of type expression base type

mk-<identifier>(name.surroundingQualifier0, name).anonCompStateTypeName // name base type

), // base type identifer

empty // empty actual context parameters

) // type expression

) // typebased compsoite state

A <composite state list item> is a shorthand that is transformed to a <typebased composite state> that defines the name and an <composite state type<identifier> that references a <composite state type definition> transformed from the <composite state definition>. See 11.2 *Model* of [ITU‑T Z.102].

b=<state>(<asterisk state list>(exceptStates), triggers)

=15=>

mk-<state>(< s ∈ ((b.surroundingScopeUnit0.stateNameSet0) \ ( exceptStates.toSet )) >, triggers)

A <state> with an <asterisk state list> is transformed to a list of <state>s, one for each <state name> of the body in question, except for those <state name>s contained in the <asterisk state list>. See 11.2 *Model* of [ITU‑T Z.103].

st.stateTimerIdentifier

provided

st ∈ <state>

∧ st.s-<state timer part> ≠ undefined // expression or set clause

∧ st.stateTimerIdentifier = undefined

=8=>

(stateTimerIdentifier\{( st, undefined )})

∪ {( st,

if st.s-<state timer part>.s-<state timer> ∈ <expression>

then newName // expression – implicit name for parameterless timer definition

else st.s-<state timer part>.s-<state timer>.s-<identifier> // set clause – timer name

endif

)}

stm=<state timer>(exprn)

provided

exprn ∈ <Time<expression>

=8=>

if exprn ∈ <expression> then

mk-<state timer>( mk-<set clause>(exprn, st.stateTimerIdentifier,empty))

else // set clause

stm

endif

and

entities = parentAS0ofKind(stm,

<system type definition>

∪ <block type definition>

∪ <process type definition>).s-<agent structure>.s-<entity in agent>-seq

=>

entities ⁀ // add implicitly named parameterless timer definition to agent

mk-<timer definition> (< mk-<timer definition item>(st.stateTimerIdentifier,empty,undefined) >)

A <state timer> with **state** **timer** <Time expression> is equivalent to <state timer> with a <set clause> that has an implicitly named, parameterless timer definition, which is set to the <Time expression>. See 11.2 *Model* of [ITU‑T Z.102].

Mapping to abstract syntax

| <state>(< <basic state name>(name,) >, triggers, statetimerpart) then

mk-State-node(Mapping(name), // *State-name*

let TriggerSet = { Mapping(triggers[i]): i=1..triggers.length } in

{ ss ∈ TriggerSet: ss ∈ Save-signalset}, // Save-signalset

{ in ∈ TriggerSet: in ∈ Input-node-set }, // Input-node**-set**

// a basic state has (possibly empty) continuous signal or spontaneous transition sets

{ { cs ∈ TriggerSet: cs ∈ Continuous-signal}, // *Continuous-signal***-set**

{ sp ∈ TriggerSet: sp ∈ Spontaneous-transition} // *Spontaneous-transition***-set**

},

endlet // TriggerSet

Mapping(statetimerpart)

)

NOTE – A <state> with a <composite state list item> is transformed to a <state> with a <typebased composite state> before mapping takes place.

| <state>(< <typebased composite state>(name, basetypeid) >, triggers, statetimerpart) then

mk-State-node(Mapping(name),

let TriggerSet = { Mapping(triggers[i]): i=1.. triggers.length } in

{ ss ∈ TriggerSet: ss ∈ Save-signalset},

{ in ∈ TriggerSet: in ∈ Input-node},

// a state with a typebased composite state shall not have continuous signals or spontaneous transitions

{ Mapping(basetypeid),

{ cn ∈ TriggerSet: cn ∈ Connect-node}

},

Mapping(statetimerpart)

)

| stp = <state timer part>(\*, statetimer, transition) then

if statetimer ∈ <expression>

then // state timer - time expression

mk-State-timer(

Mapping(statetimer), // *Time-expression*

Mapping(parentAS0ofKind(stp, <state>).stateTimerIdentifier)), // *Timer-identifier*

empty, // empty *Expression list*

Mapping(transition)

)

else // state timer - set clause

mk-State-timer(

if statetimer.s-<expression> ≠ undefined // set clause has time expression

then Mapping(statetimer.s-<expression>) // *Time-expression*

else // no Time expresssion -use default value of timer

Mapping(

mk-<operator application>("+",

now,

statetimer.s-<identifier>.refersto0.s-<timer default initialization>

)

) // *Time-expression*

endif, // Time expression present/absent

Mapping(statetimer.s-<identifier>), // *Timer-identifier*

Mapping(statetimer.s-<expression>-seq) ,// set cause Expression list (possibly empty)

Mapping(transition)

)

endif // state timer kind

Auxiliary functions

The auxiliary function stateTimerIdentifier stores the name of a state timer for a state.

controlled stateTimerIdentifier: <state> → <identifier>

initially ∀ s ∈ <state>: s.stateTimerIdentifier = undefined

The function stateNameSet0 gets the set of <state name> items of a state, an agent definition, an agent type definition, a composite state, a composite state type definition or a procedure definition.

stateNameSet0(d: <state>∪<agent definition> ∪<agent type definition> ∪<composite state definition> ∪

<composite state type definition>∪<internal procedure definition>): <name>-set =def

if (d∈<state>) then

if d.s-<state list>∈<asterisk state list>then

d.surroundingScopeUnit0.stateNameSet0\{n∈<name>: n.parentAS0=d.s-<state list>}

else //d.s-<state list>∈{<basic state name>|<composite state list item>|<typebased composite state>}+

{n∈<name>: n.parentAS0=d.s-<state list>}

endif

else {n∈<name>: ∃s∈<state>: (s∈d.stateSet0)∧(n∈s.stateNameSet0)}

endif

#### F2.2.8.3 Input

Abstract syntax

Input-node :: [ *Priority-name* ]

Signal-identifier [ Gate-identifier ]

[ Provided-expression ]

[ In-choice ]

[ Variable-identifier ]\*

Transition

In-choice :: Variable-identifier

Conditions on abstract syntax

∀in ∈ Input-node: ∀sd ∈ Signal-definition:

sd= getEntityDefinition1(in.**s**-Signal-identifier, signal)⇒

(in.**s**-Variable-identifier-seq.length=sd.**s**-Sort-reference-identifier-seq.length) ∧

(∀i ∈ 1..in.**s**-Variable-identifier.length:

∃vd ∈ Variable-definition: vd=getEntityDefinition1(in.**s**-Variable-identifier[i],variable)∧

isCompatibleTo1(vd.**s**-Sort-reference-identifier, sd.**s**-Sort-reference-identifier[i]))

The length of the list of optional Variable-identifiers must be the same as the number of items in the Sort-reference-identifier list in the Signal-definition denoted by the Signal-identifier and the sorts of the variables must correspond by position to the sorts of the data items that can be carried by the signal.

Concrete syntax

<input part> ::

[<virtuality>] <input list> [<enabling condition>] <transition>

<input list> ::

{ <stimulus> [ <in choice> ] [ <priority clause> ]}+

| <asterisk input list> [ <in choice> ] [ <priority clause> ]

| <encoded input>

<stimulus> :: <signal list item> [<variable>]\* [<via path>]

<asterisk input list> :: {}

<via path> :: <channel<identifier> | <gate<identifier>

<in choice> :: <choice<variable> | <signal expression>

<encoded input> :: <variable> [ <encoding path> ]

Further study is needed for <encoded input>.

Further study is needed for <in choice>.

Conditions on concrete syntax

∀s∈<state>: |s.asteriskInputListSet0| ≤ 1

A <state> shall contain at most one <asterisk input list>. See clause 11.3 *Concrete* *grammar* of [ITU‑T Z.103].

∀s∈<state>: (s.asteriskInputListSet0=∅) ∨ (s.asteriskSaveListSet0 = ∅)

A <state> shall not contain both <asterisk input list> and <asterisk save list>. See clause 11.3 *Concrete* *grammar* of [ITU‑T Z.103].

∀stim∈(<input part> ∪ <priority input> ): ∀sli∈<signal list item>:

isAncestorAS0(stim,sli)⇒

(let idKind = sli.**s**-<identifier>.idKind0in

(idKind≠ remote variable)∧

(idKind = remote procedure∨ idKind = signallist⇒

sli.parentAS0.**s**-<variable>-seq=empty)

endlet)

A <signal list item> of a <stimulus> (of an <input list> or <priority input>) shall not denote a <remote variable identifier> and if it denotes a <remote procedure identifier> or a <signal list identifier>, the <stimulus> parameters (including the parentheses) shall be omitted. See clause 10.4 *Concrete* *grammar* of [ITU‑T Z.102].

∀v in <stimulus>.**s**-<variable>-seq: ¬(isGlobalBlockVar0(v))

A <variable> of a <stimulus> shall not be a global variable of a system (type) or block (type) except if the <stimulus> is within the state machine actions of system (type) or block (type). See clause 11.3 *Concrete* *grammar* of [ITU‑T Z.101].

∀chid ∈ { id = stim.**s**-<via path>.**s**-<identifier> :

stim ∈ ( <stimulus> ∪ <save item> ) ∧

id.refersto0 ∈ <channel definition> ∧ // not a gate definition

stim.surroundingScopeUnit0 ∈ <composite state type definition> } // only when in compsosite state type

:

( ∃!gid ∈

( reachableSmGates(chid.refersto0.s1-<channel path>.s2-<channel endpoint>.s-<gate>,

chid.surroundingScopeUnit0)

∪

if chid.refersto0.s2-<channel path> ≠ undefined

then reachableSmGates(chid.refersto0.s2-<channel path>.s2-<channel endpoint>.s-<gate>,

chid.surroundingScopeUnit0)

else ∅

endif

)

:

( gid.refersto0.s-<gate constraint>.direction0 = in ∧

chid.parentAS0.parentAS0 in gid.refersto0.s-<gate constraint>.s1-<signal list>

)

)

The <channel<identifier> for a <via path> of a <stimulus> or <save item> in a composite state type (that is, after application of the models for agent definition, agent structure with an interaction and channel to channel connection) shall identify a channel such that the enclosing state machine of the via path is reachable from the channel with the signal given in the stimulus or save through exactly one gate of the state machine. See clause 11.3 *Concrete* *grammar* of [ITU‑T Z.103].

∀chid ∈ { id = cc.**s**-<via path>.**s**-<identifier> :

cc ∈ <comunication constraints> ∧

id.refersto0 ∈ <channel definition> ∧ // not a gate definition

cc.surroundingScopeUnit0 ∈ <composite state type definition> } // only when in compsosite state type

:

( ∃!gid ∈

( reachableSmGates(chid.refersto0.s1-<channel path>.s2-<channel endpoint>.s-<gate>,

chid.surroundingScopeUnit0)

∪

if chid.refersto0.s2-<channel path> ≠ undefined

then reachableSmGates(chid.refersto0.s2-<channel path>.s2-<channel endpoint>.s-<gate>,

chid.surroundingScopeUnit0)

else ∅

endif

)

:

( gid.refersto0.s-<gate constraint>.direction0 = in ∧

chid.parentAS0.parentAS0 in gid.refersto0.s-<gate constraint>.s1-<signal list>

)

)

A <channel<identifier> for a <via path> of <communication constraints> output in a composite state type (that is, after application of the models for agent definition, agent structure with an interaction, channel to channel connection, remote procedure and import expressions) shall identify a channel such that the enclosing state machine of the via path is reachable from the channel with the signal given in the stimulus or save through exactly one gate of the state machine. See clause 11.3 *Concrete* *grammar* of [ITU‑T Z.103].

∀chid ∈ { id ∈ <identifier> : id.parentAS0 ∈ <via path> ∧ id.refersto0 ∈ <channel definition> }

:

(let ch = chid.refersto0 in

(ch.s2-<channel path> ≠ undefined ) // bidirectional

⇒

( ch.s1-<channel path>.s2-<channel endpoint>.**s**-<identifier>.refersto0 ∉ (<state machine> ∪ **this**) ) ∨

( ch.s1-<channel path>.s2-<channel endpoint> ≠ ch.s2-<channel path>.s2-<channel endpoint> )

endlet) // ch

If a <via path> has a <channel identifier> this shall not be a bidirectional channel with both ends connected to the same state machine. See clause 11.3 *Concrete* *grammar* of [ITU‑T Z.103].

∀v ∈ <encoded input>.**s**-<variable>: ¬(isGlobalBlockVar0(v))

A <variable> of an <encoded input> shall not be a global variable of a system (type) or block (type) except if the <encoded input> is within the state machine actions of system (type) or block (type). See clause 11.3 *Concrete* *grammar* of [ITU‑T Z.104].

∀v ∈ <in choice>.**s**-<variable>: ¬(isGlobalBlockVar0(v))

A <variable> of an <in choice> shall not be a global variable of a system (type) or block (type) except if the <in choice> is within the state machine actions of system (type) or block (type). See clause 11.3 *Concrete* *grammar* of [ITU‑T Z.104].

Transformations

< <stimulus>(<signal list item>(**signallist**, id), varlist, path) >

=8=> < mk-<stimulus>(sig, varlist, path) | sig in id.refersto0.s-<signal list item>-seq >

A <stimulus> whose <signal list item> is an <interface<identifier> is derived syntax for a list of <stimulus> items that replaces the <interface<identifier> in the enclosing <input list> or <priority input list>. In this list, there is a one to one correspondence between the <stimulus>s and the members of the signal list. See clause 11.3 *Model* of [ITU‑T Z.103].

<input part>(virt, <input list>(< (stim, choice, pri) > ⁀ rest), cond, trans) provided rest ≠ empty

=8=>

mk-<input part>(virt, mk-<input list>(< (stim, choice, pri) >), cond, trans) ⁀

mk-<input part>(virt, mk-<input list>(rest), cond, trans)

When the <stimulus> list of an <input part> contains more than one <stimulus>, a copy of the <input part> is created for each such <stimulus>. Then the <input part> is replaced by these copies. See clause 11.3 *Model* of [ITU‑T Z.103].

<input part>(virt, <input list>(< (stim, choice, pri) >), \*, trans) provided pri ≠ undefined

=8=>

mk-<priority input>(virt, mk-<priority input list>(< (stim, choice, pri) >), trans)

In an <input part> with one <stimulus>, if the <stimulus> is followed by a <priority clause>, the input symbol is replaced by a <priority input> containing the <stimulus> followed by the <priority clause>. See clause 11.3 *Model* of [ITU‑T Z.103].

v.anonCompVar

provided

v ∈ (<indexed variable> ∪ <field variable>) ∧

(∃ ip ∈ <input part>:(v in stm.s-<variable>-seq ∧ stm ∈ <stimulus> ∧ stm in ip.s-<input list>.s-implicit )) ∧

v.anonCompVar = undefined

=8=>

(anonCompVar\{( v, undefined )} ) ∪ {( v, newName)}

ip = <input part>(virt, <input list>(< (<stimulus>(item, vars, path), choice, pri) >), cond, trans)

provided

{ v ∈ vars.toSet: v ∈ (<indexed variable> ∪ <field variable>) ∧ v.anonCompVar ≠ undefined } ≠ ∅

// one or more of the stimulus variable items of a <stimulus> is an indexed/field variable

// and anonymous name allocated for indexed/field variable

=8=>

let newvars =

< (if v ∈ (<indexed variable> ∪ <field variable>) then v.anonCompVar else v endif) : v in vars >

in

mk-<input part>(

virt,

mk-<input list>(< (mk-<stimulus>(item, newvars, path), choice, pri) >),

cond,

if trans ∈ <transition action items>

then // put assigns from newvars to vars before actions and terminator

mk-<transition action items>(

< assignNewvarsToExtendedVars0(vars, newvars) > ⁀ trans.s-<action>-seq,

trans.s-<terminator>) // transition action items

else // transition was just a terminator, // put assigns from newvars to vars before terminator

mk-<transition action items>( < assignNewvarsToExtendedVars0(vars, newvars) >, trans)

endif

) // <input part>

endlet // newvars

and

items = parentAS0ofKind(ip, ScopeUnit0).getEntities0

=>

// add implicitly defined variable definitions to the surrounding scope

items ⁀ defineNewvars0(vars,

< (if v ∈ (<indexed variable> ∪ <field variable>) then v.anonCompVar else v endif) : v in vars >

) // defineNewvars0

When one or more of the stimulus variable items of a <stimulus> is an <extended variable> (an <indexed variable> or <field variable>), then each <extended variable> is replaced by a unique, new, anonymous implicitly declared <variable<identifier> of the same sort as the original <extended variable>. Directly following the input area (<input area> or <priority input area>), a <task area> is inserted which in its <task body> contains an <assignment statement> for each of the   
<extended variable> items, assigning the result of the corresponding new variable to the   
<extended variable>. The results are assigned in the order from left to right of the <extended variable> list. This <task area> becomes the first <action area> in the <transition area> of the input area. See clause 11.3 *Model* of [ITU‑T Z.103].

< ip = <input part>( virt, <input list>(a = <asterisk input list>, choice, pri), cond, trans) >

provided

parentAS0ofKind(ip,<state>).s-<state list> ∉ <asterisk state list> ∧

parentAS0ofKind(ip,<state>).s-<state list>.length= 1

=8=>

<

if pri = undefined

then

mk-<input part>(

virt,

mk-<input list>(

< mk-<stimulus>(item, undefined, undefined), choice, undefined)

> ), //input list

cond,

trans

) // input part

else

mk-<priority input>(

virt,

mk-<priority input list>(

< mk-<priority stimulus>(item, undefined, undefined), choice, pri)

> ), // priority input list

trans

) // priority input

endif

: item ∈ (explicitValidInputSigs0(parentAS0ofKind(ip,<agent type definition>)) \

{ sig ∈ <signal list item>:

parentAS0ofKind(sig,<state>) = st ∧

parentAS0ofKind(ip,<state>).s-<state list>.head in st.s-<state list> // same state

} // set of signals in other inputs or saves of state

) //

> // replacement list of input parts

An <asterisk input list> is transformed to a list of <input part>s, one for each member of the complete valid input signal set of the enclosing <agent definition>, except for <signal identifier>s of implicit input signals introduced by the concepts in clauses 10.5 and 10.6 of [ITU-T Z.102] and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>.

vp = <via path>( chid)

provided

chid.refersto0 ∈ <channel definition> ∧ // not gate id

vp.parentAS0 ∈ ( <stimulus> ∪ <save item> ) ∧ // is a via path of a stimulus/save

surroundingScopeUnit0(vp) ∈ <composite state type definition>) ∧

// agent definition and agent type definition have transformed, therefore vp is in a composite state type defn.

(

let chEndPt12 = chid.refersto0.s1-<channel path>.s2-<channel endpoint> in

let chEndPt22 = chid.refersto0.s1-<channel path>.s1-<channel endpoint> in

let sigidIn = vp.parentAS0.s-<signal list item> in //signal identifier of stimulus or save

( sigidIn in chEndPt12.s-<signal list> ∧ // signal of stimulus/save carried by the channel - forward path

chEndPt12.s-<gate> ≠ undefined ∧ // endpoint has gate

chEndPt12.s-<gate>.refersto0.s-<gate constraint>.direction0 = in ∧ //with in signal list

sigidIn in chEndPt12.refersto0.s-<gate constraint>.s1-<signal list> ∧ // sig carried by gate

chEndPt12.s-<gate>.refersto0.surroundingScopeUnit0 = vp.surroundingScopeUnit0(vp) //same scope

)

∨ // or consider reverse path

( chid.refersto0.s2-<channel path> ≠ undefined ∧ // reverse path defined

sigidIn in chEndPt22.s-<signal list> ∧ // signal of stimulus/save carried by the channel - revese path

chEndPt22.s-<gate> ≠ undefined ∧ // endpoint has gate

chEndPt22.s-<gate>.refersto0.s-<gate constraint>.direction0 = in ∧ //with in signal list

sigidIn in chEndPt22.s-<gate>.refersto0.s-<gate constraint>.s1-<signal list> ∧ // sig carried by gate

chEndPt22.s-<gate>.refersto0.surroundingScopeUnit0 = vp.surroundingScopeUnit0 // same scope

)

endlet // sigidIn

endlet // chEndPt22

endlet // chEndPt12

)

=12=>

<via path>(

take( // static condition requires there is only one reachable gate in the set

reachableSmGates(chid.refersto0.s1-<channel path>.s2-<channel endpoint>.s-<gate>,

vp.surroundingScopeUnit0)

∪

if chid.refersto0.s2-<channel path> ≠ undefined

then reachableSmGates(chid.refersto0.s2-<channel path>.s2-<channel endpoint>.s-<gate>,

vp.surroundingScopeUnit0)

else ∅

endif

) ) // take, via path

A <via path> of a <stimulus> with a <channel identifier> is transformed to a <via path> with a (possibly anonymous) <gate identifier> for the gate where the channel connects (directly or indirectly) to the enclosing state machine. See clause 11.3 *Model* of [ITU‑T Z.103].

vp = <via path>( chid)

provided

chid.refersto0 ∈ <channel definition> ∧ // not gate id

vp.parentAS0 ∈ <output body> ∧ // is a via path of an output body

// remote procedure calls and import expressions transformed

surroundingScopeUnit0(vp) ∈ <composite state type definition>) ∧

// agent definition and agent type definition have transformed, therefore vp is in a composite state type defn.

(

let chEndPt12 = chid.refersto0.s1-<channel path>.s2-<channel endpoint> in

let chEndPt22 = chid.refersto0.s1-<channel path>.s1-<channel endpoint> in

let sigidOut = vp.parentAS0.parentAS0.s-<output body item>-seq[1] in // signal identifier output

( sigidOut in chEndPt12.s-<signal list> ∧ // signal of output carried by the channel - forward path

chEndPt12.s-<gate> ≠ undefined ∧ // endpoint has gate

chEndPt12.s-<gate>.refersto0.s-<gate constraint>.direction0 = in ∧ //with in signal list

sigidOut in chEndPt12.s-<gate>.refersto0.s-<gate constraint>.s1-<signal list> ∧ // carried by gate

chEndPt12.s-<gate>.refersto0.surroundingScopeUnit0 = vp.surroundingScopeUnit0(vp) //same scope

)

∨ // or consider reverse path

( chid.refersto0.s2-<channel path> ≠ undefined ∧ // reverse path defined

sigidOut in chEndPt22.s-<signal list> // signal of output carried by the channel - revese path

chEndPt22.s-<gate> ≠ undefined ∧ // endpoint has gate

chEndPt22.s-<gate>.refersto0.s-<gate constraint>.direction0 = in ∧ //with in signal list

sigidOut in chEndPt22.s-<gate>.refersto0.s-<gate constraint>.s1-<signal list> ∧ // carried by gate

chEndPt22.s-<gate>.refersto0.surroundingScopeUnit0 = vp.surroundingScopeUnit0 // same scope

)

endlet // sigidOut

endlet // chEndPt22

endlet // chEndPt12

)

=12=>

<via path>(

take( // static condition requires there is only one reachable gate in the set

reachableSmGates(chid.refersto0.s1-<channel path>.s2-<channel endpoint>.s-<gate>,

vp.surroundingScopeUnit0)

∪

if chid.refersto0.s2-<channel path> ≠ undefined

then reachableSmGates(chid.refersto0.s2-<channel path>.s2-<channel endpoint>.s-<gate>,

vp.surroundingScopeUnit0)

else ∅

endif

) ) // take, via path

A <channel<identifier> as a <via path> of <communication constraints> is transformed to the (possibly anonymous) <gate<identifier> for the gate of the channel that connects (directly or indirectly) to the enclosing state machine. See clause 11.3 *Model* of [ITU‑T Z.103].

Mapping to abstract syntax

| <input part>(\*, < <stimulus>(item, vars, vp), choice, undefined >, cond, trans) then

mk-Input-node(undefined, // no priority name – see mapping of priority input

Mapping(item), // signal identifier

Mapping(vp), // optional gate identifier

Mapping(cond), // enabling condition -> optional provided expression

Mapping(choice), // optional in choice

< Mapping(vars[i]) : i ∈ 1..vars.length >, // list - each element is a Varible-identifier or undefined

Mapping(trans)) // transition

| <via path>( id) then Mapping(id)

| <in choice>( var ) then Mapping(var)

Auxiliary functions

Find the explicit valid input signal set of an <agent type definition> (that is, a system type definition, a block type definition or process type definition). The explicit valid input signal set is the complete valid input signal set (see clause 9 *Semantics* of [ITU-T Z.101]) excluding implicit signals introduced and therefore includes:

1. the set of signals in all channels or gates leading to the state machine of the agent;
2. the valid input signal set defined explicitly for the agent;
3. the valid input signal set defined explicitly for the state machine of the agent;
4. the timer signals.

explicitValidInputSigs0(atd:<agent type definition>):<signal list item>-set =def

let stateMachine = atd.s-<agent structure>.s-<interaction>.s-<typebased composite state> in

let stateMachineId = mk-<identifier>(atd.fullQualifierWithin0, stateMachine.s-<name>) in

let chanSigs ={sig: sig in chPath.s-<signal list item>-seq ∧

chPath ∈ <channel path> ∧ chPath.s2-<channel endpoint>= stateMachineId} in

let entitiesInCompType =

if stateMachine.s-<type expression>.refersto0.s-implicit ∈ <composite state type graph>

then stateMachine.s-<type expression>.refersto0.s-<composite state type graph>.

s-<composite state structure>.s-<entity in composite state>-seq

else stateMachine.s-<type expression>.refersto0.s-<state aggregationtype>.

s-<aggregation structure>.s-<entity in state aggregation>-seq

endif

in

**let** gateSigs ={sig: sig **in** gate.**s1-**<signal list item>**-seq** ∧ gate.**s-**in ≠ undefined ∧

gate ∈ <gate in definition> ∧ gate **in** entitiesInCompType}

in

let validSigInputSetAgent ={visa: visa in viss.s-<signal list item>-seq ∧

viss ∈ <valid input signal set> ∧ viss in atd.s-<agent structure>.s-<entity in agent>-seq}

in

let validSigInputSetComp ={visc: visc in viss.s-<signal list item>-seq ∧

viss ∈ <valid input signal set> ∧ viss in entitiesInCompType}

in

chanSigs ∪ gateSigs ∪ validSigInputSetAgent ∪ validSigInputSetComp

endlet // validSigInputSetComp

endlet // validSigInputSetAgent

endlet // gateSigs

endlet // entitiesInCompType

endlet // chanSigs

endlet // stateMachineId

endlet // stateMachine

Make an action list that is assignments of new variables to extended variables.

assignNewvarsToExtendedVars0(varsl:<variable>\*, newvarsl:<variable>\*):<action>\*=def

if varsl =empty then empty

else

let v = varsl.head in

let newv = newvarsl.head in

if v ∈ (<indexed variable> ∪ <field variable>)

then

// create an assignment of the result of the corresponding newv to the extended variable v

<action>(undefined,<task>(<assignment>(v, newv),undefined))⁀

assignNewvarsToExtendedVars0(varsl.tail, newvarsl.tail)

else assignNewvarsToExtendedVars0(varsl.tail, newvarsl.tail)

endif

endlet // newv

endlet // v

endif

Definitions for new variables that replace extended variables in a stimulus.

defineNewvars0(varsl:<variable>\*, newvarsl:<variable>\*):<variable definition>\*=def

if varsl =empty

then empty

else

let v = varsl.head in

let newv = newvarsl.head in

if v ∈ (<indexed variable> ∪ <field variable>)

then

// create a definition of the newv that corresponds to the extended variable v

<variable definition>( <variables of sort>(PART, newv, getVariableSort0(v)))⁀

defineNewvars0 (varsl.tail, newvarsl.tail)

else defineNewvars0 (varsl.tail, newvarsl.tail)

endif

endlet // newv

endlet // v

endif

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

asteriskInputListSet0(s: <state>): <asterisk input list>-set =def

{ail∈<asterisk input list>: isAncestorAS0(s,ail)}

The function reachableSmGates is list of destination gates in instances of the composite state type reachable from the given gate. This function should be used only after any agent definition has been replaced by a typebased agent definition, and any agent structure with an agent body has been replaced by an agent structure with an interaction.

reachableSmGates(gid: <identifier>, cstd : <composite state type definition>):<identifier>-set =def

if gid.refersto0 ∉ <gate definition>

then ∅

elseif gid.refersto0.surroundingScopeUnit0 = cstd

then { gid }

elseif gid.refersto0.surroundingScopeUnit0 ∈ <composite state type definition> // but not cstd

then ∅

elseif gid.refersto0.surroundingScopeUnit0 ∈

( <block type definition> ∪ <process type definition> ∪ <system type definition> )

then

**U**{ if chdef.s1-<channel path>.s2-<channel endpoint>.s-<gate> = gid

then reachableSmGates(chdef.s1-<channel path>.s1-<channel endpoint>.s-<gate>, cstd)

else reachableSmGates(chdef.s1-<channel path>.s2-<channel endpoint>.s-<gate>, cstd)

endif // find reachable state machine gates for the other channel endpoint

: gid.refersto0.surroundingScopeUnit0 = chdef.surroundingScopeUnit0 ∧ // same scope as gate def

( chdef.s1-<channel path>.s1-<channel endpoint>.s-<gate> = gid // gate originates ch fwd path

∨ chdef.s1-<channel path>.s2-<channel endpoint>.s-<gate> = gid // or, gate terminates ch fwd path

) // rev path – if it exists – interchanges the gates

} // take the distributed union of the set of sets

else ∅

endif

The function anonCompVar is a mapping from an indexed or extended variable to an anonymous name, used for a temporary variable to store a value to assign to the indexed or extended variable.

controlled anonCompVar: (<indexed variable> ∪ <field variable>) → <name>

initially ∀ v ∈ (<indexed variable> ∪ <field variable>) : v.anonCompVar = undefined

#### F2.2.8.4 Priority input

Concrete syntax

<priority input> ::

[<virtuality>] <priority input list> <transition>

<priority input list> =

<priority stimulus>+

| <asterisk input list> [ <in choice> ] [ <priority clause> ]

<priority stimulus> :: <stimulus> [ <in choice> ] [ <priority clause> ]

<priority clause> :: [ <priority name> ]

<priority name> = <Natural<name>

NOTE – In [ITU‑T Z.103] clause 11.4 Priority Input, *Concrete grammar* <priority name> is defined as a <integer name>, which is equivalent to AS0 <Natural<name>, because AS0 <name> includes the SDL‑2010 lexical units <name>, <integer name> and <real name>.

Transformations

pc = <priority clause>(pname)

provided

pname = undefined ∧ // no name give, just the keyword **priority**

parentAS0ofKind(pc,<state>).s-<state list> ∉ <asterisk state list> ∧ // not asterisk state list

parentAS0ofKind(pc,<state>).s-<state list>.length= 1 // only consider when state list reduced to one state

=8=>

let piSet = { pi : pi ∈ <priority input> ∧

parentAS0ofKind(pc,<state>).s-<state list>.head in

parentAS0ofKind(pi,<state>).s-<state list> }

in

let priSet = { ps.s-<priority clause>.s-<priority name>.s-IntToken0.*intTokenToNat*:

pi ∈ piSet ∧ ps in pi.s-<priority stimulus>-seq ∧ ps.s-<priority clause> ≠ *undefined*}

in

let highest = take({n ∈ Nat

: (¬∃ m )( m ∈ priSet ∧ m > n) // not exists m

}) // set for n, take

in

mk-<priority clause>(mk-<priority name>(natToIntToken(highest+1)))

endlet // highest

endlet // priSet

endlet // piSet

A <priority clause> without a <priority name> is transformed into a <priority clause> that is **priority** n, where n is one greater than the highest explicit <priority name> for a <priority stimulus> of the same state. The same value is used for all such transformations of the state, so that each priority stimulus without an explicit <priority name> has the same priority. See clause 11.4 *Model* of [ITU‑T Z.103].

<priority input>(virt, < stim > ⁀ rest, trans) provided rest ≠ empty

=8=> mk-<priority input>(virt, < stim >, trans) ⁀ mk-<priority input>(virt, rest, trans)

When the <priority stimulus> list of a <priority input> contains more than one <priority stimulus>, a copy of the <priority input> is created for each such <priority stimulus>. Then the <priority input> is replaced by these copies. See clause 11.4 *Model* of [ITU‑T Z.103].

< pi = <priority input>( virt, <priority input list>(a = <asterisk input list>, choice, pri), trans) >

provided

parentAS0ofKind(pi,<state>).s-<state list> ∉ <asterisk state list> ∧

parentAS0ofKind(pi,<state>).s-<state list>.length= 1

=8=>

<

mk-<priority input>(

virt,

mk-<priority input list>(

< mk-<priority stimulus>(item, undefined, undefined), choice, pri)

> ), // priority input list

cond,

trans

) // priority input

: item ∈ ( explicitValidInputSigs0(parentAS0ofKind(ip,<agent type definition>)) \

{ sig ∈ <signal list item>:

parentAS0ofKind(sig,<state>) = st ∧

parentAS0ofKind(pi,<state>).s-<state list>.head in st.s-<state list> // same state

} // set of signals in other inputs or saves of state

) //

> // replacement list of input parts

An <asterisk input list> in a <priority input list> is transformed to a <priority stimulus> list in the same way as an <asterisk input list> in an <input list> is transformed to a <stimulus> list.

Mapping to abstract syntax

| <priority input>(\*, <priority input list>(< <stimulus>(item, vars, vp), choice, pri >), trans) then

mk-Input-node(natToIntToken(pri.s-Token),

Mapping(item), // signal identifier

Mapping(vp), // optional gate identifier

undefined, // no enabling condition in a priority input

Mapping(choice), // optional in choice

< Mapping(vars[i]) : i ∈ 1..vars.length >, // list - each element is a Varible-identifier or undefined

Mapping(trans)) // transition

#### F2.2.8.5 Continuous signal

Abstract syntax

Continuous-signal :: Continuous-expression

[ Priority-name ]

Transition

Continuous-expression = Boolean-expression

Priority-name = Nat

Concrete syntax

<continuous signal> ::

[<virtuality>] <continuous expression> [<priority name>] <transition>

<continuous expression> = <Boolean<expression>

Mapping to abstract syntax

| <continuous signal>(\*, expr, prio, trans) then

mk-Continuous-signal(Mapping(expr), Mapping(prio), Mapping(trans))

#### F2.2.8.6 Enabling condition

Abstract syntax

Provided-expression = Boolean-expression

Boolean-expression = Expression

Concrete syntax

<enabling condition> = <provided expression>

<provided expression>:: <Boolean<expression>

Mapping to abstract syntax

| <provided expression>(expr) then Mapping(expr)

#### F2.2.8.7 Save

Abstract syntax

Save-signalset :: Save-item-set

Save-item = Signal-identifier [ Gate-identifier ]

Concrete syntax

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

<save list> :: <save item>+ | <asterisk save list>

<asterisk save list> :: ()

<save item> :: <signal list item> [<via path>]

Conditions on concrete syntax

∀s∈<state>: |s.asteriskSaveListSet0| ≤ 1

A <state> is allowed to contain at most one <asterisk save list>. See clause 11.7 *Concrete grammar* of [ITU‑T Z.103].

∀s∈<state>: (s.asteriskInputListSet0= ∅) ∨ (s.asteriskSaveListSet0 = ∅)

A <state> shall not contain both <asterisk input list> and <asterisk save list>. See clause 11.3 *Concrete grammar* of [ITU‑T Z.103].

Transformations

sp = <save part>( virt, <save list>(a = <asterisk save list>)) >

provided

parentAS0ofKind(sp,<state>).s-<state list> ∉ <asterisk state list> ∧

parentAS0ofKind(sp,<state>).s-<state list>.length= 1

=8=>

mk-<save part>(

virt,

mk-<save list>(<

mk-<save item>( item , undefined) // signal list item, omitted via path

: item ∈ ( explicitValidInputSigs0(parentAS0ofKind(sp,<agent type definition>)) \

{ sig ∈ <signal list item>:

parentAS0ofKind(sig,<state>) = st ∧

parentAS0ofKind(sp,<state>).s-<state list>.head in st.s-<state list> // same state

} // set of signals in other inputs or saves of state

>) // list of save items, save list

) // revised save part

An <asterisk save list> is transformed to a list of <stimulus>s containing the complete valid input signal set of the enclosing <agent definition>, except for <signal identifier>s of implicit input signals introduced by the concepts in clauses 10.5 and 10.6 of [ITU-T Z.102] and for <signal identifier>s contained in the other <input list>s and <save list>s of the <state>.

Mapping to abstract syntax

| <save part>(\*, svlist ) then

mk-Save-signalset({ Mapping(svlist.s-<save item>-seq[i]) : i = 1..svlist.s-<save item>-seq.length })

| <save item>( item, via) then < Mapping(item.s-<signal list item>), Mapping(via) >

Auxiliary functions

Get the set of <asterisk save list> for s <state>.

asteriskSaveListSet0(s: <state>): <asterisk save list>-set =def

{asl∈<asterisk save list>: isAncestorAS0(s,asl)}

#### F2.2.8.8 Implicit transition

Transformations

*The following statement is handled by the dynamic semantics.*

Any signal not handled by an explicit input or save is consumed by an implicit transition (a transition of an implicit <input association area>) without a change of state.

#### F2.2.8.9 Spontaneous transition

Abstract syntax

Spontaneous-transition :: [ Provided-expression ]

Transition

Concrete syntax

<spontaneous transition> ::

[<virtuality>] [<enabling condition>] <transition>

Mapping to abstract syntax

| <spontaneous transition>(\*, cond, trans) then

mk-Spontaneous-transition(Mapping(cond), Mapping(trans))

#### F2.2.8.10 Label

Abstract syntax

Free-action :: Connector-name Transition

Concrete syntax

<label> :: <connector name>

<connector name> = <name>

NOTE – An AS0 <connector name> is defined to be the same as an AS0 <name>, therefore the terms <connector name> and <connector<name> are interchangeable. In [ITU‑T Z.101] a distinction is made between a <name> and an <integer name>, and a <connector name> can be either a <name> or an <integer name>.

<free action> :: <transition> [ <label> ]

Conditions on concrete syntax

∀b∈<composite state body>∪<operation body>∪<procedure body>∪<agent body>: ∀l, l1∈<label>:

(isAncestorAS0(b,l) ∧ isAncestorAS0 (b,l1) ∧ (l ≠ l1) ⇒ (l.**s**-<name> ≠ l1.**s**-<name>))

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

∀fa∈<free action>:

if fa.s-<transition action items> ≠ undefined ∧ fa.s-<transition action items>.**s**-<action>-seq ≠ empty

then fa.s-<transition action items>.**s**-<action>-seq.head.**s**-<label> ≠ undefined ∧

fa.s-<transition action items>.**s**-<action>-seq.head.**s**-<label> = fa.**s**-<label>

else fa.s-<terminator>≠ undefined ∧ fa.s-<terminator>.**s**-<label> ≠ undefined ∧

fa.s-<terminator>.**s**-<label> = fa.**s**-<label>

endif

If the <transition string> of the <transition> in <free action> is non-empty, the first <action> shall have a <label> otherwise the <terminator> shall have a <label>. If present, the <label> ending the <free action> shall be the same as this <label>.

Auxiliary functions

The function getLabel extracts the first label from the transition.

getLabel(t: <transition>): <label> =def

if t.s-<action> = empty then t.s-<terminator>.s-<label>

else t.s-<action>.head.s-<label>

endif

Transformations

<transition action items>( actions1 ⁀ < la=<action>(l, \*) > ⁀ actions2, term=<terminator>(tl, tn ))

provided actions1 ≠ empty ∧ l ≠ undefined

=5=>

mk-<transition action items>( actions1 ⁀ mk-<terminator>(undefined, mk-<join>(l.s-<name>))) ⁀

if actions2 ≠ empty then

mk-<free action>( mk-<transition action items> (< la > ⁀ actions2, term))

elseif term ≠ undefined then

if tl ≠ undefined then

mk-<free action>(mk-<transition action items>(< la > ⁀

mk-<terminator>(undefined, mk-<join>(tl.s-<name>)))) ⁀

mk-<free action>( term )

else

mk-<free action>( mk-<transition action items>(< la > ⁀ mk-<terminator>( l, tn)))

endif

endif

<transition action items>( actions, term(tl, tn ))

provided tl ≠ undefined

=5=>

mk-<transition action items> ( actions ⁀ < mk-<terminator>(undefined, mk-<join>( tl.s-<name>)) >) ⁀

mk-<free action>( term)

If a <label> is not the first label of a <transition string>, the <transition string> is split into two parts. All <action>s preceding the <label> are preserved in the original transition, which is terminated with a <join> to the <label>. All <action>s following <label> are copied to a new <free action>, which starts with the <label>. If the <transition string> is followed by a <terminator> with a label, the new <free action> is terminated with a <join> to the label of the <terminator> and another <free action> is made that contains the labelled <terminator>. If the <transition string> is followed by a <terminator> without a label, the new <free action> is terminated with the unlabelled <terminator>. If a <transition string> without any labels is followed by a labelled <terminator>, a <free action> is made that contains the labelled <terminator> and the <terminator> is replaced by a <join> to the label of the <terminator>. See clause 5.7.9 Model of [ITU‑T Z.106].

Mapping to abstract syntax

| <free action>(trans) then

mk-Free-action(Mapping(getLabel(trans)), Mapping(trans))

#### F2.2.8.11 State machine and composite state

Abstract syntax

Composite-state-formal-parameter = Agent-formal-parameter

State-entry-point-definition = Name

State-exit-point-definition = Name

Exit-procedure-definition = Procedure-definition

Entry-procedure-definition = Procedure-definition

Named-start-node :: State-entry-point-name

Transition

Conditions on abstract syntax

(∀d ∈ Entry-procedure-definition:

(d.**s**-Procedure-name= "entry")∧(d.formalParameterList1 = empty)∧(d.stateNodeSet1=∅))

(∀d ∈ Exit-procedure-definition:

(d.**s**-Procedure-name="exit")∧(d.formalParameterList1 = empty)∧(d.stateNodeSet1=∅))

*Entry-procedure-definition* represents a procedure with the name entry. *Exit-procedure-definition* represents a procedure with the name exit. These procedures may not have parameters, and may only contain a single transition.

Concrete syntax

<composite state definition> ::

<composite state graph> | <state aggregation>

Transformations

anonCompStateTypeName(fullIdentifier0(

cs = <composite state definition>(graph)

) ) // fullIdentifier0, anonStateTypeName

provided

cs.fullIdentifier0.anonCompStateTypeName = undefined

=12=>

(anonCompStateTypeName \{( cs.fullIdentifier0, undefined )} ) ∪ {( cs.fullIdentifier0, newName )}

and // <state> with a <state list> item that is a <composite state list item> - corr. ref. case (a)

st = <state>(statellist,triggers,timer)

provided

st.surroundingScopeUnit0 = cs.surroundingScopeUnit0 ∧

( ∃ i ∈ 1..statelist.length :

statelist[i].s-<composite state name> = graph.s-<composite state heading>.s-<composite state name> )

=>

let name = graph.s-<composite state heading>.s-<composite state name> in

mk-<state>(

< if statelist[i].s-<composite state name> = name

then // modify this element

mk-<typebased composite state>(name,

statelist[i].s-<nextstate parameters>,

mk-<type expression>(

mk-<identifier>(cs.surroundingQualifier0,

cs.fullIdentifier0.anonCompStateTypeName)

), // type expression refering to composite state type

undefined

) // typebased composite state

else statelist[i] // unmodified elements

endif

: i ∈ 1..statelist.length

>, // state list

triggers,

timer

) // state

endlet // name

and // <composite state reference> in a <state partition> - corr. ref. case (b)

sp = <state partition>(csr = <composite state reference>(csrname))

provided

sp.surroundingScopeUnit0 = cs.surroundingScopeUnit0 ∧

csrname = graph.s-<composite state heading>.s-<composite state name>

=>

mk-<typebased state partition heading>( csrname,

mk-<type expression>(

mk-<identifier>(cs.surroundingQualifier0, cs.fullIdentifier0.anonCompStateTypeName)

), // type expression refering to composite state type

), // typebased state partition heading

and // <state machine> with a <composite state list item> - corr. ref. case (c)

sm = <composite state list item>( smname, smpars)

provided

sm.parentAS0 ∈ <interaction>∧ // the composite state list item is a state machine

sm.surroundingScopeUnit0 = cs.surroundingScopeUnit0 ∧

smname = graph.s-<composite state heading>.s-<composite state name>

=>

mk-<typebased composite state>( smname,

smpars,

mk-<type expression>(

mk-<identifier>(cs.surroundingQualifier0, cs.fullIdentifier0.anonCompStateTypeName),

empty // actual context parameter list

) // type expression refering to composite state type

) // typebased composite state

and // replace any qualifiers in cs structure that refer to cs

q = <qualifier>

provided

let structure =

if graph ∈ <composite state graph>

then graph.s-<composite state structure>

else graph.s-<aggregation structure>

endif

in

isAncestorAS0(structure, q) ∧ // is a qualifier in the composite state structure

q.parentAS0 ∈ <identifier> ∧ // part of an identifier

∀ i ∈ 1..cs.fullQualifierWithin0.s-<path item>*-*seq.length: \\ full qualifier match up to end of qual within cs

q.parentAS0.refersto0.fullQualifier0.s-<path item>*-*seq[i] = cs.fullQualifierWithin0.s-<path item>*-*seq[i]

endlet // structure

=>

mk-<qualifier>(<

if i ≠ cs.fullQualifierWithin0.s-<path item>*-*seq.length

then q.parentAS0.refersto0.fullQualifier0.s-<path item>*-*seq[i]

else mk-<path item>(state type, cs.fullIdentifier0.anonCompStateTypeName)

endif :

i = 1..q.parentAS0.refersto0.fullQualifier0.length

>)

chanStateMachineGate(fullIdentifier0(cd = <channel definition>( \*, \*, \*, cp, \*)))

provided // first channel endpoint is a state machine (in which case other endpoint is not)

chanStateMachineGate( cd.fullIdentifier0 ) = undefined ∧

cp.s1-<channel endpoint>.refersto0 ∈ <composite state definition> ∧

(∃ sm ∈ <composite state list item>:

sm.parentAS0 ∈ <interaction>∧ // composite state list item that is a state machine

sm.surroundingScopeUnit0 =

cp.s1-<channel endpoint>.refersto0.surroundingScopeUnit0 ∧ //same scope as sm

sm.s-<name> = //same name as channel endpoint

cp.s1-<channel endpoint>.refersto0.s-<composite state graph>.s-<composite state heading>.s-<name>)

=12=>

(chanStateMachineGate\{(cd.fullIdentifier0, undefined )} ) ∪ {( cd.fullIdentifier0, newName )}

chanStateMachineGate(fullIdentifier0(cd = <channel definition>( \*, \*, \*, cp, \*)))

provided // second channel endpoint is a state machine (in which case other endpoint is not)

chanStateMachineGate( cd.fullIdentifier0 ) = undefined ∧

cp.s2-<channel endpoint>.refersto0 ∈ <composite state definition> ∧

(∃ sm ∈ <composite state list item>:

sm.parentAS0 ∈ <interaction>∧ // composite state list item that is a state machine

sm.surroundingScopeUnit0 =

cp.s2-<channel endpoint>.refersto0.surroundingScopeUnit0 ∧ //same scope as sm

sm.s-<name> = //same name as channel endpoint

cp.s2-<channel endpoint>.refersto0.s-<composite state graph>.s-<composite state heading>.s-<name>)

=12=>

(chanStateMachineGate\{(cd.fullIdentifier0, undefined )} ) ∪ {( cd.fullIdentifier0, newName )}

cs = <composite state definition>(

// transform <composite state definition> cs to <composite state type definition>

graph = <composite state graph>(uses, // package uses

hdg = <composite state heading>(virt, // virtuality

qual // qualifier – should be undefined = defined in context

name, // name

special, // specialization

params // agent formal parameters

), // composite state heading

structure = <composite state structure>(connects, entities, body)

) // composite state graph

) // composite state

provided

cs.fullIdentifier0.anonCompStateTypeName ≠ undefined ∧

((∃ sm ∈ <composite state list item>:

(sm.parentAS0 ∈ <interaction>∧ // composite state list item that is a state machine

sm.surroundingScopeUnit0 = cs.surroundingScopeUnit0 ∧ //same scope as cs

sm.s-<name> = //same name as composite state

cs.s-<composite state graph>.s-<composite state heading>.s-<name>)

)

⇒ // for a state machine, the gate name for a channel

(∃ cd ∈ <channel definition>: // channel with defined gate in state machine

(chanStateMachineGate( cd.fullIdentifier0 ) ≠ undefined ∧

( cd.s-<channel path>.s1-<channel endpoint>.refersto0 = cs ∨

cd.s-<channel path>.s2-<channel endpoint>.refersto0 = cs ))

))

=12=>

mk-<composite state type definition>(

mk-<composite state type graph>( uses, // package uses

mk-<composite state type heading>( virt, // virtuality

qual // qualifier – should be undefined = defined in context

cs.fullIdentifier0.anonCompStateTypeName, // anonymous name

undefined, // formal context parameters

undefined, // virtuality constraint

special, // specialization

params // agent formal parameters

), // composite state type heading

mk-<composite state structure>(connects,

entities ⁀ // which includes procedure refs and composite state type refs

if (∃ sm ∈ <composite state list item>:

(sm.parentAS0 ∈ <interaction>∧ // composite state list item that is a state machine

sm.surroundingScopeUnit0 = cs.surroundingScopeUnit0 ∧ //same scope as cs

sm.s-<name> = //same name as channel endpoint

cs.s-<composite state graph>.s-<composite state heading>.s-<name>))

then

< // list of gate definitions for channels connected to a state machine

mk-<textual gate definition>(

ch.fullIdentifier0.chanStateMachineGate, // name

ch.s-<encoding rules>,

if ch.s-<channel path>.s2-<channel endpoint>.refersto0 = cs

then // channel to state machine

if ch.s2-<channel path> ≠ undefined

then // in and out signals

mk-<gate constraint>(**in**, undefined, ch.s-<channel path>.s-<signal list>,

**out**, undefined, ch.s2-<channel path>.s-<signal list>)

else // in only

mk-<gate constraint>(**in**, undefined, ch.s-<channel path>.s-<signal list>,

undefined, undefined, undefined)

endif

else // channel from state machine

if ch.s2-<channel path> ≠ undefined

then // in and out signals

mk-<gate constraint>(**in**, undefined, ch.s2-<channel path>.s-<signal list>,

**out**, undefined, ch.s-<channel path>.s-<signal list>)

else // out only

mk-<gate constraint>(**out**, undefined, ch.s-<channel path>.s-<signal list)

endif

endif

)

: ch ∈ <channel definition> ∧

( (ch.s-<channel path>.s1-<channel endpoint>.refersto0 = cs) ∨

(ch.s-<channel path>.s2-<channel endpoint>.refersto0 = cs)

)

> // end of list of gate in definitions for channels connected to a state machine

else empty

endif , // entities

body

),

cs.fullIdentifier0.anonCompStateTypeName // optional matching name

) // composite state type graph

) // composite state type definition

and // fill in the gate channel endpoint connecting to a state machine

<channel definition>( cname,

encoding,

delay,

cp1 = <channel path>(

ce11 = <channel endpoint>( id11, g11),

ce12 = <channel endpoint>( id12, g12),

signals1

), // cp1 channe path

cp2 = <channel path>(

ce21 = <channel endpoint>( id21, g21),

ce22 = <channel endpoint>( id22, g22),

signals2

) // cp2 channe1 path

) // channel definition

=>

mk-<channel definition>( cname,

encoding,

delay,

mk-<channel path>(

mk-<channel endpoint>( id11,

if g11 ≠ undefined then g11 else id11.chanStateMachineGate endif

), // first channel endpoint of first channel path

if ce12 = undefined then undefined else

mk-<channel endpoint>( id12,

if g12 ≠ undefined then g12 else id12.chanStateMachineGate endif

endif), // second channel endpoint of first channel path

signals1

), // first channel path

if cp2 = undefined then undefined else

mk-<channel path>(

mk-<channel endpoint>( id21,

if g21 ≠ undefined then g21 else id21.chanStateMachineGate endif

if ce22 = undefined then undefined else

mk-<channel endpoint>( id22,

if g22 ≠ undefined then g22 else id22.chanStateMachineGate endif

endif), // second channel endpoint of second channel path

signals2

endif) // second channe1 path

) // channel definition

*The following paragraph is a general description copied from clause* 9 *Model* of [ITU‑T Z.103] *and modified to apply to AS0*.

A <composite state definition> is shorthand for a <composite state type definition> and it has a corresponding <state> or <composite state reference> or <state machine> referencing the <composite state definition>. A <state> or <state machine> that references the <composite state definition> contains a <composite state list item> with a <composite state name> that identifies the <composite state definition>. The <composite state list item> is transformed into a <typebased composite state> as described in clause 11.2 of [ITU‑T Z.103]. Each <gate> of the transformed <state machine> identifies the anonymously named <textual gate definition> of the <composite state type> that corresponds to the set of channels connected to the original <state machine> at the same point. The identified channels all join the transformed <state machine> at the same <gate in definition> inside the <state machine>. A <composite state reference> that references a <composite state definition> is transformed as described in clause 9.5 of [ITU‑T Z.103].

*The following paragraphs are further description copied from clause* 11.11.1 *Model* of [ITU‑T Z.103] *and modified to apply to AS0*.

A <composite state definition> is shorthand for a <composite state type definition> transformed as described after the NOTE below. The corresponding reference references the <composite state definition> by its <state name> and is:

a) a <state> with a <state list> item that is a <composite state list item>; or

b) a <composite state reference> in a <state partition>; or

c) a <state machine> with a <composite state list item>.

A corresponding <state> is shorthand for a <state> containing a <typebased composite state> that uses the <composite state type definition> (see also NOTE below). A corresponding <composite state reference> is shorthand for a <textual typebased state partition def> (that is, a <typebased state partition heading>) that uses the <composite state type definition> (see also NOTE below). A corresponding <state machine> is shorthand for a <state> containing a <typebased composite state> that uses the <composite state type definition> (see also NOTE below).

NOTE – In the graphical grammar a corresponding <state area>, <composite state reference area> or <state machine area> is also shorthand for a <composite state type reference area> to the <composite state type diagram>, but in AS0 such a <composite state type reference> to the <composite state type definition> is not needed because the <composite state type definition> is created in the local scope.

If the composite state is not an aggregation, the <composite state type definition> is formed from the <composite state definition> by inserting the keyword **type** after the keyword **state** in the heading and changing the name to an anonymous name. The <composite state type heading> therefore has the <virtuality> given in the <composite state heading> of the <composite state definition>. The anonymous name is generated in the same way for redefinitions, so that the names match for the virtual type and redefinitions. The <composite state structure> is copied from the <composite state definition> to the <composite state type definition>. In the <composite state structure>, any part of a <qualifier> that refers to the composite state of the <composite state definition> is changed to refer to the composite state type of the <composite state type definition>. If the <composite state definition> is for a state machine, there is also a connected <gate in definition> for the composite state type definition with a unique anonymous name, for each channel attached to the <state machine area> for the <composite state definition>.

The model for a composite state that is an aggregation is described in clause 11.11.2 of [ITU‑T Z.103] and is transformed in clause F2.2.8.11.2 below.

The transform for a composite state with only an asterisk state is in F2.2.8.11.1.

A <composite state definition> that has a specialization (with a <composite state heading> that contains a <specialization>) is shorthand for defining an implicit composite state type and one typebased composite state of this type.

A <procedure reference> is moved with the same <procedure reference heading> to the composite state type.

A <composite state type reference> is moved with the same <type preamble>, <qualifier>, <composite state type name> and <formal context parameters> to the composite state type.

Auxiliary functions

The function anonCompStateTypeName is a mapping from composite state identities to names, used to store the name of the composite state type created from the transform of a <composite state definition> of a scope unit.

controlled anonCompStateTypeName: { cs.fullIdentifier0: cs ∈ <composite state definition> } → <name>

initially ∀ cs ∈ <composite state definition> : cs.fullIdentifier0 = undefined

The function chanStateMachineGate is a mapping from channel identities to names, used to store the name of the gate for channel connection. The gate is in composite state type from the transform of a <composite state definition>.

controlled chanStateMachineGate: { cd.fullIdentifier0: cd ∈ <channel definition> } → <name>

initially ∀ cd ∈ <channel definition> : cd.fullIdentifier0 = undefined

##### F2.2.8.11.1 Composite state graph

Abstract syntax

Composite-state-graph :: State-transition-graph

[ Entry-procedure-definition ]

[ Exit-procedure-definition ]

Concrete syntax

<composite state graph> ::

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

<composite state heading> ::

<virtuality> [ <qualifier> ] <composite state name> [ <specialization> ] <agent formal parameters>

<composite state structure> ::

<state connection points>\* <entity in composite state>\*

<composite state body>

<entity in composite state> =

<valid input signal set>

| <variable definition>

| <data definition>

| <select definition>

| <procedure definition>

| <procedure reference>

| <composite state definition>

| <composite state type definition>

| <composite state type reference>

| <gate in definition>

<composite state body> ::

<start>\* {<state> | <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 definition>∪<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>))⇒ csg.surroundingScopeUnit0.startSet0≠∅

If a <composite state body> contains at least one <state> different from asterisk state, a <start> must be present.

∀cs∈<composite state definition>: ∀vd∈<variable definition>:

(vd.surroundingScopeUnit0 = cs) ∧ (cs.surroundingScopeUnit0∈<internal procedure definition>) ⇒

(vd.s-exported = undefined)

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

Transformations

<composite state body>(empty,

items1 ⁀ < <state>(<asterisk state list>(undefined), triggers, undefined) > ⁀ items2)

provided < item in (items1 ⁀ items2): (item ∈ <state>) > = empty

=8=>

let nn = newName in // anonymous state name

<composite state body>(

< mk-<start>(undefined, // virtuality

undefined, // state entry point name – default entry with no name

mk-<terminator>( // start <transition> = <terminator>

undefined, // no label

mk-<nextstate body name>(nn) // <nextstate> = <next state body> = <next state body name>

) // terminator

) // start

>, // start list

items1 ⁀ < mk-<state>(< nn >, triggers, undefined) > ⁀ items2) // state/free-action list

endlet // nn

If a <composite state definition> consists of no <state>s with <state name>s but only a <state> with <asterisk>, the asterisk state is transformed into a <state> with an anonymous <state name> and a <start> leading to this <state>. Modified from clause 11.11.1 *Model* of [ITU‑T Z.103].

Auxiliary functions

The function defaultEntryPoint associates each <composite state type definition> with an implicit anonymous name for the unnamed start of the graph.

controlled defaultEntryPoint: <composite state type definition> → <name>

initially ∀ cstd ∈ <composite state type definition>: cstd.defaultEntryPoint = undefined

The function defaultExitPoint associates each <composite state type definition> with an implicit anonymous name for the unnamed return from the graph.

controlled defaultExitPoint: <composite state type definition> → <name>

initially ∀ cstd ∈ <composite state type definition>: cstd.defaultExitPoint = undefined

Get the set of entry names used in a <composite state definition> or a <composite state type definition>.

usedEntryNameSet0(csd: <composite state definition>∪<composite state type definition>):<name>-set=def

{n∈<name>: n.parentAS0∈csd.startSet0}

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

usedExitNameSet0(csd: <composite state definition>∪<composite state type definition>):<name>-set=def

{n∈<name>: ∃s∈csd.stateSet0: isAncestorAS0(s, n)∧(n.parentAS0.parentAS0∈<return>)}

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

definedEntryNameSet0(csd: <composite state definition>∪<composite state type definition>):<name>-set=def

{n∈<name>: (n.parentAS0∈<state entry points>)∧

(n.parentAS0.parentAS0=csd.s-<composite state structure>)}

Get the set of exit names defined in a <composite state definition> or a <composite state type definition>.

definedExitNameSet0(csd: <composite state definition>∪<composite state type definition>):<name>-set=def

{n∈<name>: (n.parentAS0∈<state exit points>)∧

(n.parentAS0.parentAS0=csd.s-<composite state structure>)}

##### F2.2.8.11.2 State aggregation

Abstract syntax

State-aggregation-node :: State-partition-set

[ Entry-procedure-definition ]

[ Exit-procedure-definition ]

State-partition :: Name

Composite-state-type-identifier

Connection-definition-set

Connection-definition = Entry-connection-definition

| Exit-connection-definition

Entry-connection-definition :: Outer-entry-point Inner-entry-point

Outer-entry-point :: State-entry-point-name

Inner-entry-point :: Nextstate-parameters

Exit-connection-definition :: Outer-exit-point Inner-exit-point

Outer-exit-point :: State-exit-point-name

Inner-exit-point :: State-exit-point-name

Conditions on abstract syntax

∀pn∈State-entry-point-name: (pn.parentAS1∈Outer-entry-point) ⇒

(pn∈ pn.surroundingScopeUnit0.entryPointSet1)∧

(pn.surroundingScopeUnit0.s-implicit∈State-aggregation-node)

The State-entry-point-name in the Outer-entry-point must denote a State-entry-point-definition of the Composite-state-type-definition where the State-aggregation-node occurs.

∀pn∈State-entry-point-name: (pn.parentAS1∈Inner-entry-point) ⇒

(pn∈parentAS1ofKind(pn, State-partition).baseType1.entryPointSet1)

The State-entry-point-name of the Inner-entry-point must denote a State-entry-point-definition of the composite state in the State-partition.

∀pn∈State-exit-point-name: (pn.parentAS1∈Outer-exit-point) ⇒

(pn∈ pn.surroundingScopeUnit0.exitPointSet1)∧

(pn.surroundingScopeUnit0.s-implicit∈State-aggregation-node)

∀pn∈State-exit-point-name: (pn.parentAS1∈Inner-exit-point) ⇒

(pn∈parentAS1ofKind(pn, State-partition).baseType1.exitPointSet1)

Likewise, the Outer-exit-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 pointSet1=

{pn∈State-entry-point-definition∪State-exit-point-definition: isAncestorAS1(cd,pn)} in

pointSet ⊆ pointSet1

endlet)

All entry and exit points of both the container state and the state partitions must appear in exactly one Connection-definition.

∀sp,sp1∈ State-partition: (sp≠sp1)∧ (sp.parentAS1= sp1. parentAS1)⇒

sp.inputSignalSet1 ∩ sp1. inputSignalSet1=∅

The input signal sets of the *State-partition*s within a composite state must be disjoint.

Concrete syntax

<state aggregation> ::

<package use clause>\* <state aggregation heading> <aggregation structure>

<state aggregation heading> =

<composite state heading>

<aggregation structure> ::

{ <state connection points> | <entity in state aggregation> }\*

<state aggregation body>

<state aggregation body> = { <state partition> | <state partition connection> }\*

<state partition connection> = <state partition connection entry>

| <state partition connection exit>

<state partition connection entry> ::

[ <composite state<identifier> ] <outer entry points>

<composite state<identifier> <inner entry point>

<outer entry points> :: <state entry point>+ | default

<inner entry point> :: <nextstate parameters> | <state entry point> | default

<state entry point> = <state entry point<name>

<state partition connection exit> ::

[ <composite state<identifier> ] <outer exit point>

<composite state<identifier> <inner exit points>

<outer exit point> :: <state exit point> | default

<inner exit points> :: <state exit points> | default

<state partition> =

<textual typebased state partition def> | <composite state reference> | <composite state definition>

<entity in state aggregation> =

<valid input signal set>

| <variable definition>

| <data definition>

| <select definition>

| <procedure definition>

| <procedure reference>

| <composite state type definition>

| <composite state type reference>

| <gate in definition>

Transformations

NOTE – The allocation of an anonymous name for the composite state and transform of references is the same for a composite state graph and a state aggregation, and is defined in the first transform (and its dependent transforms) in F2.2.8.11.

cs = <composite state definition>(

// transform <composite state definition> cs to <composite state type definition>

graph = <state aggregation>(uses, // package uses

hdg = <composite state heading>(virt, // virtuality

qual // qualifier – should be undefined = defined in context

name, // name

special, // specialization

params // agent formal parameters

), // composite state heading (alias state aggregation heading)

structure

) // state aggregation

) // composite state

provided

cs.fullIdentifier0.anonCompStateTypeName ≠ undefined ∧

((∃ sm ∈ <composite state list item>:

(sm.parentAS0 ∈ <interaction>∧ // composite state list item that is a state machine

sm.surroundingScopeUnit0 = cs.surroundingScopeUnit0 ∧ //same scope as cs

sm.s-<name> = //same name as composite state

cs.s-<composite state graph>.s-<composite state heading>.s-<name>)

)

⇒ // for a state machine, the gate name for a channel

(∃ cd ∈ <channel definition>: // channel with defined gate in state machine

(chanStateMachineGate( cd.fullIdentifier0 ) ≠ undefined ∧

( cd.s-<channel path>.s1-<channel endpoint>.refersto0 = cs ∨

cd.s-<channel path>.s2-<channel endpoint>.refersto0 = cs ))

))

=12=>

mk-<composite state type definition>(

mk-<state aggregation type>( uses, // package uses

mk-<state aggregation type heading>( virt, // virtuality

qual // qualifier – should be undefined = defined in context

cs.fullIdentifier0.anonCompStateTypeName, // anonymous name

undefined, // formal context parameters

undefined, // virtuality constraint

special, // specialization

params // agent formal parameters

), // state aggregation type heading

structure

) // state aggregation type

) // composite state type definition

The <composite state type definition> for the aggregation is formed from the <composite state definition> by changing the <state aggregation> into a <state aggregation type> by inserting the keyword **type** after the keyword **aggregation** in the heading and changing the name to an anonymous name. The <state aggregation type heading> therefore has the <virtuality> given in the <state aggregation heading> of the <composite state definition>. The anonymous name is generated in the same way for redefinitions, so that the names match for the virtual type and redefinitions. The <aggregation structure> is copied from the <composite state definition> to the <composite state type definition>. In the <composite state structure>, any part of a <qualifier> that refers to the composite state of the <composite state definition> is changed to refer to the composite state type of the <composite state type definition>. Modified text from clause 11.11.2 *Model* of [ITU‑T Z.103].

*The following paragraph* (see clause 11.11.2 *Semantics* of [ITU‑T Z.102]) *is formalized by the dynamic semantics.*

If there are signals in the complete valid input set of the Composite-state-type-definition where a State-aggregation-node occurs that are not consumed by any *State-partition* of the *State-aggregation-node*, there is an implied additional *State-partition*. The implied *Composite-state-type-definition* for this *State-partition* has a *Composite-state-graph* that is a *State- transition-graph* with a single unlabelled *State-start-node* with a transition to a single *State-node*. This *State-node* has an *Input-node* for each signal handled (including those for exported procedures and exported variables) and a *Continuous-signal*. The *Transition* of each *Input-node* is an empty transition back to the state. The *Continuous-signal* has a *Continuous-expression* that is a logical 'and' that accesses implicit Boolean variables, one for each (non-implicit) partition. These variables are shared by all partitions and each initialized to False. When each of the other partitions has interpreted either an *Action-return-node* or *Named-return-node*, its implicit Boolean variable is set to True. The *Transition* of the *Continuous-signal* contains just an *Action-return-node*. Therefore, when other partitions have completed, and the additional partition has consumed all the signals for it (if any) in the input port, the partition exits through an *Action-return-node*. See clause 11.11.2 *Semantics* of [ITU‑T Z.102].

Auxiliary functions

Get the set of input signals appearing in a type definition, a state partition or a state node.

inputSignalSet1(sp: TypeDefinition1∪State-partition ∪State-node): Signal1 =def

if sp∈State-node then

sp.s-Save-signalset∪{in.s-Signal-identifier | in∈sp.s-Input-node-set}∪

{getEntityDefinition1(cn.**s**-Procedure-identifier,procedure).inputSignalSet1 |

cn∈{cn∈Call-node: isAncestorAS1(sp, cn)}}

else // sp∈TypeDefinition1∪State-partition

{sn.inputSignalSet1 | sn∈sp.stateNodeSet1}

endif

Get the base type of a definition.

baseType1(as:

Agent-definition∪Composite-state-type-definition∪State-machine∪State-partition∪State-node):

Agent-type-definition∪Composite-state-type-definition=def

case as of

| Agent-definition then

if as.s-Agent-type-identifier=undefined

then undefined

else getEntityDefinition1(as.s-Agent-type-identifier, agent type)

endif

| Composite-state-type-definition∪State-machine∪State-partition∪State-node then

if as.s-Composite-state-type-identifier=undefined

then undefined

else getEntityDefinition1 (as.s-Composite-state-type-identifier, state type)

endif

otherwise undefined

endcase

Get the set of the state entry points of a *Composite-state-type-definition*.

entryPointSet1(d: Composite-state-type-definition): Name-set=*def*

d.s-State-entry-point-definition-set

Get the set of the state exit points of a *Composite-state-type-definition*.

exitPointSet1(d: Composite-state-type-definition): Name-set=*def*

d.s-State-exit-point-definition-set

Mapping to abstract syntax

| <state partition connection entry>(<outer entry points>( n1 ), <inner entry point>( n2 )) then

let innerEntryPoint =

mk-Inner-entry-point(

case n2 of

| <nextstate parameters> then Mapping(n2)

| <state entry point> then mk-Nextstate-parameters(empty,Mapping(n2))

otherwise // default

mk-Nextstate-parameters(empty,Mapping(newName))

endcase

)

in

if n1 = **default**

then {mk-Entry-connection-definition(Mapping(newName), innerEntryPoint)}

else // n1 = state entry point list

{ mk-Entry-connection-definition(

mk-Outer-entry-point(n1[i]),

innerEntryPoint)

: i ∈ 1.. n1.length

}

endif

endlet // innerEntryPoint

| <state partition connection exit>(<outer exit point>(n1), <inner exit points>( n2 )) then

let outerExitPoint =

mk-Outer-exit-point(

case n1 of

| <state exit point> then Mapping(n1)

otherwise // default

Mapping(newName)

endcase

)

in

if n2 = **default**

then {mk-Exit-connection-definition(outerExitPoint), Mapping(newName)}

else // n2 = state exit point list

{ mk-Exit-connection-definition(

outerExitPoint,

mk-Inner-exit-point(Mapping(n2[i]))

: i ∈ 1.. n2.length

}

endif

endlet // outerExitPoint

##### F2.2.8.11.3 State connection point

Concrete syntax

<state connection points> = <state entry points> | <state exit points>

<state entry points> :: <state entry point>+

<state exit points> :: <state exit point>+

<state exit point> = <state exit point<name>

Mapping to abstract syntax

| <state entry points>(x) then Mapping(x)

| <state exit points>(x) then Mapping(x)

##### F2.2.8.11.4 Connect

Abstract syntax

Connect-node :: [ State-exit-point-name ] Transition

∀ sn ∈ State-node: ∀cn,cn1∈Connect-node:

(cn≠cn1)∧(in.parentAS1=sn)∧(in1.parentAS1=sn)⇒

cn.**s**-State-exit-point-name≠cn1.**s**-State-exit-point-name

The *Connect-node-****set*** shall contain at most one unnamed *Connect-node*. See clause 11.2 of [ITU‑T Z.101].

Each *Connect-node* in the *Connect-node-set* of a composite state application shall either be the only *Connect-node* without a *State-exit-point-name* or have a *State-exit-point-name* that is different from every other *Connect-node* in the *Connect-node-set*. See clause 11.2 of [ITU‑T Z.102].

Concrete syntax

<connect part> ::

[<virtuality>] <connect list> <exit transition>

<exit transition> = <transition>

<connect list> = <state exit point list> | <asterisk connect list>

<state exit point list> = <state exit point<name>\*

<asterisk connect list> :: ()

Conditions on concrete syntax

∀cl∈<connect list>: ∀pn∈<name>:

isAncestorAS0(cl,pn) ⇒

(∃scp∈<state connection points>: ∃sep∈<state exit points>:

isAncestorAS0 (scp, sep) ∧ isAncestorAS0 (scp.parentAS0, pn) ∧ (pn = sep))

The <connect list> must only refer to visible <state exit point>s.

Transformations

< <connect part>(virt, < n > ⁀ rest, trans) > provided rest ≠ empty

=8=> < <connect part>(virt, < n >, trans), <connect part>(virt, rest, trans) >

When the <connect list> of a certain <connect part> contains more than one <state exit point>, a copy of the <connect part> is created for each such <state exit point>. Then the <connect part> is replaced by these copies. See clause 11.11.4 *Model* of [ITU‑T Z.103].

c=<connect part>(virt, <asterisk>, trans)

=8=>

(let parentType = c.parentAS0.s-<state list>.head.s-<type expression>.refersto0 in

let allExits = bigSeq(< < ex.s-<name> | ex in exits > |

exits in parentType.s-<composite state structure>.s-<state connection points>-seq:

(exits ∈ <state exit points>) >) in

<connect part>(virt, allExits, trans)

endlet

endlet)

A <connect list> that contains an <asterisk connect list> is transformed into a list of <state exit point>s, one for each <state exit point> of the <composite state definition> in question. The list of <state exit point>s is then transformed as described above. See clause 11.11.4 *Model* of [ITU‑T Z.103].

Mapping to abstract syntax

| <connect part>(\*, < name >, trans) then

mk-Connect-node(Mapping(name),Mapping(trans))

#### F2.2.8.12 Transition

##### F2.2.8.12.1 Transition body

Abstract syntax

Transition :: Graph-node\* { Terminator | Decision-node}

Graph-node :: { Task-node

| Output-node

| Create-request-node

| Call-node

| Compound-node

| Set-node

| Reset-node

}

Terminator :: { Nextstate-node

| Stop-node

| Return-node

| Join-node

| Continue-node

| Break-node

}

Concrete syntax

<transition> = <transition action items> | <terminator>

<transition action items> :: <transition string> [<terminator>]

<transition string> = <action>+

<action> :: [<label>] <action item>

<action item> =

<task>

| <output>

| <create request>

| <decision>

| <set>

| <reset>

| <export statement>

| <procedure call>

| <remote procedure call>

| <transition option>

<terminator> :: [<label>] <terminator node>

<terminator node> = <nextstate> | <join> | <stop> | <return>

Conditions on concrete syntax

∀t ∈<transition>: (t ∉ <terminator>) ∧ (t.**s**-<terminator> = undefined) ∧

(t.parentAS0.parentAS0.parentAS0∉<decision>∪<transition option>) ⇒

(**let** asl = t.**s**-<action>-seq **in**

(asl.last.s-<action item>∈<decision>∪<transition option>)∧

isTransitionTerminating0(asl.last.<action item>) endlet)

endlet)

If the <terminator> of a <transition> is omitted, then the last action in the <transition> must contain a terminating <decision> or terminating <transition option>, except when a <transition> is contained in a <decision> or <transition option>.

Transformations

*The following transformation is handled in the transformations for remote procedure call and import expression.*

A transition action may be transformed to a list of actions (possibly containing implicit states) according to the transformation rules for <import expression> and <remote procedure call>.

Mapping to abstract syntax

| <transition action items>(s, t) then

if t = undefined

then mk-Transition(Mapping(<s[i]: i=1..s.length-1∧ (s[i] ∉ <decision>) >), Mapping(s.last))

else mk-Transition(Mapping(s), Mapping(t))

endif

| <action>(\*, a) then Mapping(a)

| <terminator>( l, t) then

if l ≠ undefined

then mk-Free-action(Mapping(l)),mk-Transition(empty, Mapping(t)))

else mk-Transition(empty, Mapping(t))

endif

Auxiliary functions

Determine if a <decision> or a <transition option> is terminating.

isTransitionTerminating0(dt: <decision>∪<transition option>): Boolean =def

∀t∈<transition>: (t.parentAS0.parentAS0.parentAS0= dt) ⇒

((t∈<terminator>)∨

((t∉<terminator>)∧(t.**s**-<terminator> ≠ undefined))∨

((t∉<terminator>)∧(t.**s**-<terminator> = undefined)∧

isTransitionTerminating0( t.**s**-<action>-seq.last.<action item>))

##### F2.2.8.12.2 Nextstate

Abstract syntax

Nextstate-node = Named-nextstate | Dash-nextstate

Dash-nextstate :: [ HISTORY ]

Named-nextstate :: State-name [ Nextstate-parameters ]

Nextstate-parameters :: Actual-parameters

[ State-entry-point-name ]

Conditions on abstract syntax

∀nn ∈ Nextstate-node: nn.**s**-Nextstate-parameters ≠ undefined⇒

((nn.**s**-State-name = sn.**s**-State-name)∧

(parentAS1ofKind(nn,State-transition-graph∪Procedure-graph) =

parentAS1ofKind(sn,State-transition-graph∪Procedure-graph))∧

(nn.**s**-Nextstate-parameters ≠ undefined))⇒

sn.**s**-Composite-state-type-identifier≠ undefined)

The State-name specified in a nextstate must be the name of a state within the same State-transition-graph or Procedure-graph.If a *Named-nextstate* includes *Nextstate-parameters*, the *State-name* shall refer to a composite state (a *State-node* with a *Composite-state-type-identifier*).

Concrete syntax

<nextstate> = <nextstate body>

<nextstate body> = <nextstate body name> | <dash nextstate> | <history dash nextstate>

<nextstate body name> ::

<basic state<name>

| <composite state<name> <nextstate parameters>

<nextstate parameters> :: <actual parameters> [<state entry point<name>]

<dash nextstate> :: ()

<history dash nextstate> :: ()

Conditions on concrete syntax

∀s∈<state>: ∀t∈<transition>:

(t.parentAS0∈<input part>∪<priority input>∪<spontaneous transition>∪<continuous signal>) ∧

(t.parentAS0.parentAS0= s) ∧

(∃hdn∈<history dash nextstate>: isAncestorAS0(t,hdn)) ⇒

(s.surroundingScopeUnit0∈<composite state definition> ∪<composite state type definition>)

If a transition is terminated by a <history dash nextstate>, the <state> must be a <composite state definition>.

∀t∈<transition>: (t.parentAS0∈<start>) ⇒ ¬(∃dn∈<dash nextstate>: isAncestorAS0(t,dn))

The <transition> contained in a <start> must not lead, directly or indirectly, to a <dash nextstate>.

∀t∈<transition>: (t.parentAS0∈<start>) ⇒

¬(∃dn∈<history dash nextstate>: isAncestorAS0(t,dn))

The <transition> contained in a <start> must not lead, directly or indirectly, to a <history dash nextstate>.

Transformations

*The following text is handled by the dynamic semantics.*

In each <nextstate> of a <state> the <dash nextstate> is replaced by the <state name> of the <state>. This model is applied after the transformation of <state>s and all other transformations except those for trailing commas, synonyms, priority inputs, continuous signals, enabling conditions, implicit tasks for imperative actions and remote variables or procedures.

Mapping to abstract syntax

| <nextstate parameters>( params, n) then mk-Nextstate-parameters(Mapping(params), Mapping(n))

##### F2.2.8.12.3 Join

Abstract syntax

Join-node :: Connector-name

Concrete syntax

<join> :: <connector<name>

Conditions on concrete syntax

∀b∈<agent body>∪<procedure body>∪<operation body>∪<composite state body>:

∀j∈<join>: isAncestorAS0(b, j) ⇒

(∃!l∈<label>: isAncestorAS0(b, l)∧(j.**s**-<name> = l.**s**-<name>))

There must be exactly one <connector<name> corresponding to a <join> within the same body.

Mapping to abstract syntax

| <join>(name) then mk-Terminator(mk-Join-node(Mapping(name)), undefined)

##### F2.2.8.12.4 Stop

Abstract syntax

Stop-node :: ()

Conditions on abstract syntax

Concrete syntax

<stop> :: ()

Mapping to abstract syntax

| <stop>() then mk-Terminator(mk-Stop-node(), undefined)

##### F2.2.8.12.5 Return

Abstract syntax

Return-node = Action-return-node

| Value-return-node

| Named-return-node

Action-return-node :: ()

Value-return-node :: Expression

Named-return-node :: State-exit-point-name

Conditions on abstract syntax

∀rn ∈ Return-node: ∃pg ∈ Procedure-graph: isAncestorAS1(pg, rn)

A Return-node must be contained in a Procedure-graph.

∀rn ∈ Action-return-node: ∃d ∈ Procedure-definition:

isAncestorAS1(d.**s**-Procedure-graph, rn) ∧ d.**s**-Result = undefined

An Action-return-node must only be contained in the Procedure-graph of a Procedure-definition without Result.

∀rn ∈ Value-return-node: ∃d ∈ Procedure-definition:

isAncestorAS1(d.**s**-Procedure-graph, rn) ∧ d.**s**-Result ≠ undefined

A Value-return-node must only be contained in the Procedure-graph of a Procedure-definition containing Result.

∀rn ∈ Named-return-node: ∃sg ∈ Composite-state-graph: isAncestorAS1(sg, rn)

A Named-return-node must only be contained in a Composite-state-graph.

Concrete syntax

<return> :: <return body> | <state exit point<name>

<return body> :: [<expression>]

Mapping to abstract syntax

| <return>(x) then

if x = undefined then mk-Terminator(mk-Action-return-node())

elseif x ∈ <name> then mk-Terminator(mk-Named-return-node(Mapping(x)))

else mk-Terminator(mk-Value-return-node(Mapping(x)

endif

#### F2.2.8.13 Action

##### F2.2.8.13.1 Task

Abstract syntax

Task-node = Assignment

| Informal-text

Concrete syntax

<task> :: { <task body> | <informal text> | <legacy task body> }

<task body> :: <variable definitions> <statements>

<legacy task body> = <assignment>+

Transformations

< <task>(<task body>(<statements>(\*, empty))) >

=3=> empty

If a <task body> of <task> is empty, the <task> is removed. Any syntactic item leading to such an empty <task> shall then lead directly to the item following the <task>. Modified from clause 11.13.1 *Model* of [ITU‑T Z.103].

< <task>(<task body>( vl, sl )) >

=3=> < <compound statement>( undefined, vl, sl ) >

Each <non terminating statement> of <task body> of a <task> represents an element of the *Graph-node* list for the *Transition* of the <transition> containing the <task> in the order that each <non terminating statement> occurs in the <task>. See clause 11.13.1 *Concrete grammar* of [ITU‑T Z.102].

NOTE 1 – For this mapping of <task body>, the <task body> is transformed into a <compound statement> as above, and then the <compound statement> is transformed and mapped as shown in clause F2.2.8.14.1. As a consequence, the transform of a <task> containing a <task body> is not necessarily mapped onto a Task-node.

< <task>( al ) >

provided al ∉ (<task body> ∪ <informal text>)

=3=> < <compound statement>( undefined, undefined, al ) >

NOTE 2 – Similar to the mapping of <task body> with variable definitions and a<statements>, the <task body> with a <legacy task body> is transformed into a <compound statement>, and then the <compound statement> is transformed and mapped as shown in clause F2.2.8.14.1.

NOTE 3 – The concrete syntax of <legacy task body> is distinguished from a <task body> only containing assignments by the punctuation used: a <legacy task body> has assignments separated by commas, whereas in a <task body> each assignment is terminated by a semi-colon. See clause 11.13.1 *Concrete grammar* of [ITU‑T Z.103].

Mapping to abstract syntax

| <task>(t) then mk-Graph-node(Mapping(t))

Auxiliary functions

The function assignmentTaskAction0 provides the representation of a single assignment in a task symbol (used - for example - in the model for remote procedures) as an <action>, where the variable <identifier> and <operand5> expression are given as parameters. This function is therefore used where an <action> to assign to a simple variable is needed. This function is not used in this clause.

assignmentTaskAction0(v: <identifier>, e: <operand5>): <action> =def

mk-<action>(

undefined, // omitted <label> of <action>

mk-<task>( mk-<task body>(

empty, // empty variable definition list of <task body>

< // <task body> <statements>=<non terminating statements>=<non terminating statement> list

undefined, // omitted <connector name> of <non terminating statement>

mk-<assignment statement>( v, e)

> // end <task body> <statements>=<non terminating statements>=<non terminating statement> list

)) // )<task body>)<task>

) //)<action>

##### F2.2.8.13.2 Create

Abstract syntax

Create-request-node :: { Agent-identifier | THIS }

Actual-parameters

Conditions on abstract syntax

∀n ∈ Create-request-node: ∀d ∈ Agent-definition:

(d=getEntityDefinition1(n.**s**-Agent-identifier,agent))⇒

(∃t ∈ Agent-type-definition:

(t=getEntityDefinition1(d.s-Agent-type-identifier,agent type)) ∧

isActualAndFormalParameterMatched1(n.**s**-Expression-seq, t.formalParameterSortList1))

The length of the list of Actual-parameter*s* must be the same as the number of Agent-formal-parameters in the Agent-definition of the Agent-identifierand each Expression of Actual-parameters corresponding by position to an Agent-formal-parametermust have a sort that is compatible to the sort of the Agent-formal-parameter in the Agent-definition denoted by Agent-identifier.

Concrete syntax

<create request> :: <create body>

<create body> :: { <identifier> | **this** } [ <actual parameters> ]

Conditions on concrete syntax

∀cr∈<create body>: (cr.s-implicit = this) ⇒

(cr.surroundingScopeUnit0∈<agent type definition> ) ∧

(cr.surroundingScopeUnit0.surroundingScopeUnit0∈<agent type definition> )

**this** may only be specified in an <agent type definition> and in scopes enclosed by an <agent type definition>.

Transformations

*The following statement is formalized in the dynamic semantics.*

Stating **this** is derived syntax for the implicit <process identifier> that identifies the set of instances of the agent in which the create is being interpreted.

c=<create request>(<create body>(id, params))

provided id.refersto0 ∈ <agent type definition> ∧ c.possibleInstances0 ≠ empty

=8=>

(let inst = c.possibleInstances0 in

if inst.length = 1 then <create request>(<create body>(inst.head.identifier0, params))

else <decision>(any, <textual decision body>(<

<textual answer part>(undefined,

<transition action items>(<action>(undefined,

<create request>(<create body>(elem.identifier0, params)),

undefined))

| elem in inst >, undefined))

endif

endlet)

If <agent type identifier> is used in a <create request> and there exists one instance set of the indicated agent type in the agent containing the instance that performs the create, the <agent type identifier> is derived syntax denoting this instance set.

If there is more than one instance set it is determined at interpretation time in which set the instance will be created. The <create request> is in this case replaced by a non-deterministic decision using any followed by one branch for each instance set. In each of the branches a create request for the corresponding instance set is inserted.

let nn = newName in

c=<create request>(<create body>(id, params)

provided id.refersto0 ∈ <agent type definition> ∧ c.possibleInstances0 = empty

=8=>

<create request>(<create body>(<identifier>(undefined, nn), params)

and

entities = parentAS0ofKind(c, <agent definition> ∪ <agent type definition>).getEntities0

=>

entities ⁀

if id.refersto0 ∈ <system type definition>

then <textual typebased system definition>(

<typebased system heading>(nn, <type expression>(id, empty)))

elseif id.refersto0 ∈ <block type definition>

then <textual typebased block definition>(

<typebased block heading>(nn, <number of instances>(undefined, undefined),

<type expression>(id, empty)))

else id.refersto0 ∈ <process type definition>

then <textual typebased process definition>(nn, <number of instances>(undefined, undefined),

<type expression>(id, empty))

endif

endlet // nn

If there does not exist any instance set of the indicated agent type in the containing agent then:

a) an implicit instance set of the given type with a unique name is created in the containing agent; and

b) the <agent identifier> in the <create request> is derived syntax for this implicit instance set.

Auxiliary functions

The following function aims at finding the possible instances for an agent type create request.

possibleInstances0(c: <create request>): <agent definition>\* =def

< e in parentAS0ofKind(c, <agent definition> ∪ <agent type definition>).getEntities0:

e ∈ <agent definition> ∧ e.s-<type expression>.s-<base type> = c.s-<create body>.s-implicit >

Mapping to abstract syntax

| <create request>(c then mk-Graph-node(Mapping(c))

| <create body>(id, params) then mk-Create-request-node(Mapping(id), Mapping(params))

##### F2.2.8.13.3 Procedure call

Abstract syntax

Call-node :: [ **THIS** ] Procedure-identifier Actual-parameters

Conditions on abstract syntax

∀n ∈ Call-node ∪ Value-returning-call-node: ∀d ∈ Procedure-definition:

(d=getEntityDefinition1(n.s-Procedure-identifier,procedure)) ⇒

(isActualAndFormalParameterMatched1(n.**s**-Expression-seq, d.formalParameterSortList1)∧

(∀i ∈ 1..n.**s**-Expression-seq.length:

d.formalParameterList1[i] ∈ Inout-parameter ∪ Out-parameter⇒

n.**s**-Expression[i]∈ Identifier∧n.**s**-Expression[i].idKind1= variable))

The length of the list of optional Expressions must be the same as the number of the Procedure-formal-parameters in the Procedure-definition denoted by theProcedure-identifierand 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 orOut-parameter must be a Variable-identifier that is sort compatible to the sort identified by the   
Sort-reference-identifier of the Procedure-formal-parameter.

Concrete syntax

<procedure call> :: <procedure call body>

<procedure call body> ::

[ **this** ] {<procedure<identifier> | <procedure<type expression>} [ <actual parameters> ]

Conditions on concrete syntax

∀pc∈<procedure call>:

(**let** apl = pc.s-<procedure call body>.s-<actual parameter>-seq in

let fpl = pc.calledProcedure0.procedureFormalParameterList0 **in**

(fpl.length = apl.length) ∧

(∀i∈1..fpl.length:

(fpl[i].parentAS0.parentAS0.**s**-<parameter kind>∈{inout, out}) ⇒

(apl[i] ≠ undefined) ∧ (apl[i]∈<variable access>∪<extended primary>)) endlet)))

An <expression> in <actual parameters> corresponding to a formal **in**/**out** or **out** parameter cannot be omitted and must be a <variable access> or <extended primary>.

∀pcd∈<procedure call body>: (pcd.**s**-this ≠ undefined) ⇒

parentAS0ofKind(pcd, <internal procedure definition>) =

getEntityDefinition0(pcd.**s**-<identifier>, procedure)

If **this** is used, <procedure identifier> must denote an enclosing procedure.

Transformations

p=<procedure call body>(id, params)

provided parentAS0ofKind(id.refersto0, <agent type definition>) ≠

parentAS0ofKind(p, <agent type definition>)

=8=>

let par=parentAS0ofKind(p, <agent type definition>) in

mk-<procedure call body>(

mk-<identifier>(par.fullQualifierWithin0, id.refersto0.entityName0), // local procedure id - same name

params)

endlet // par

and // add the new definition if it does not exist

let par=parentAS0ofKind(p, <agent type definition>) in

let defs= par.s-<agent structure>.s-<entity in agent>-seq in

let procDef =

mk-<internal procedure definition>(empty, // <package use clause>\*

mk-<procedure heading>(

mk-<procedure preamble>(

mk-<type preamble>(**finalized**, undefined), //finalized, not abstract

undefined), // not exported – end procedure preamble

empty, // defined in context – no qualifier

id.refersto0.entityName0, // same name as original

empty, // no formal context parameters

undefined, // virtuality constraint

mk-<specialization>(mk-<type expression>(id,empty)), // super type, no actual context params

empty, // no additional formal parameters

id.refersto0.**s**-<procedure heading>.**s**-<result> // same result

), // procedure heading

mk-<procedure body>(undefined, empty) // no additional start, empty {state/free action} list

) // internal procedure definition

in

defs => defs ⁀ if procDef in defs then empty else < procDef > endif

endlet // procDef

endlet // defs

endlet // par

If the procedure identified by the <procedure type expression> of the <procedure call body>is not defined within the agent type enclosing the call, within the enclosing agent type there is an implicitly defined local procedure with the same name as identified by the <procedure type expression> and the call uses this local procedure. In the local procedure, identifiers of items (such as variables) external to the procedure definition are bound in the context of the original procedure definitionrather than the context of the procedure call if that is different. An implicitly defined local procedure is an inherited subtype of the procedure identified by the <procedure<type expression> of the <procedure call body>. See clause 11.13.3 *Model* of [ITU‑T Z.101].

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. See 12.3.5 *Concrete grammar* of [ITU‑T Z.104].

The keyword **this** implies that when the procedure is specialized, the <procedure identifier> is replaced by the identifier of the specialized procedure.

Further study is needed for replacement of **this** in a specialized procedure.

Mapping to abstract syntax

| <procedure call>(p then mk-Graph-node(Mapping(p))

| <procedure call body>(t, id, params) then

mk-Call-node(Mapping(t), Mapping(id), Mapping(params))

##### F2.2.8.13.4 Output

Abstract syntax

Output-node :: { Signal-identifier *Actual-parameters*

| Expression

| Encoded-expression }

Activation-delay

Signal-priority

[ Signal-destination ]

Direct-via

Activation-delay = Expression

Signal-priority = Expression

Signal-destination = { Expression | Agent-identifier | THIS } [Destination-number]

Destination-number = Expression

Direct-via = Gate-identifier-set

Encoded-expression :: Expression

Further study is needed for the first element of an Output-node being an Expression or Encoded-expression.

Further study is needed for a Signal-destination with a Destination-number.

Conditions on abstract syntax

∀ onode ∈ Output-node:

n.**s**-Identifier ≠ undefined ⇒ // Signal-identifier rather than Expression or Encoded-expression

(∃sd ∈ Signal-definition:

(sd=getEntityDefinition1(onode.**s**-Identifier,signal)) ∧

isActualAndFormalParameterMatched1(onode.**s**-Expression-seq, sd.formalParameterSortList1))

The length of the *Actual-parameters* list shall be the same as the number of *Signal-parameter* items in the *Signal-definition* denoted by the *Signal-identifier*.

Each *Expression* of the *Actual-parameters* shall be sort compatible with the *Sort-reference-identifier* of the corresponding (by position) *Signal-parameter* of the *Signal-definition.*

NOTE – The ASM code above requires both the conditions above are met.

∀ onode ∈ Output-node:

let sigid = onode.**s**-Identifier // Signal-identifier

in

let dv = onode.**s**-Identifier-set // Direct-via

in

sigid ≠ undefined ∧ dv ≠ ∅ ⇒

(∀ gi ∈ dv:

gi ∈ reachableGatesWithSignal1(

enclosingScopeGates1(parentAS1ofKind(onode, Composite-state-type-definition)), sigid)

∧ sigid ∈ getEntityDefinition1(gi, gate).**s**-Out-signal-identifier-set)

endlet

endlet

For each Gate-identifier in Direct-via, there shall exist zero or more channels such that the gate via this path is reachable with the Signal-identifier from the agent and the Out-signal-identifier-**set** of the gate shall include the Signal-identifier.

∀ onode(\*, \*, \*, sd(d,\*), \*) ∈ Output-node: // sd ∈ Signal-destination

let sigid = onode.**s**-Identifier // Signal-identifier

in

sigid ≠ undefined ∧ sd ≠ undefined ⇒

case d of

| Expression then

if staticSort1(d) = predefinedId1("Pid") then true

else

let ifd = getEntityDefinition1(staticSort1(d), sort)

in

if ifd = undefined ∨ ifd ∉ Interface-definition then false

else sigid ∈ ifd.**s**-Identifier-set

endif

endlet

endif

| Identifier then

d ∈ reachableAgentsWithSignal1(

enclosingScopeGates1(parentAS1ofKind(onode, Composite-state-type-definition)), sigid)

otherwise true

endcase

The sort of Expression of a Signal-destination shall either be the sort Pid (see clause 12.1.5), or the sort Interface-definition with the Signal-identifier in its Signal-identifier-**set**.

The Agent-identifier of a Signal-destination shall identify an agent reachable from the originating agent.

NOTE – The ASM code above requires both the conditions above are met.

∀ sigdest ∈ Signal-destination: sigdest.**s**-implicit ∈ Expression ⇒ sigdest.**s**-Destination-number = undefined

The *Destination-number* is always omitted for a *Signal-destination* that is an *Expression*.

Concrete syntax

<output> :: <output body>

<output body> :: <output body item>+ <communication constraints>

<output body item> ::

{ <signal<identifier> [<actual parameters>] | <expression output> | <encoded output> }

[<activation delay>][<signal priority>]

<expression output> :: <expression>

<encoded output> :: <expression>

<communication constraints> = { <timer communication constraint> | <destination> | <via path> }\*

<destination> :: { <pid<expression> | { <agent<identifier> | this } [<destination number>] }

<activation delay> :: < Duration<expression>

<signal priority> :: <Natural<expression>

<destination number> :: <Natural<expression>

Further study is needed for <expression output>, <encoded output> and <destination number>.

Conditions on concrete syntax

Transformations

<output>(<output body>(olist, clist))

provided < clist[i] | i in 1..clist.length] : clist[i] ∈ <destination> >.length > 1

=8=>

<output>(<output body>( olist, // list with just first dest

< clist[i] | i in 1..clist.length :

(clist[i] ∉ <destination> ) ∨ ¬(∃ j ∈ 1..clist.length : j < i ∧ clist[j] ∈ <destination>)

>

))

⁀

<output>(<output body>( olist, // list without first dest

< clist[i] | i in 1..clist.length :

(clist[i] ∉ <destination> ) ∨ (∃ j ∈ 1..clist.length : j < i ∧ clist[j] ∈ <destination>)

>

))

If <communication constraints> of an <output body> of an <output area> contains more than one **to** <destination> clause, this is a shorthand for replacing the <output area> by an <output area> sequence, one for each **to** <destination>. Each <output area> has the same original <output body item> list, but in each case the <communication constraints> contains only one **to** <destination> taken in order from the original <communication constraints>. See clause 11.13.4 *Model* of [ITU‑T Z.103].

<output>(<output body>(< o > ⁀ rest, constr)) provided rest ≠ empty =8=>

<output>(<output body>(< o >, constr)) ⁀ <output>(<output body>(rest, constr))

If there is more than one <output body item> specified in an <output body> of an <output>, the <output> is transformed into an <output> sequence each with a single <output body item> in the same order as specified in the original <output >. The <communication constraints> are repeated in each <output body>. See clause 11.13.4 *Model* of [ITU‑T Z.103].

*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) then mk-Graph-node(Mapping(o)

| <output body>(< <output body item>(id, params, delay, priority) >, constr) then

mk-Output-node(Mapping(id), Mapping(params), Mapping(delay), Mapping(priority),

Mapping(head(< c in constr: (c ∈ <destination>) >)),

{ Mapping(c in constr): c ∈ <via path> } // set of via path gate id mappings

| <destination>(d) then Mapping(d)

| <via path>(gate\_id) then Mapping(gate\_id)

| <activation delay>(delay) then

**if** delay=undefined

**then** <Duration<expression>(0)

**else** <Duration<expression>(delay)

**endif**

| <signal priority>(priority) then

**if** priority=undefined

**then** <Natural<expression>(0)

**else** < Natural<expression>(priority)

**endif**

Auxiliary functions

The function usedSignalSet0 provides the set of signal identifiers defined by the list of signal list items in the interface use lists of the given interface definition and any interface use lists of inherited interface definitions. It does not include signals defined by the interface definition or defined by inherited interface definitions.

usedSignalSet0(id: <interface definition>): Signal0 =def

{sig∈( interfaceUseList.s-<signal list item>-seq.signalSet0):

( interfaceUseList ∈ <interface use list>

∧ parentAS0ofKind (interfaceUseList, <interface definition>) = id

)}

∪

{ sig∈(interfaceUseList1.**s**-<signal list item>-seq.signalSet0):

( interfaceUseList1 ∈ <interface use list>

∧ (∃id1∈<interface definition>: (isSubtype0(id, id1))

∧ parentAS0ofKind (interfaceUseList1, <interface definition>) = id1)

)}

The function definedSignalSet0 provides the set of signal definitions in the signal definition list items of the given interface definition and any signal definitions of inherited interface definitions.

definedSignalSet0(id: <interface definition>): <signal definition>-set =def

{sd∈<signal definition>:

( (sd.parentAS0∈<signal definition list>) ∧ (sd.parentAS0.parentAS0 = id))

∨

( (∃id1∈<interface definition>: isSubtype0(id, id1)) ∧

(sd.parentAS0∈<signal definition list>) ∧ (sd.parentAS0.parentAS0 = id1))}

The function enclosingScopeGates1 is used to find the nearest Composite-state-type-definition or Agent-type-definition item with gates that encloses an Output-node, and returns a gate Identifier-set for the gates.

enclosingScopeGates1(td: Composite-state-type-definition ∪ Agent-type-definition):

Identifier-set =def

if td = undefined then ∅

else

if td.s-Gate-definition-set = ∅

then

enclosingScopeGates1(parentAS1ofKind(td,Composite-state-type-definition ∪ Agent-type-definition))

else

{ gd.identifier1 | gd ∈ td.s-Gate-definition-set }

endif

endif

The function reachableGatesWithSignal1 returns the gate Identifier-set for gates that are reachable from the given gate Identifier-set with the given signal Identifier. A gate is reachable from one of the gates given if there is a channel path with the one of the given gates as an originating gate and the gate as a destination gate. The given signal has to be conveyed by the channel. The originating gate has to be a gate defined for the origin channel endpoint and the signal has to be a valid signal in the correct direction for this gate. The destination gate has to be a gate defined for the destination channel endpoint and the signal has to be a valid signal in the correct direction for this gate.

reachableGatesWithSignal1(gset: Identifier-set, sigid: Identifier): Identifier-set =def

let gis = { gi |

cp ∈ Channel-path

∧ cp.s-Originating-gate ∈ gset

∧ gi ∈ cp.s-Destination-gate

∧ sigid ∈ cp.s-Identifier-set

∧ getEntityDefinition1(cp.s-Originating-gate, gate) ∈

case cp.s1-Channel-endpoint of

| id = Identifier then

if getEntityDefinition1(id, agent) ≠ undefined

then

getEntityDefinition1(getEntityDefinition1(aid, agent).s-Identifier,

agent type).s-Gate-definition-set

else

getEntityDefinition1(parentAS1ofKind(cp, Agent-type-definition).s-State-machine.s-Identifier,

state type).s-Gate-definition-set

endif

| ENV then

parentAS1ofKind(cp, Agent-type-definition).s-Gate-definition-set

endcase

∧ sigid ∈

case cp.s1-Channel-endpoint of

| Identifier then getEntityDefinition1(cp.s-Originating-gate, gate).s2-Identifier-set

| ENV then getEntityDefinition1(cp.s-Originating-gate, gate).s1-Identifier-set

endcase

∧ getEntityDefinition1(cp.s-Destination-gate, gate) ∈

case cp.s2-Channel-endpoint of

| id = Identifier then

if getEntityDefinition1(id, agent) ≠ undefined

then

getEntityDefinition1(getEntityDefinition1(aid, agent).s-Identifier,

agent type).s-Gate-definition-set

else

getEntityDefinition1(parentAS1ofKind(cp, Agent-type-definition).s-State-machine.s-Identifier,

state type).s-Gate-definition-set

endif

| ENV then

parentAS1ofKind(cp, Agent-type-definition).s-Gate-definition-set

endcase

∧ sigid ∈

case cp.s2-Channel-endpoint of

| Identifier then getEntityDefinition1(cp.s-Destination-gate, gate).s1-Identifier-set

| ENV then getEntityDefinition1(cp.s-Destination-gate, gate).s2-Identifier-set

endcase }

in

gset ∪ if gis = ∅ then ∅ else reachableGatesWithSignal1(gis, sigid) endif

endlet

The function reachableAgentsWithSignal1 returns the agent Identifier-set for agents that are reachable from the given gates with the given signal Identifier. The function is called with a gate Identifier-set for the gates on an agent. The paths from these gates that carry the given signal are found. The Identifier-set with the name agents is the set of agents directly reachable via these paths. If there are paths that lead to gates for environment of the agent, there can be additional agents reachable from these gates.

reachableAgentsWithSignal1(sourceGates: Identifier-set, sigid: Identifier): Identifier-set =def

let paths = {cp ∈ Channel-path:

cp.s1-Channel-endpoint ∈ sourceGates

∧ sigid ∈ cp.s-Identifier-set

∧ sigid ∈

case cp.s1-Channel-endpoint of

| Identifier then getEntityDefinition1(cp.s-Originating-gate, gate).s2-Identifier-set

| ENV then getEntityDefinition1(cp.s-Originating-gate, gate).s1-Identifier-set

endcase

∧ sigid ∈

case cp.s2-Channel-endpoint of

| Identifier then getEntityDefinition1(cp.s-Destination-gate, gate).s1-Identifier-set

| ENV then getEntityDefinition1(cp.s-Destination-gate, gate).s2-Identifier-set

endcase }

in

let agents = {a ∈ Identifier:

cp ∈ paths

∧ a ∈ cp.s2-Channel-endpoint

∧ atd ∈ Agent-type-definition

∧ a = atd.Identifier1}

in

let gates = {g ∈ Identifier:

cp ∈ paths

∧ g ∈ cp.s2-Identifier}

in

agents ∪ if gates = ∅ then ∅ else reachableAgentsWithSignal1(gates, sigid) endif

endlet

endlet

endlet

##### F2.2.8.13.5 Decision

Abstract syntax

Decision-node :: Decision-body | Any-decision

Decision-body :: Decision-question

Decision-answer-set

[ Else-answer ]

Decision-question = Expression | Informal-text

Decision-answer :: { Range-condition | Informal-text }

Transition

Else-answer :: Transition

Any-decision :: Transition-set

Conditions on abstract syntax

∀dn ∈ Decision-question: ∀r, r1 ∈ Range-condition: ∀ce,ce1 ∈ Constant-expression:

isAncestorAS1(r,ce)∧ isAncestorAS1(r1,ce1) ∧ (ce≠ce1)∧

r1. parentAS1∈ dn.**s**-Decision-answer-set ∧ r. parentAS1∈ dn.**s**-Decision-answer-set⇒

( isCompatibleTo1( ce. staticSort1, ce1. staticSort1)∨

isCompatibleTo1( ce1. staticSort1, ce. staticSort1) ) ∧

(dn.**s**-Decision-question∈Expression⇒

isCompatibleTo1( ce. staticSort1, dn.**s**-Decision-question. staticSort1))

Each *Constant-expression* of the *Range‑condition* shall be sort compatible with the sort of the *Decision-question*. If the Decision-question is an Expression, the Range-condition of each Decision-answer shall be sort compatible with the sort of the Decision-question. See clause 11.13.5 *Abstract grammar* of [ITU‑T Z.106].

Concrete syntax

<decision> :: <question> <textual decision body>

<textual decision body> :: <textual answer part>+ [<textual else part>]

<textual answer part> :: [<answer>] [<transition>]

<answer> = <range condition> | <informal text>

<textual else part> :: [<transition>]

<question> = <expression> | <informal text> | **any**

Conditions on concrete syntax

∀d∈<decision>: (d.**s**-<question> = any) ⇔

¬(∃ap∈<textual answer part>: (ap.parentAS0.parentAS0= d)) ∧ (ap.**s**-<answer> ≠ undefined) ∧

(d.s-<textual decision body>.**s**-<textual else part> = undefined)

The <answer> of <textual answer part> shall be omitted if and only if the <question> consists of the keyword **any**. In this case, a <textual else part> shall be absent. See clause 5.7.12.5 [ITU‑T Z.106].

Transformations

See clause 5.7.12.5 [ITU‑T Z.106].

<textual else part>(undefined) =5=>

<textual else part>(<transition action items>(empty, undefined))

<textual answer part>(a, undefined) =5=>

<textual answer part>(a, <transition action items>(empty, undefined))

These first two transformations are used to insert an empty transition instead of an undefined one. This empty transition will be filled with a terminator within the step below (inserting terminating actions into the transition).

t=<transition action items>(a, undefined)

provided a.last ∉ <decision> ∧ t.parentAS0.parentAS0.parentAS0 ∈ <decision> ∧

t.findContinueLabel ≠ undefined

=5=> <transition action items>(a,

<terminator>(undefined,<join>(findContinueLabel(t))))

If a <decision> is not terminating, it is derived syntax for a <decision> where all not terminating <textual answer part>s and the <textual else part> (if not terminating) have inserted at the end of their <transition> a <join> to the first <action> following the decision or (if the decision is the last <action> in a <transition string>) to the following <terminator>.

< d =<decision>(\*, \*), <action>(undefined, a) > provided ¬ terminatingDecision(d)

=5=> < d, <action>(newName, a) >

<transition action items>(str, <terminator>(undefined, t))

provided str.last ∈ <decision> ∧ ¬ str.last.terminatingDecision

=5=> <transition action items>(str, <terminator>(newName, t))

The rules above insert a new label after a non-terminating decision.

Mapping to abstract syntax

| <decision>(q, <textual decision body>(answers, elseAnswer)) then

if q≠ any

then mk-Decision-node(Mapping(q), Mapping(answers).toSet, Mapping(elseAnswer))

else mk-Any-decision(Mapping(answers))

endif

| <textual answer part>(ans, trans) then

if ans≠undefined

then mk-Decision-answer(Mapping(ans), Mapping(trans))

else mk-Transition(Mapping(trans))

endif

If ans≠undefined the <textual answer part> is the Decision-answer of a Decision-body, otherwise the <textual answer part> is the Transition of an Any-decision.

| <textual else part>(trans) then

if trans=undefined

then undefined

else mk-Else-answer(Mapping(trans))

endif

Auxiliary functions

rangeConditionList0(d:<decision>):<range condition>\*=def

let apl = d.s-<textual decision body>.s-<textual answer part> in

apl.answerPartRangeConditionList0

endlet

answerPartRangeConditionList0(apl:<textual answer part>\*):<range condition>\*=def

if apl.head.s-<answer>∈<range condition> then

apl.head.s-<answer>⁀apl.tail. answerPartRangeConditionList0

else apl.tail. answerPartRangeConditionList0

The function findContinueLabel computes the continuation label after a decision within a transition string.

findContinueLabel(x: DefinitionAS0): <name> =def

if x ∈ <transition action items> ∧ x.s-<terminator> ≠ undefined ∧

x.s-<terminator>.s-<label> = undefined ∧

x.s-<terminator>.s-<terminator node> ∈ <join>

then x.s-<terminator>.s-<terminator node>.s-<name>

else findContinueLabel(x.parentAS0)

endif

terminatingDecision(d: <decision>): Boolean =def

(∀ a ∈ d.s-<textual answer part>: terminatingTransition(a.s-<transition>) ) ∧

(d.s-<textual else part> = undefined ∨

terminatingTransition(d.s-<textual else part>.s-<transition>) )

A <decision> is a terminating decision, if each <textual answer part> and <textual else part> in its <textual decision body> is a terminating <textual answer part> or <textual else part> respectively.

terminatingTransition(t: <transition>): Boolean =def

t ∈ <terminator> ∨

t.s-<terminator node> ≠ undefined ∨

(let d= t.s-<action>.last in d ∈ <decision> ∧ terminatingDecision (d) endlet)

A <textual answer part> or <textual else part> in a decision is a terminating <textual answer part> or <textual else part> respectively if it contains a <transition> where a <terminator> is specified, or contains a <transition string> whose last <action> contains a terminating decision.

#### F2.2.8.14 Statement lists

Concrete syntax

<statements> ::

<non terminating statements> [ [ <connector name> ] <terminating statement> ]

| [ <connector name> ] <terminating statement>

<non terminating statements> = <non terminating statement>+

<non terminating statement> ::

[ <connector name> ] <statement>

| <compound statement>

| <loop statement>

| <decision statement>

<statement> =

<assignment statement>

| <set statement>

| <reset statement>

| <output statement>

| <create statement>

| <export statement>

| <call statement>

| <expression statement>

<set statement> = <set body>

<reset statement> = <reset body>

<output statement> = <output body>

<create statement> = <create body>

<terminating statement> =

<return statement>

| <break statement>

| <stop statement>

<variable definitions> = <variable definition statement>\*

<variable definition statement> :: <local variables of sort>+

<local variables of sort> :: <aggregation kind> <variable<name>+ <sort> [<expression>]

Transformations

<local variables of sort>(ak, < v > ⁀ rest, s, expr) provided rest ≠ empty =5=>

<local variables of sort>(ak, < v >, s, expr) ⁀ <local variables of sort>(ak, rest, s, expr)

A <local variables of sort> contains one or more several <variable<name> items. This is derived syntax for specifying a sequence of <local variables of sort> items, one for each <variable<name>. See clause 11.14.1 *Concrete grammar* of [ITU‑T Z.102].

<variable definition statement>(< v > ⁀ rest) provided rest ≠ empty =5=>

<variable definition statement>(< v >), <variable definition statement>(rest)

A <variable definition statement> contains one or more <local variables of sort> items. This is derived syntax for specifying a sequence of <variable definition statement> items, one for each <local variables of sort>. See clause 11.14.1 *Concrete grammar* of [ITU‑T Z.102].

Mapping to abstract syntax

| <variable definition statement>(< var >) then Mapping(var)

| <local variables of sort>(ak, < var >, s, expr) then

mk-Variable-definition(Mapping(var), Mapping(s), Mapping(ak), Mapping(expr))

##### F2.2.8.14.1 Compound statement

Abstract syntax

Compound-node :: Connector-name

Variable-definition-set

Init-graph-node\*

While-graph-node

Transition

Step-graph-node\*

Init-graph-node = Graph-node

While-graph-node = Expression\*

[ Finalization-node ]

Finalization-node = Graph-node

Step-graph-node = Graph-node

Continue-node :: Connector-name

Break-node :: Connector-name

Concrete syntax

<compound statement> :: [<connector name>] [<variable definitions>] <statements>

Conditions on abstract syntax

∀cn∈Continue-node: ∃comp∈ (Compound-node) :

isAncestorAS1(comp, cn) ∧ (cn.**s**-Connector-name = comp.s-Connector-name)

A *Continue-node* shall be contained in a *Compound-node* that has been labelled with the given *Connector-name*.

∀bn∈Break-node: ∃comp∈ (Compound-node) :

isAncestorAS1(comp, bn) ∧ (bn.**s**-Connector-name = comp.s-Connector-name)

A *Break-node* shall be contained in a *Compound-node* that has been labelled with the given *Connector-name*.

Transformations

<compound statement>(undefined,vardefs,stmts)

=5=> <compound statement>(newName,vardefs,stmts)

If the <compound statement> has a <connector name> this represents the *Connector-name*; otherwise a newly created anonymous name represents the *Connector-name*. See clause 11.14.1 *Concrete grammar* of [ITU‑T Z.102].

*The following statements are handled by the Mapping.*

A <compound statement> creates its own scope for any variables defined in the statement and the connector name.

If the <compound statement> contains <variable definitions>, the following is performed for each <variable definition statement>. A new <variable name> is created for each <variable name> in the <variable definition statement>. Each occurrence of <variable name> in the following <variable definition statement>s and within <statements> is replaced by the corresponding newly created <variable name>.

For each <variable definition statement>, a <variable definition> is formed from the <variable definition statement> by omitting the initializing <expression> (if present) and inserted as a <variable definition statement> in place of the original <variable definition statement>. If an initializing <expression> is present, an <assignment statement> is constructed for each <variable name> mentioned in the <local variables of sort> in the order of their occurrence, where <variable name> is given the result of <expression>. These <assignment statement> items are inserted at the front of <statement> items in the order of their occurrence.

If the <statements> list does not end in a <terminating statement>, the *Transition* is a *Graph-node* list represented by the <non terminating statements> of the <statements> list followed by a *Break-node* with the *Connector-name* as the *Terminator* of the *Transition*. If the <statements> list ends in a <terminating statement>, the *Transition* is a *Graph-node* list represented by the <non terminating statements> of the <statements> list followed by the *Terminator* represented by the <terminating statement>.

If the <statements> list is omitted, the *Transition* is an empty *Graph-node* list followed by a *Break-node* with the *Connector-name* as the *Terminator* of the *Transition*.

Mapping to abstract syntax

| <compound statement>( cn, vars, allstats = stats ⁀ < laststatement >) then

mk-Compound-node(Mapping(cn), Mapping(vars).toSet, undefined, empty,

if laststatement ∈ <terminating statement>

then mk-Transition(Mapping(stats), Mapping(laststatement))

else mk-Transition(Mapping(allstats), mk-Break-node(Mapping(cn)))

endif,

empty)

| <compound statement>( cn, vars, empty) then

mk-Compound-node(Mapping(cn), Mapping(vars).toSet, undefined, empty,

mk-Transition(Mapping(empty), mk-Break-node(Mapping(cn))), empty)

##### F2.2.8.14.2 Transition actions and terminators as statements

Concrete syntax

<assignment statement> :: <assignment>

<return statement> :: <return body>

<call statement> :: <procedure call body> | <remote procedure call body>

Conditions on concrete syntax

∀rs∈<return statement>:

(parentAS0ofKind(rs, <internal procedure definition>)≠undefined)∨

(parentAS0ofKind(rs,<operation definition>)≠undefined)

A <return statement> is only allowed within an <internal procedure definition> or within an <operation definition>.

Transformations

*The following statements are handled by the Mapping.*

<assignment statement> is transformed into the <task>

task <assignment>;

A <call statement> is derived syntax for <procedure call> and is transformed into a <procedure call> with the same <procedure call body>:

call <procedure call body>;

The transform of an <algorithm action statement> and a <return statement> is obtained by dropping the trailing <end>.

Mapping to abstract syntax

| <assignment statement>(a) then Mapping(a)

| <return statement>(r) then Mapping(r)

| <call statement>(c) then Mapping(c)

##### F2.2.8.14.3 Expressions as statements

Concrete syntax

<expression statement> :: <operator application>

Transformations

let nn=newName in

<expression statement>(expr) =3=>

<compound statement>(

< <variable definition statement>

(< <local variables of sort>(< nn >, expr.staticSort0, undefined) >) >,

< <assignment statement>(<assignment>(<identifier>(undefined, nn), expr)) >

)

endlet // nn

A new <variable name> is created. A <variable definition> is constructed that declares the newly created <variable name> to be of the same sort as the result of <operation application>. Finally, the expression statement is transformed to a <compound statement> consisting of the newly constructed <variable definition>, followed by an <assignment> between the variable with <variable name> and the <operation application>.

##### F2.2.8.14.4 If statements

Concrete syntax

<if statement> ::

[<connector name>] <Boolean<expression> <consequence statement> [<alternative statement>]

<consequence statement> = [ <non terminating statement> ]

<alternative statement> = <non terminating statement> | <terminating statement>

<loop if statement> ::

[<connector name>]

<Boolean<expression> <loop consequence statement> [<loop alternative statement>]

<loop consequence statement> = [ <statement in loop> | <loop terminating statement>]

Transformations

<if statement>( connectorname, expr, cons, alt) =3=>

<decision statement>(connectorname, expr,

<decision statement body>(<

<algorithm answer part>("true", cons),

if alt ≠ undefined then <algorithm else part>(alt) endif

>, undefined))

<loop if statement>( connectorname, expr, cons, alt) =3=>

<loop decision statement>(connectorname, expr,

<decision statement body>(<

<algorithm answer part>("true", cons),

if alt ≠ undefined then <algorithm else part>(alt) endif

>, undefined))

The <if statement> (or <loop if statement>) is transformed to the following <decision statement> (or <loop decision statement> respectively):

decision *boolean expression* {

( true ): consequence statement

else : alternative statement

};

where *boolean\_expression*, *consequence\_statement* and *alternative\_statement* represent actual text for the <Boolean expression>, <consequence statement> (<loop consequence statement> respectively) and <alternative statement> (<loop alternative statement> respectively).

##### F2.2.8.14.5 Decision statements

Concrete syntax

<decision statement> ::

[ <connector name> ] <question> <decision statement body>

| <if statement>

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

<algorithm answer part> ::<answer> [<non terminating statement> | <terminating statement>]

<algorithm else part> :: <alternative statement>

<loop decision statement> ::

[ <connector name> ] <question> <loop decision statement body>

| <loop if statement>

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

<loop answer part> ::<answer> { <statement in loop> | <loop terminating statement> }

<loop else part> :: <loop alternative statement>

<loop alternative statement> = <statement in loop> | <loop terminating statement>

A <loop decision statement> differs from a <decision statement> only because it uses <loop decision statement body> instead of <decision statement body> (or <loop if statement> instead of <if statement>) so that <loop break statement> and <loop continue statement> are allowed in the loop. Similarly <loop answer part>, <loop else part>, <loop alternative statement>, <statement in loop> and <loop terminating statement> are used instead of <algorithm answer part>, <algorithm else part>, <alternative statement>, <non terminating statement> and <loop terminating statement>, respectively.

Mapping to abstract syntax

| <decision statement>( connectorname, ques, body) then

Mapping(<loop decision statement>( connectorname, ques, body))

The main difference between a <decision statement> and a <loop decision statement> is the inclusion of <loop continue statement> and <loop break statement>, but these are syntactically excluded from <decision statement>, therefore the *Mapping* for <loop decision statement> is used here.

Each <algorithm answer part> of a <decision statement> represents a *Decision-answer* of the *Decision-node* of the *Transition* of the *Compound-node*. If the <algorithm answer part> is a <terminating statement>, the *Transition* of the *Decision-answer* is the *Terminator* represented by the <terminating statement>. If there is no <non terminating statement> or <terminating statement> in the <algorithm answer part>, the *Transition* of the *Decision-answer* is an empty *Graph-node* list followed by a *Break-node* with the *Connector-name* of the *Compound-node*. If the <algorithm answer part> is a <non terminating statement>, the *Transition* of the *Decision-answer* is the *Graph-node* represented by the <non terminating statement> followed by a *Break-node* with the *Connector-name* of the *Compound-node*.

An <algorithm else part> of a <decision statement> represents the *Else-answer* of the *Decision-node* of the *Transition* of the *Compound-node*. If the <alternative statement> is a <terminating statement>, the *Transition* of the *Else-answer* is the *Terminator* represented by the <terminating statement>. If there is no <alternative statement> or no <algorithm else part> in the <decision statement> the *Transition* of the *Else-answer* contains only a *Break-node* with the *Connector-name* of the *Compound-node*. If the <alternative statement> is a <non terminating statement>, the *Transition* of the *Else-answer* is the *Graph-node* represented by the <non terminating statement>.

The mapping of <algorithm answer part> and <algorithm else part> is included in the mapping of <decision statement>. The <if statement> alternative of <decision statement> is removed by a transformation in F2.2.8.14.4 If statements.

| <loop decision statement>( connectorname, ques, <loop decision statement body>(answers, elsePart)) then

let cn= if connectorname = undefined then newName else connectorname endifin

mk-Compound-node(Mapping(cn), ∅, empty,

mk-Transition(

mk-Decision-node(

mk-Decision-body(

Mapping(ques),

{ mk-Decision-answer(

Mapping(ans),

if stat(cc) ∈ <loop continue statement>

then mk-Transition(empty,

if cc = undefined

then mk-Continue-node(cn)

else mk-Continue-node(cc)

endif)

elseif stat ∈ <loop terminating statement>

then mk-Transition(empty, Mapping(stat))

elseif stat = undefined

then mk-Transition(empty, mk-Break-node(cn))

else mk-Transition(Mapping(stat), mk-Break-node(cn))

endif)

| aap(ans,stat) in answers },

mk-Else-answer(

if elsePart ∈ <loop terminating statement>

then mk-Transition(empty, Mapping(elsePart))

elseif elsePart = undefined

then mk-Transition(empty, mk-Break-node(cn))

else mk-Transition(Mapping(elsePart), mk-Break-node(cn))

endif

)

)

)

)

)

endlet

A <loop decision statement> represents a *Compound-node* in the same way as <decision statement> except that a *Transition* of a *Decision-answer* or *Else-answer* has a *Break-node* for a <loop break statement> or a *Continue-node* for a <loop continue statement>. The mapping of <loop answer part> and <loop else part> is included in the mapping of <loop decision statement>. The <loop if statement> alternative of <loop decision statement> is removed by a transformation in in F2.2.8.14.4 If statements.

##### F2.2.8.14.6 Loop statements

Concrete syntax

<loop statement> ::

[ <connector name> ] <loop clause>\* <loop body statement> [<finalization statement>]

<loop body statement> = <statement in loop>

<statement in loop> =

<statement>

| <loop statement>

| <loop compound statement>

| <loop decision statement>

<loop compound statement> ::

[ <connector name> ] <variable definitions> <loop statements>

A <loop compound statement> differs from a <compound statement> because it uses   
<loop statements> instead of <statements>.

<loop statements> ::

<statement in loop>+ [ <loop terminating statement> ]

| <loop terminating statement>

A <loop statements> list differs from a <statements> because it optionally ends with a   
<loop terminating statement> instead of <terminating statement>, and also because it has   
<statement in loop> instead of <statement>.

<loop terminating statement> =

<terminating statement>

| <loop break statement>

| <loop continue statement>

A <loop terminating statement> allows <loop break statement> and <loop continue statement> as well as <terminating statement>.

<finalization statement> = <statement> | <compound statement>

<loop clause> ::

[<loop variable indication>] [<Boolean<expression>] <loop step>

<loop step> = [<expression> | <procedure call body>]

<loop variable indication> =

<loop variable definition> | <loop variable indication identifier>

<loop variable indication identifier> :: <variable<identifier> [<expression>]

<loop variable definition> :: <aggregation kind> <variable<name> <sort> <expression>

<loop break statement> :: ()

<loop continue statement> :: [<connector<name>]

Conditions on concrete syntax

∀lc∈<loop clause>:

**if** = lc.s-<loop step>∈<expression> ∪ <procedure call body> **then**

**then** lc.s-<loop variable indication> ≠ undefined

end**if**

If a <loop step> of a <loop clause> has an <expression> or <procedure call body> the <loop variable indication> of the <loop clause> shall not be omitted*.*

∀ls∈<loop step>:

(**let** pd = ls.s-<procedure call body>.calledProcedure0 **in**

pd ≠undefined∧pd.s-<procedure heading>.**s**-<procedure result> = undefined

endlet)

The <procedure identifier> in the <procedure call body> of a <loop step> must not refer to a value returning procedure call.

Transformations

l=<loop statement> ∧ l.s-<connector name> = undefined =3=>

l.s-<connector name> = newName

Generate a name for every unlabelled <loop statement>.

Mapping to abstract syntax

| l=<loop statement>(cn, lc, body, final) then

mk-Compound-node(Mapping(cn),

// *Variable-definition-set* of the *Compound-node*

{ mk-Variable-definition(

Mapping(c.s-<loop variable indication>.s-<name>),

Mapping(c.s-<loop variable indication>.s-<sort>),

PART,

undefined) // Variable definition

| c ∈ lc.toSet ∧ c.s-<loop variable indication> ∈ <loop variable definition>

}, // *Variable-definition-set* of the *Compound-node*

// *Init-graph-node* list of the *Compound-node*

< let lvi=lc.s-<loop variable indication> in

if lvi ∈ <loop variable definition>

then mk-Graph-node(mk-Task-node(

mk-Assignment(

Mapping(mk-<identifier>(<mk-<path item>(composition, cn)>,lvi.s-<name>)),

Mapping(lvi.s-<expression>))

))

elseif // lvi ∈ <loop variable definition indicator>

lvi.s-<expression> ≠ undefined

then mk-Graph-node(mk-Task-node(

mk-Assignment(Mapping(lvi.s-<identifier>), Mapping(lvi.s-<expression>))

))

endif

endlet

| c in lc: c.s-<loop variable indication> ≠ undefined >,

// *While-graph-node* of the *Compound-node*

mk-While-graph-node(

< mk-Expression(c.s-<expression>) | c in lc: c.s-<expression> ≠ undefined >,

// *Finalization-node* of the *Compound-node*

if final ≠ undefined then mk-Graph-node(Mapping(final)) endif

),

// *Transition* of the *Compound-node*

if body(cc) ∈ <loop continue statement>

then mk-Transition(empty,

if cc = undefined

then mk-Continue-node(Mapping(cn))

else mk-Continue-node(Mapping(cc))

endif)

elseif body ∈ <loop terminating statement>

then mk-Transition(empty, Mapping(body))

elseif body = undefined

then mk-Transition(empty, mk-Continue-node(Mapping(cn)))

else mk-Transition(< Mapping(body) >, mk-Continue-node(Mapping(cn)))

endif,

// *Step-graph-node* of the *Compound-node*

< let lvi=c.s-<loop variable indication> in

let ls=c.s-<loop step> in

if lvi ∈ <loop variable definition>

then mk-Graph-node(mk-Task-node(

mk-Assignment (

Mapping(mk-<identifier>(<mk-<path item>(composition, cn)>,lvi.s-<name>),

Mapping(ls))

))

else mk-Graph-node(mk-Task-node(

mk-Assignment (Mapping(lvi.s-<identifier>), Mapping(ls))

))

endif

endlet

endlet

| c in lc: c.s-<loop step> ≠ undefined >

) // end of *Compound-node* for <loop statement>

A <loop statement> represents a *Compound-node*. If the <loop statement> has a <connector name> this represents the *Connector-name*; otherwise a newly created anonymous name represents the *Connector-name*.

Each <aggregation kind> <variable name> <sort> set in a <loop variable definition> represents the *Aggregation-kind*, *Variable-name* and *Sort-reference-identifier* of an element of the   
*Variable-definition****-set*** of the *Compound-node*. Each <variable name> and associated <expression> in a <loop variable definition> also represents the *Variable-identifier* and *Expression* of an *Assignment* in a *Task-node* of the *Init-graph-node* list of the *Compound-node*. The <variable name> in a <loop variable definition> also represents the *Variable-identifier* in any *Step-graph-node* for the <loop step> following the <loop variable indication>*.*

Each <variable identifier> and associated <expression> in a <loop variable indication> represents the *Variable-identifier* and *Expression* of an *Assignment* in a *Task-node* of the *Init-graph-node* list of the *Compound-node*. The <variable identifier> in a <loop variable indication> also represents the *Variable-identifier* in the *Step-graph-node* for the <loop step> following the <loop variable indication>. The <variable identifier> in the <loop variable indication> also represents the   
*Variable-identifier* in any *Step-graph-node* for the <loop step> following the <loop variable indication>.

The *Task-node* items from the <loop variable indication> items occur in the *Init-graph-node* list in the same order (left to right, top to bottom) as the <loop clause> items in the <loop statement>.

Each <Boolean expression> in a <loop clause> represents a Boolean *Expression* in the *While-graph-node* of the *Compound-node* for the <loop statement> in the order of each <loop clause> in the <loop statement>.

A <loop body statement> represents the *Transition* of the *Compound-node*. The *Transition* is the *Graph-node* represented by the <statement in loop> (a <non terminating statement> or   
<loop statement> or <loop compound statement> or <loop decision statement>) followed by a *Continue-node* with the *Connector-name* of the *Compound-node*.

Each <expression> or <procedure call body> of a <loop step> represents an *Expression* of an *Assignment* in a *Task-node* of the *Step-graph-node* list of the *Compound-node*. The *Variable-identifier* of the *Assignment* is the one represented by the <variable identifier> or   
<variable name> of the preceding <loop variable indication>*.*

The *Task-node* items from the <loop step> items occur in the *Step-graph-node* list in the same order (left to right, top to bottom) as the <loop clause> items in the <loop statement>.

The mapping of <finalization statement>, <loop clause>, <loop step>, <loop variable indication>, <loop variable indication identifier> and <loop variable definition> is included in the mapping of <loop statement>.

| <loop compound statement>( cn, vars, allstats = stats ⁀ < laststatement >) then

mk-Compound-node(Mapping(cn), Mapping(vars).toSet, undefined, empty,

if laststatement ∈ <loop terminating statement>

then mk-Transition(Mapping(stats), Mapping(laststatement))

else mk-Transition(Mapping(allstats), mk-Break-node(Mapping(cn)))

endif,

empty)

A <loop compound statement> represents a *Compound-node* in the same way as a <compound statement> except that the *Transition* represented by <loop statements> optionally ends with a   
*Break-node* for a <loop break statement> or a *Continue-node* for a <loop continue statement>.

| lbs = <loop break statement> then

mk-Break-node(Mapping(parentAS0ofKind(lbs, <loop statement>).s-<connector name>))

A <loop break statement> represents a *Break-node* with the *Connector-name* of the immediately enclosing *Compound-node* for a <loop statement>.

| lcs = <loop continue statement>(cn) then

if cn = undefined

then mk-Continue-node(Mapping(parentAS0ofKind(lcs, <loop statement>).s-<connector name>))

else mk-Continue-node(Mapping(cn))

endif,

A <loop continue statement> without a <connector name> represents a *Continue-node* with the *Connector-name* of the immediately enclosing *Compound-node* for a <loop statement>. A   
<connector name> in a <loop continue statement> represents the *Connector-name* of the *Continue-node*.

##### F2.2.8.14.7 Break statement

Conditions on abstract syntax

∀bs∈Break-node: ∃ls∈ (Compound-node) :

isAncestorAS1(ls, bs) ∧ (bs.**s**-Connector-name = ls.s-Connector-name)

A *Break-node* shall be contained in a *Compound-node* that has been labelled with the given *Connector-name*.

Concrete syntax

<break statement> ::<connector<name>

Mapping to abstract syntax

| <break statement>(name) then mk-Break-node(Mapping(name))

#### F2.2.8.15 Timer

Abstract syntax

Timer-definition :: Timer-name

Sort-reference-identifier\*

[ Timer-default-initialization ]

Timer-default-initialization = Constant-expression

Set-node :: Time-expression Timer-identifier Expression\*

Time-expression = Expression

Reset-node :: Timer-identifier Expression\*

Conditions on abstract syntax

∀n ∈ Set-node ∪ Reset-node: ∀d ∈ Timer-definition:

(d=getEntityDefinition1(n.**s**-Timer-identifier,timer)) ⇒

isActualAndFormalParameterMatched1(n.**s**-Expression-seq, d.formalParameterSortList1)

The sorts of the list of 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 list> ] [<timer default initialization>]

<timer default initialization> :: <Duration<constant expression>

<set> :: <set body>

<set body> :: <set clause>+

<set clause> :: [<Time<expression>] <timer<identifier> <expression>\*

<reset> :: <reset body>

<reset body> :: <reset clause>+

<reset clause> :: <timer<identifier> <expression>\*

Transformations

<set clause>(undefined, id, exprList)

=8=> <set clause>

(<operator application>("+", now, id.refersto0.s-<timer default initialization>), id, exprList)

A <set clause> with no <Time expression> is derived syntax for a <set clause> where   
<Time expression> is

now + <Duration constant expression>

where <Duration constant expression> is derived from the <timer default initialization> in timer definition.

<set>(<set body>(< s > ⁀ rest)) provided rest ≠ empty =5=>

<set>(<set body>(< s >)) ⁀ <set>((<set body>(rest))

A <set> is allowed to contain several <set clause> items. This is derived syntax for specifying a sequence of <set> items, one for each <set clause> such that the original order in which they were specified in the <set> is retained.

<set statement>(<set body>(< s > ⁀ rest)) provided rest ≠ empty =5=>

<set statement>(<set body>(< s >)) ⁀ <set statement>((<set body>(rest))

A <set statement> is allowed to contain several <set clause> items. This is derived syntax for specifying a sequence of <set statement> items, one for each <set clause> such that the original order in which they were specified in the <set> is retained.

<reset>(<reset body>(< r > ⁀ rest)) provided rest ≠ empty =5=>

<reset>(<set body>(< r >)) ⁀ <reset>(<set body>(rest)) >

A <reset> is allowed to contain several <reset clause> items. This is derived syntax for specifying a sequence of <reset> items, one for each <reset clause> such that the original order in which they were specified in the <reset> is retained.

<reset statement>(<reset body>(< s > ⁀ rest)) provided rest ≠ empty =5=>

<reset statement>(<reset body>(< s >)) ⁀ <reset statement>((<reset body>(rest))

A <reset statement> is allowed to contain several <reset clause> items. This is derived syntax for specifying a sequence of <reset statement> items, one for each <reset clause> such that the original order in which they were specified in the <reset> is retained.

The shorthand items for <set>, <set statement>, <reset> and <reset statement> are expanded before shorthand items in the contained expressions are expanded. The shorthand items are in clause 11.5 *Model* of [ITU‑T Z.103].

<timer definition>(< t > ⁀ rest) provided rest ≠ empty =5=>

<timer definition>(< s >) ⁀ <timer definition>(rest)

A <timer definition> is allowed to contain several <timer definition item>s. This is derived syntax for specifying a sequence of <timer definitions>s, one for each <timer definition item>. See clause 11.5 *Model* of [ITU‑T Z.103].

Mapping to abstract syntax

| <timer definition>(< item >) then Mapping(item)

| <timer definition item>(name, sortList, \*) then

mk-Timer-definition(Mapping(name), Mapping(sortList))

| <set>(< clause > then mk-Graph-node(Mapping(clause))

| <set clause>(expr, id, params) then mk-Set-node(Mapping(expr), Mapping(id), Mapping(params))

| <reset>(< clause > then mk-Graph-node(Mapping(clause))

| <reset clause>(id, params) then mk-Reset-node(Mapping(id), Mapping(params))

### F2.2.9 Data

#### F2.2.9.1 Data definitions

Concrete syntax

<data definition> =

<entity in data type> | <interface definition>

<sort> ::

<basic sort> [ <range condition> ]

| <anchored sort>

| <pid sort>

| <inline data type definition>

| <inline syntype definition>

<basic sort> =

<datatype<type expression>

| <as signal>

| <as interface>

| <as channel>

| <as gate>

| <syntype>

<anchored sort> :: { this | parent } [ <basic sort> ]

<pid sort> = <sort<identifier>

<inline data type definition> :: [ <data type specialization> ] <data type definition body>

<inline syntype definition> :: <basic sort> { <default initialization> [<constraint> ] | <constraint> }

Conditions on concrete syntax

∀s∈<sort>:

(∀bsd = getEntityDefinition0(s.s-<basic sort>.s-<identifier>, sort): s.s-<basic sort> ∈ <type expression>)

∧

(∀rc = s.s-<range condition>:(rc ≠ undefined) )

⇒

bsd.s-<data type definition body>.s-implicit ∈ <literal list>

∨ isSizeConstraintValid (bs)

A <range condition> after a <basic sort> of a sort is only valid if the <basic sort> has been constructed using the literal data type constructor and the **constants (**<range condition>**)** is valid as a <constraint> for the <basic sort>, or if the **size (**<range condition>**)** is valid as a <constraint> for the <basic sort>. See clause 12.1 *Concrete grammar* of [ITU‑T Z.104].

Transformations

s = < sort>(anchSort = <anchored sort>)

=14=>

mk-< sort>(

mk-<type expression>(sortId0(s),empty),

undefined) // range condition omitted

An <anchored sort> without a <basic sort> is a shorthand for specifying a <basic sort> with the name of the data type definition (or syntype definition) in the context where the <anchored sort> occurs. See clause 12.1 *Model* of [ITU‑T Z.104].

Every occurrence of an <anchored sort> of the form **parent** T in a specialization of T, the <anchored sort> is replaced by an unambiguous qualified <basic sort> for the base type T: that is, the qualified <identifier> for T. Every occurrence of an <anchored sort> of the form **this** T in a specialization of T, the <anchored sort> is replaced by the <basic sort> of the subsort: that is, the <identifier> for the subsort ST. See clause 12.1.9 *Model* of [ITU‑T Z.104].

let nn = newName in

s = <sort>(<type expression>(bsort, empty), rc)

=8=>

mk-< sort>(mk-<syntype<identifier>( s.surroundingScopeUnit0.fullQualifierWithin0, nn),undefined)

and // add the implicit syntype definition

=>

let parent = s.surroundingScopeUnit0 in entities=parent.getEntities0 in

entities ⁀

mk-<syntype definition>(empty,

mk-<syntype definition syntype>(nn,

bsort,

undefined,

if getEntityDefinition0(bsort, sort).s-<data type definition body>.s-implicit ∈ <literal list>

then rc

else mk-<size constraint>(rc)

endif

) )

endlet // parent

endlet // nn

A <sort> that is a <basic sort> with a <range condition> is derived concrete syntax for a <syntype> of an implied <syntype definition> having an anonymous name for the <syntype name> and the <basic sort> as the <parent sort identifier>. This anonymously named <syntype definition> is defined with no <default initialization> and with its elements restricted by a <constraint> based on the <range condition>. If the <basic sort> has been constructed using the literal data type constructor the <range condition> is transformed to a <constraint> of the <syntype definition> that is **constants (** <range condition> **)**; otherwise, the <range condition> is transformed to a <constraint> of the <syntype definition> that is **size (** <range condition> **).** See clause 12.1 *Model* of [ITU‑T Z.104].

NOTE – The conditions on concrete syntax described below (in section F2.2.9.8.4 Constraint) apply to the <syntype> and its <constraint> based respectively on <basic sort> and <range condition>.

let nn = newName in

s = <inline data type definition>(dtspecial, dtdbody)

=2=>

let parent = s.surroundingScopeUnit0 in

mk-<type expression>( mk-<identifier>( parent.fullQualifierWithin0,nn), empty)

and // add the implicit data type definition

entities=parent.getEntities0

=>

entities ⁀ mk-<data type definition>(empty, undefined, nn, dtspecial, dtdbody, undefined)

endlet // parent

endlet // nn

An <inline data type definition> is derived concrete syntax for a <basic sort> of an implied <data type definition> having an anonymous name. This anonymously named <data type definition> is derived from the <inline data type definition> by inserting **type** and the anonymous name after **value** in the <inline data type definition>. Each <inline data type definition> defines a different implied <data type definition>. See clause 12.1 *Model* of [ITU‑T Z.104].

<inline syntype definition>(bsort, x )

=2=>

case x of

| <default initialization> then mk-<inline syntype definition>(bsort, x, undefined)

| <constraint> then mk-<inline syntype definition>(bsort, undefined, x)

endcase

let nn = newName in

s = <inline syntype definition>( bsort, init, constr)

=2=>

mk-<syntype<identifier>( s.surroundingScopeUnit0.fullQualifierWithin0, nn)>

and // add the implicit syntype definition

let parent = s.surroundingScopeUnit0 in

entities=parent.getEntities0

=>

entities ⁀ mk-<syntype definition>(empty, mk-<syntype definition syntype>(nn, bsort, init, constr))

endlet // parent

endlet // nn

An <inline syntype definition> is derived concrete syntax for a <basic sort> of an implied <syntype definition> having an anonymous name. This anonymously named <syntype definition> is derived from the <inline syntype definition> by inserting the anonymous name and <equals sign> after syntype in the <inline syntype definition>. See clause 12.1 *Model* of [ITU‑T Z.104].

Auxiliary functions

The function parentSortId0 gets the <identifier> of the parent data type definition of a sort, which is the same as the given sort except if it as a syntype, in which case it returns the identity of the data type definition on which the syntype is based.

parentSortId0(s: <sort<identifier>): <sort <identifier>=def

getEntityDefinition0(s, type).derivedDataType0.identifier0

The function sortId0 gets the <identifier> of a <sort>. Any <inline data type definition> or <inline syntype definition> is replaced at transformation step 2 by a <basic sort> or <syntype> (respectively), therefore these would not be offered as a parameter to sortId0.

sortId0(s: <sort>): <identifier>=def

case s of

| <type expression> then *s*.s-<type expression>.s-<identifier>

| <as signal>(sigId) then sigId.asSignal

| <as interface> then undefined

| <as channel> then undefined

| <as gate>(gid) then

if gid.refersto0.asGateName ≠ undefined

then mk-<type expression>(mk-<identifier>(gid.**s**-<qualifier>, gid.refersto0.asGateName), undefined)

else undefined

endif

| anchSort = <anchored sort> then

let parentDataTypeDef = parentAS0ofKind(anchSort, <data type definition>) in

let basicSort = anchSort.s-<type expression>.s-<identifier> in

case anchSort.s-implicit of

| this then

if basicSort = undefined ∨ isSubSort0(basicSort, parentDataTypeDef.identifier0)

then parentDataTypeDef.identifier0 // change to subsort id if basic sort undefined or supersort

else basicSort // otherwise specified basic sort id

endif

| parent then

if basicSort = undefined

then // if basic sort undefined

if parentDataTypeDef*.specialization0* ≠ undefined // if a specialized sort

then parentDataTypeDef*.specialization0*.s-<type expression>.s-<identifier> // sort id of supersort

else parentDataTypeDef.identifier0 // otherwise the sort id of the enclosing data type

endif

elseif isSubSort0(basicSort, parentDataTypeDef.identifier0) // basic sort defined and supersort

then basicSort // if enclosing data type subsort of basic sort, change to basic sort id

else parentDataTypeDef.identifier0 // otherwise the sort of the enclosing data type

endif

endcase

endlet // basicSort

endlet // parentDataTypeDef

| <identifier> then s.s-<identifier> // pid sort or syntype

| <inline data type definition> then undefined

| inlsd = <inline syntype definition> then inlsd.s-<basic sort>.sortId0

otherwise undefined

endcase

Further study is needed for <as channel>, <as interface> and <inline data type definition>.

The function isSizeConstraintValid determines if the range condition is a valid size constraint for the data type definition.

A <size constraint> shall only be used in the concrete syntax of a <constraint> if Length has been defined as an operation with the <operation signature>:

length (**in** P ) -> <<**package** Predefined>>Integer;

Where P is the sort of the syntype for the context of the <constraint>. See clause 12.1.8.2 *Concrete grammar* of [ITU‑T Z.101].

isSizeConstraintValid(dtd: <data type definition>): Boolean =def

(∃ opSig ∈ <operation signature>:

opSig.s-<name> = mk-<name>("length")

∧ opSig.s-<arguments>.length = 1

∧ opSig.s-<arguments>[1].s-<formal parameter> =

mk-<formal parameter> (in, mk-<sort>(mk-<type expression>(dtd.identifier0,empty)))

∧ opSig.s-<result> = mk-<sort>(mk-<type expression>( predefinedId0("Integer"),empty)

)

#### F2.2.9.2 Data type definition

Abstract syntax

Data-type-definition = Value-data-type-definition

| Interface-definition

Value-data-type-definition :: Sort

[ Data-type-identifier ]

Literal-signature-set

Null-literal-signature

Static-operation-signature-set

Dynamic-operation-signature-set

Procedure-definition-set

Data-type-definition-set

Syntype-definition-set

Variable-definition-set

[ Default-initialization ]

[ Abstract ]

Concrete syntax

<data type definition> ::

<package use clause>\* <type preamble> <data type heading> [<data type specialization>]

[ <data type definition body> ]

<data type heading> ::

<data type<name> [ <formal context parameters> ] [<virtuality constraint>]

<data type definition body> ::

<entity in data type>\* [<data type constructor>] <operations> [<default initialization>]

<entity in data type> =

<data type definition> | <syntype definition> | <synonym definition>

<operations> :: <operation signatures> <operation definitions>

<operation definitions> :: <operation definition item>\*

Conditions on concrete syntax

∀dtd∈<data type definition>:∀fcp∈ <formal context parameter>:

fcp∈dtd.localFormalContextParameterList0.toSet ⇒

fcp∈<sort context parameter> ∪<synonym context parameter list>

A <formal context parameter> of <formal context parameters> must be either a <sort context parameter> or a <synonym context parameter list>.

∀anchSort∈<anchored sort>:

parentAS0ofKind(anchSort, <data type definition>) ≠ undefined

An <anchored sort> is legal concrete syntax only if it occurs within a <data type definition>. The <basic sort> in the <anchored sort> shall name the <sort> introduced by an enclosing <data type definition>. See clause 12.1 *Model* of [ITU‑T Z.104].

∀sid∈<pid sort>:isPidSort0(sid)

The <sort<identifier> in a <pid sort> shall identify a pid sort. See clause 12.1 *Concrete grammar* of [ITU‑T Z.101].

Mapping to abstract syntax

| dtd=<data type definition>(\*, // package use clause

tp, // type preamble

<data type heading>( name, \*, \*),

base, // data type specialization

body) // data type definition body

then

let bodyMapping = Mapping(body)

in

mk-Value-data-type-definition(Mapping(name), Mapping (base),

{ ls ∈ bodyMapping : ls ∈ Literal-signature },

mk-Literal-signature(mk-Name("null"), //null literal signature - literal name

mk-Result(mk-Identifier(Mapping(dtd.fullQualifier0), Mapping(name)), REF)

\*), //null literal signature - literal natural – arbitrary value as not used

{ so ∈ bodyMapping : so ∈ Static-operation-signature },

{ do ∈ bodyMapping : do ∈ Dynamic-operation-signature },

{ pr ∈ bodyMapping : pr ∈ Procedure-definition },

{ dt ∈ bodyMapping : dt ∈ Data-type-definition },

{ sy ∈ bodyMapping : sy ∈ Syntype-definition},

{ vd ∈ bodyMapping : vd ∈ Variable-definition },

if body.s-<default initialization> ≠ undefined

then Mapping(body.s-<default initialization>.s-<operand5>)

else undefined

endif,

if tp.s-<abstract> ≠ undefined then mk-Abstract else undefined endif

) // *Value-data-type-definition*

endlet // bodyMapping

| body = <data type definition body>(entities, ctor, operations, init)

then

case ctor of

| <literal list> then

{ Mapping(entities[i]) | i ∈ 1..entities.length } ∪ Mapping(ctor) ∪ Mapping(operations)

| <structure definition>(visi, fields) then

let d= parentAS0ofKind(body, <data type definition>) in

Mapping(

mk-<data type definition body>(

entities, undefined,

mk-<operations>(

mk-<operation signatures>(

mk-<operator list> (

d.getOpSigs0⁀d.structMakeSig0⁀d.structOpSigs0

), // operator list

empty // method list – changed to operators and included in operators list

), // operation signatures

operations.s-<operation definitions>⁀ d.structMakeDef0⁀ d.structOpDefs0

), // operations

init // default initialization

) // data type definition body

) // Mapping

endlet // d

| <choice definition>(visi, choices) then

let nn= newName in // name for anonPresent

let d= parentAS0ofKind(body, <data type definition>) in

Mapping(

mk-<data type definition body>(

entities⁀ anonLiterals0(ctor, nn), undefined,

mk-<operations>(

mk-<operation signatures>(

mk-<operator list>( d.getOpSigs0⁀d.choiceOpSigs0(d, nn) )

empty // method list – changed to operators and included in operators list

), // operation signatures

operations.s-<operation definitions>⁀ choiceOpDefs0(d, nn)

), // operations

init // default initialization

) // data type definition body

) // Mapping

endlet // d

endlet // nn

otherwise

{ Mapping(entities[i]) | i ∈ 1.. entities.length } ∪ Mapping(operations) // ctor is undefined

endcase

The parameter *ctor* of the <data type definition body> is an optional <data type constructor>, which is a <literal list>, a <structure definition> or a <choice definition>.

| <operations>( operationsignatues, operationdefinitions)

then

Mapping(operationsignatures) ∪ Mapping(operationdefinitions)

| <operation definitions>( operationdefinitionitems)

then

{ Mapping(operationdefinitionitems[i]) | i ∈ 1..operationdefinitionitems.length }

The following paragraphs are from clause 12.1.2 *Concrete grammar* of [ITU‑T Z.101], and appear here because they are concerned with implicit *Interface-definition* items added to *Data-type-definition*sets.

Each agent type (and agent and state machine) implicitly defines an *Interface-definition*. This *Interface-definition* is in the same context as the definition of the agent type (or agent or state machine), so that (for example) the implicit *Interface-definition* for an item of the   
*Agent-type-definition-****set*** of an *Agent-type-definition* is an item of the *Data-type-definition-****set*** of the *Agent-type-definition*.

Interfaces are implicitly defined by each agent type definition and each agent definition (except the outermost agent) and by the state machine of each agent type definition. The implicitly defined interface for an agent or an agent type has the same name and is defined in the same scope unit as the agent or agent type that defined it. The implicitly defined interface for a state machine has the same name as the containing agent type but is defined in the same scope unit as the state machine that defined it: that is, inside the agent type.

An <interface name> represents the *Sort* of the *Interface-definition*, and this <name> is the *Interface-qualifier* as the <name> of a <path item> for a <qualifier> to identify the *Interface-definition* as a scope unit.

A <signal list> of an <interface use list> denotes items in the *Signal-identifier****-set*** of the   
*Interface-definition* as follows.

a) Each <signal identifier> of a <signal list> of an <interface use list> represents a corresponding *Signal-identifier* in the *Signal-identifier****-set*** that identifies a *Signal-definition*.

b) Each <timer identifier> of a <signal list> of an <interface use list> represents a corresponding *Signal-identifier* in the *Signal-identifier****-set*** that identifies a *Timer-definition*.

c) Each <interface identifier> of a <signal list> of an <interface use list> represents corresponding *Signal-identifier* items in the *Signal-identifier****-set***: one for each *Signal-definition* of *Signal-definition-****set*** of the identified *Interface-definition*.

d) Each *Signal-identifier* in the *Signal-identifier****-set*** appears only once, even if it corresponds to more than one item in the <signal list> of an <interface use list>.

The defining context of entities defined in the interface (<entity in interface>) is the scope unit of the interface, and the entities defined are visible where the interface is visible.

The following paragraphs are from clause 12.1.2 *Semantics* of [ITU‑T Z.101].

The *Sort* of an *Interface-definition* is a pid sort. Each element of the *Sort* is the identity of an agent instance that is able to receive each of the signals identified by the *Signal-identifier-****set*** of the *Interface-definition*.

The *Null-literal-signature* of an *Interface-definition* is the signature for the null literal operator.

The *Data-type-identifier* set of an *Interface-definition* identifies the base interface types (super types) of an interface specialization. Pid is directly or indirectly the base interface types of any *Interface-definition* for a pid sort.

The *Signal-identifier-****set*** of an *Interface-definition* identifies signals defined outside the interface that are used by the interface and also each signal defined by a *Signal-definition* of the *Interface-definition*.

The *Signal-definition-****set*** of an *Interface-definition* is the set of signals defined for the interface. The scope of the signals is such that they are visible wherever the interface is visible. These signals are included in the *Signal-identifier-****set*** of the *Interface-definition*.

The *Signal-identifier-****set*** of an *Interface-definition* is the set of signal identities that apply when the interface appears in the syntax, and the set of signal identities for the pid sort of the *Interface-definition*. An identity of an agent instance is compatible with the pid sort if every   
*Signal-identifier* set of the pid sort is in the valid input signal set of the agent.

The implicit *Interface-definition* for an agent type (or agent or state machine) has a *Sort* with the same *Name* as the agent type (or agent or state machine respectively).

Internally connected gates of an agent type are gates of the agent type that are connected via channels to the gates of either a contained agent or the state machine of the agent type. The internally connected gates of an agent are the gates of the agent that correspond to the internally connected gates of the agent type for the agent.

The implicit *Interface-definition* defined by an agent type contains (in its interface specialization – see [ITU‑T Z.104]) all interfaces given in the incoming signal lists associated with internally connected gates. The *Interface-definition* contains in its *Signal-identifier****-set*** all signals given in the incoming signal lists associated with internally connected gates.

The implicit *Interface-definition* defined by a state machine of an agent type contains (in its interface specialization – see [ITU‑T Z.104]) the interface defined by the agent type itself except any part of that interface concerned only with contained agents. The interface also contains in its interface specialization all interfaces given in the incoming signal lists associated with any gates of the state machine. The interface also contains in its <interface use list> all signals given in the incoming signal lists associated with gates of the state machine.

The *Signal-identifier****-set*** of an implicit *Interface-definition* for an agent type (or agent or state machine) has a *Signal-identifier* for each different signal in the set of signals in all channels or gates for which the destination is the agent type (or agent or state machine respectively), plus the valid input signal set defined explicitly for the agent type (or agent or state machine respectively).

The implicit *Interface-definition* of a typebased agent contains the same *Signal-identifier-****set*** as the *Interface-definition* defined by its type.

NOTE − Because every agent and agent type has an implicitly defined interface with the same name, any explicitly defined interface has to have a different name from every agent and agent type defined in the same scope; otherwise, there are name clashes.

The following paragraph is from clause 12.1.2 *Concrete grammar* of [ITU‑T Z.104].

An <interface definition> (or implicit interface definition) defines a *Data-type-definition* of a choice data type with a unique anonymous *Sort* name in the context that the *Interface-definition* is visible. A <basic sort> that is <as interface> where <interface identifier> identifies the *Interface-definition*, represents this *Data-type-definition*. The *Data-type-definition* is equivalent to defining a choice data type with a <choice of sort> for each distinct *Signal-definition* identified by the *Signal-identifier****-set*** of the *Interface-definition*. The <field name> and <field sort> for each <choice of sort> is determined in the same way as a <choice of sort> for each distinct *Signal-definition* of the *Data-type-definition* defined by a <gate definition>.

From [ITU‑T Z.101] clause 12.1.6.2 *Concrete grammar*:

The <structure definition> for a structure S represents (in the *Operation-signature* set of the *Data‑type‑definition* for S):

a) in the absence of data type specialization (see [ITU‑T Z.104] clause 12.1.9), if no operator named Make is given with an *Operation-result* that is the *Sort-reference-identifier* of the S structure sort (an S structure result), an *Operation-signature* for a generic operator named Make with:

i. a *Formal-argument* list where each item is the *Sort-reference-identifier* of the corresponding (in order) <field name> if the referenced <field> does not contain **optional** and does not contain a <field default initialization>;,

ii. an *Operation-result* that is the S structure result;

iii. the procedure identified by the *Operation-signature* having each formal parameter of its *Parameter-aggregation* derived from the <aggregation kind> of the corresponding <field name>, and a *Result-aggregation* that is **PART**.

From [ITU‑T Z.104] clause 12.1.6.2 *Concrete grammar* and *Semantics*:

b) if S has a <data type specialization> and no *Operation-signature* for an operator named Make is given with an *Operation-result* that is the *Sort-reference-identifier* of the S structure sort (an S structure result), represents (in the *Operation-signature* set of the *Data-type-definition* for S) an *Operation-signature* for a generic operator named Make for each inherited *Operation-signature* for an operator named Make with:

i. a *Formal-argument* list that begins with *Sort-reference-identifier* items of the *Formal‑argument* list of the inherited Make operator and where each subsequent item is the *Sort-reference-identifier* of the corresponding (in order) <field name> if the referenced <field> does not contain **optional** and does not contain a <field default initialization>;

ii. an *Operation-result* that is the S structure result;

iii. the procedure identified by the *Operation-signature* having each formal parameter of its *Parameter-aggregation* derived from the <aggregation kind> of the corresponding <field name>, and a *Result-aggregation* that is **PART**.

The remaining paragraphs of this clause are from [ITU‑T Z.101] clause 12.1.6.2 *Concrete grammar*:

c) for each field, if the <field name> is fn and the <field sort> is fs, an *Operation-signature* for the <operation signature>   
fnExtract ( S ) -> fs;   
for a generic operator where   
fnExtract is a *field-extract-name* formed from the concatenation of the field name and "Extract",   
S is an **in/out** parameter with an empty <aggregation kind>,   
and the result has the same <aggregation kind> as the field fn.

d) for each field, if the <field name> is fn and the <field sort> is fs, an *Operation-signature* for the <operation signature>   
fnModify ( S, fs ) -> S;   
for a generic operator where   
fnModify is a *field-modify-name* formed from the concatenation of the field name and "Modify",   
S is an **in/out** parameter with an empty <aggregation kind>,   
fs is an **in** parameter with the same <aggregation kind> as the field fn,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

e) for each field, if the <field name> is fn, an *Operation-signature* for the <operation signature>  
fnPresent ( S ) -> <<package Predefined>>Boolean;   
for a generic operator where

fnPresent is a *field-present-name* formed from the concatenation of the field name and "Present",   
S is an **in/out** parameter with an empty <aggregation kind>,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

f) an *Operation-signature* for a generic operator named Undefined based on the <operation signature>   
Undefined ( S )-> <<package Predefined>>Boolean;   
which is true if the structure is "undefined": that is, every field of the structure is "undefined",   
S is an **in/out** parameter with an empty <aggregation kind>,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

The generic operations to create structure values are applied as operators.

The generic operations for structure values that have an initial **in/out** parameter (to modify fields, access fields, test for the presence of field and test for the structure value being undefined) are treated as methods.

A structure sort has elements that are all the tuples constructed from data items belonging to the sorts given in the field list. An element of a structure sort has as many component elements as there are fields in the field list. An optional field is a field that does not have to be present. The associated operations determine the semantics of the structures sort.

The result of the generic operator equal is true if and only if for each field of the structure sort:

a) the field is not present in both operands of equal; or

b) the field is present in both operands of equal, and equal for the sort of the field between the values of the field in two structures is true.

The generic operator copy behaves as if the following is interpreted:

a) a new structure is created in which each field has no value (it is "undefined"); then

b) for each field that is present in the operand of copy, the corresponding field of the structure is associated with the data item associated with that field in the operand of copy;

c) for each field that is not present in the operand of copy, and the corresponding field of the structure has a default initialization, the field of the structure is associated with the data item for that field in the default initialization.

Additional generic operations exist for the sort defined as a structure as follows:

a) operations to create structure values;

b) operations to modify structures and to access component data items of structures values; and

c) an operation to test for the presence of optional component data items in structures values, or if the structure is "undefined".

A Make operation with an empty *Formal-argument* list creates a structure value with values associated with fields that have default initialization and all other fields "undefined".

A Make operation with a non-empty *Formal-argument* list creates a new structure and associates each field with the result of the corresponding formal parameter, or if no actual argument is given for the field, the default initialization for that field, or "undefined" if there is no default initialization for the field.

Where there is a Make operation that corresponds to a Make operation in a parent type, the Make operation in the parent type is invoked first with the actual arguments corresponding to the parameters of that Make operation given as actual arguments before associating the fields of the data type with the results of the remaining formal parameters.

If, during interpretation, a field of a structure is "undefined", applying the operation to access this field (with a *field-extract-name*) to the structure causes the predefined exception UndefinedField to be raised. Otherwise, the operation to access a field returns the data item associated with that field. The value associated with a structure is not changed by interpretation of the operation to access a field of the structure.

The operation to modify a field (with a *field-modify-name*) associates the field with the result of its argument *Expression*. The value associated with the structure after interpreting the operation has the field associated with the argument value, but no change to the value associated with any other field.

The operation to test for the presence of a field data item based on the field name (with a *field‑present‑name*) returns the predefined Boolean value false if this field is "undefined", and the predefined Boolean value true otherwise. The value associated with a structure is not changed by interpretation of the operation to test presence of a field of the structure.

The Undefined operation tests if the structure is "undefined", and returns the Boolean value true if each of the fields of the structure is "undefined". The value associated with a structure is not changed by interpretation of the operation to test if the structure is "undefined".

Auxiliary functions

The function localDataTypeDefinitionSet0 gets the local defined <data type definition>s in a scope unit.

localDataTypeDefinitionSet0(su: ScopeUnit0): <data type definition>-set=def

{d∈<data type definition>: d.surroundingScopeUnit0 = su}

The function structMakeSig0 delivers a single-item sequence containing the <operation signature> for the Make operation implied by a structure definition.

structMakeSig0(d :<data type definition>):<operation signature>\* =def

let sort = d.s-<data type definition body>.parentAS0.identifier0 in

let struct = d.s-<data type definition body>.s-<structure definition> in

let opsigs = d.s-<data type definition body>.s-<operations>.

s-<operation signatures>.s-<operator list>.s-<operation signature>-seq in

let makeSig = { os ∈ opsigs.toSet : os.s-<operation name> = "Make" ∧ os.s-<result>.s-<sort> = sort }.take in

let makeArgs = makeSig.s-<arguments> in

< mk-<operation signature>(

mk-<operation preamble>(*undefined*),

mk-<name>("Make"),

if makeArgs = undefined then *empty* else makeArgs endif ⁀

< mk-<argument>( in, struct.fieldSortList0[n]) : n in 1..struct.fieldNameList0.length ∧

expandfieldsSt0(fields)[n] ∈ <mandatory field> ∧

expandfieldsSt0(fields)[n].s-<field default initialization> = undefined

>, //arguments list

mk-<result>(mk-<sort>(mk-<type expression>(d.identifier0, sort)))

) // operation signature

> // list of signatures with one element – signature for Make

endlet // makeArgs

endlet // makeSig

endlet // opsigs

endlet // struct

endlet // sort

The function structMakeDef0 delivers the single-item sequence containing the <operation definition> for the Make operation implied by a structure definition.

structMakeDef0(d :<data type definition>):<operation definition>\* =def

let sort = d.s-<data type definition body>.parentAS0.identifier0 in

let struct = d.s-<data type definition body>.s-<structure definition> in

let opdefs = d.s-<data type definition body>.s-<operations>.

s-<operation definitions>.s-<operation definition>-seq in

let makeDef = { od ∈ opdefs.toSet : od.s-<operation heading>.s-<operation name> = "Make" ∧

od s-<operation heading>.s-<result>.s-<sort> = sort }.take in

let makeParms = makeDef. s-<operation heading>.s-<formal variable parameters>-seq in

< mk-<operation definition> (

mk-<operation heading>( operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>("Make"),

mk-<formal operation parameters>(

if makeParms = undefined then empty else makeParms endif

< mk-<formal variable parameters>( in,

fok.s-<aggregation kind>,

mk-<parameters of sort>( fok.s-<name>, fok.s-<sort>)

) // formal variable parameters

: fok in

< struct.s-<field>-seq[i].s-<mandatory field>.s-<fields of sort>.s-<field of kind>-seq[1]

: i in 1..struct.s-<field>-seq.length ∧

struct.s-<field>-seq[i] ∈ <mandatory field> ∧

struct.s-<field>-seq[i].s-<mandatory field>.s-<field default initialization> = undefined

>

>), // list of f fields, formal operation parameters

mk-<result>(mk-<sort>(mk-<type expression>(d.identifier0,empty)))

> // Make definition

endlet // makeParms

endlet // makeDef

endlet // opdefs

endlet // struct

endlet // sort

The function structOpSigs0 delivers a list of <operation signature>s for methods to modify and access the fields of a Structure data type. Each signature is constructed as an <operation signature> with the identifier of its parent <data type definition> as an additional argument, required when a <operation signature> is transferred from a <method list> to an <operator list> (see section F2.2.9.5).

The function structOpDefs0 delivers the corresponding <operation definition>s.

structOpSigs0(d :<data type definition>):<operation signature>\* =def

let struct = d.s-<data type definition body>.s-<structure definition> in

< mk-<operation signature>(

mk-<name>(s.fieldNameList0[n].s-Token + "Modify"),

< mk-<argument>(inout, d.identifier0) >⁀

< mk-<argument>(in, s.fieldSortList0[n]) >,

mk-<result>(sort) | n in 1..s.fieldNameList0.length > ⁀

< mk-<operation signature>(

mk-<name>(s.fieldNameList0[n].s-Token + "Extract"),

< mk-<argument>(inout, d.identifier0) >,

mk-<result>( s.fieldSortList0[n]),*empty*) | n in 1..s.fieldNameList0.length > ⁀

< mk-<operation signature>(

mk-<name>( s.fieldNameList0[n].s-Token + "Present"),

< mk-<argument>(inout, d.identifier0) >,

mk-<result>(predefinedId0("Boolean"))

| n in 1..s.fieldNameList0.length ∧ fields[n] ∈ <optional field> >⁀

< mk-<operation signature> (

mk-<operation preamble> (undefined, undefined),

mk-<name>("Undefined"),

mk-<arguments>( <

mk-<formal parameter>(inout, mk-<sort>(mk-<type expression>(d.identifier0,empty)))

>), // arguments

mk-<result>(mk-<sort>(mk-<type expression>(predefinedId0("Boolean"),empty)))

> // Undefined signature

endlet

The function structOpDefs0 delivers a list of <operation definition> for methods to modify and access the fields of a Structure data type.

structOpDefs0(d :<data type definition>):<operation definition>\* =def

let struct = d.s-<data type definition body>.s-<structure definition> in

let allFields = struct.s-<field list>.s-<optional field>.s-<fields of sort>⁀

struct.s-<field list>.s-<mandatory field>.s-<fields of sort> in

< mk-<operation definition> (empty, // package use

mk-<operation heading> (operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>(f.s-<field of kind>-seq[1].s-<name>.s-Token + "Modify"),

< mk-<formal variable parameters>(inout, part,

<parameters of sort>(< mk-<name>("structValue") >,

mk-<sort>(mk-<type expression>(d.identifier0,empty))

) // parameters of sort

), // formal variable parameters

mk-<formal variable parameters>(in, part,

<parameters of sort>(< mk-<name>("fieldValue") >, f.s-<sort>)

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result> (part, undefined, mk-<sort>(mk-<type expression>(d.identifier0,empty)))

) // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body>(undefined)) // replaced by actual body in F3

) // statements

) // operation definition

: f in allFields

> // list of Modify definitions for each field

⁀

< mk-<operation definition> (empty, // package use

mk-<operation heading> (operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>(f.s-<field of kind>-seq[1].s-<name>.s-Token + "Extract"),

< mk-<formal variable parameters>(inout, part,

<parameters of sort>(< mk-<name>("structValue") >,

mk-<sort>(mk-<type expression>(d.identifier0,empty))

) // parameters of sort

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result> (part, undefined, f.s-<sort>),

) // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body>(undefined)) // replaced by actual body in F3

) // statements

) // operation definition

: f in allFields

> // list of Extract definitions for each field

⁀

< mk-<operation definition> (empty, // package use

mk-<operation heading> (operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>(f.s-<field of kind>-seq[1].s-<name>.s-Token + "Present",

< mk-<formal variable parameters>(inout, part,

<parameters of sort>(< mk-<name>("structValue") >,

mk-<sort>(mk-<type expression>(d.identifier0,empty))

) // parameters of sort

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result>(part, undefined,

mk-<sort>(mk-<type expression>(predefinedId0("Boolean"),empty)))

) //operation result

) // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body>(undefined)) // replaced by actual body in F3

) // statements

) // operation definition

: f in allFields

> // list of Present definitions for each field

⁀

< mk-<operation definition> (

mk-<operation heading> (operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>("Undefined"),

< mk-<formal variable parameters>(inout, part,

<parameters of sort>(< mk-<name>("structureValue") >,

mk-<sort>(mk-<type expression>(d.identifier0,empty))

) // parameters of sort

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result>(part, undefined,

mk-<sort>(mk-<type expression>(predefinedId0("Boolean"),empty)))

) //operation result

) // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body> (undefined,)) // replaced by actual body in F3

) // statements

) // operation definition

> // Undefined operation definition

endlet // allFields

endlet // struct

The function anonLiterals0 delivers an anonymous <data type definition> with a <literal list> containing all the <field name>s from the <choice of sort>s in a Choice data type as <literal name>s.

anonLiterals0(cdef: <choice definition>, nn: <name>):<data type definition> =def

mk-<data type definition>(empty, undefined,

mk-<data type heading> (mk-<name>(nn.s-Token + "Present"), empty, undefined ), undefined,

mk-<data type definition body>(empty,

mk-<literal list> (undefined, // visibility

< c.s-<name> : c in ( cdef.s-<choice list> ) > // list of literal names

) , // literal list

mk-<operations>(<operation signatures>(<operator list>(empty), <method list>(empty)), empty),

undefined // initialization

) // data type definition body

) // data type definition

endlet

The function choiceOpSigs0 delivers an <operation signature> list for operations to: make an undefined choice value, create a choice value with the specified field with the given value, to modify and access the fields of a choice, test if a field is present, find which field is present in a choice, and check if all fields of a choice are undefined. That is,   
the operator Make()-> C; to make an undefined value,   
plus for each field fn   
the operator fn(fs)->C; create a choice value,   
the operator fnModify(C,fs)->C; modify a choice,   
the operator fnExtract(C)->fs; extract the current choice value and   
the operator fnPresent(C)-> Boolean; test if the field is present   
plus for the Choice data type as a whole   
the operator PresentExtract(C)-> AnonPresent; name of the present field and   
the operator Undefined(C)-> Boolean; check if choice undefined.

choiceOpSigs0(d :<data type definition>, nn :<name>):<operation signature>\* =def

< mk-<operation signature> (

mk-<operation preamble> (virtual, undefined), mk-<name>("Make"),

undefined, // no arguments

mk-<result>(mk-<sort>(mk-<type expression>(d.identifier0,empty)))

> // Make signature

⁀

< mk-<operation signature>(

mk-<operation preamble>(undefined, undefined),

c.s-<field of kind>-seq[1].s-<name>,

mk-<arguments>( < mk-<argument>(in, c.s-<sort>) >),

mk-<result>(mk-<sort>(mk-<type expression>(d.identifier0,empty)))

), // operation signature for fn

: c in d.s-<data type definition body>.s-<choice definition>.s-<choice list>

> // operation signature list – all fn signatures

⁀

< mk-<operation signature> (

mk-<operation preamble>(virtual, undefined),

mk-<name>(c.s-<field of kind>-seq[1].s-<name>.s-Token + "Modify"),

mk-<arguments>( <

mk-<formal parameter>(inout, mk-<sort>(mk-<type expression>(d.identifier0,empty))),

mk-<formal parameter> (in, c.s-<sort>)

>), // arguments

mk-<result>(mk-<sort>(mk-<type expression>(d.identifier0,empty)))

), // operation signature for Modify

: c in d.s-<data type definition body>.s-<choice definition>.s-<choice list>

> // operation signature list – all Modify signatures

⁀

< mk-<operation signature>(

mk-<operation preamble>(virtual, undefined),

mk-<name>(c.s-<field of kind>-seq[1].s-<name>.s-Token + "Extract"),

mk-<arguments>( <

mk-<formal parameter>(inout, mk-<sort>(mk-<type expression>(d.identifier0,empty)))

>), // arguments

mk-<result>(c.s-<sort>)

), // operation signature for Extract

: c in d.s-<data type definition body>.s-<choice definition>.s-<choice list>

> // operation signature list

⁀

< mk-<operation signature>(

mk-<operation preamble>( virtual, undefined),

mk-<name>(c.s-<field of kind>-seq[1].s-<name>.s-Token + "Present"),

mk-<arguments>( <

mk-<formal parameter>(inout, mk-<sort>(mk-<type expression>(d.identifier0,empty)))

>), // arguments

mk-<result>(mk-<sort>(mk-<type expression>(predefinedId0("Boolean"),empty)))

), // operation signature for Present

: c in d.s-<data type definition body>.s-<choice definition>.s-<choice list>

> // operation signature list – all Present signatures

⁀

< mk-<operation signature> (

mk-<operation preamble> (undefined, undefined),

mk-<name>("PresentExtract"),

mk-<arguments>( <

mk-<formal parameter>(inout, mk-<sort>(mk-<type expression>(d.identifier0,empty)))

>), // arguments

mk-<result>(mk-<sort>(

mk-<type expression>(mk-<identifier>(d.fullQualifier0, mk-<name>(nn.s-Token + “Present”)),

empty) // empty actual conext parameter list of type expression

)) // sort , result

> // PresentExtract signature

⁀

< mk-<operation signature> (

mk-<operation preamble> (undefined, undefined),

mk-<name>("Undefined"),

mk-<arguments>( <

mk-<formal parameter>(inout, mk-<sort>(mk-<type expression>(d.identifier0,empty)))

>), // arguments

mk-<result>(mk-<sort>(mk-<type expression>(predefinedId0("Boolean"),empty)))

> // Undefined signature

The function choiceOpDefs0 delivers an <operation definition> list for choice operations as listed in the description of choiceOpSigs0. The body of each operation is here given as a dummy return, which is copied to the implicit procedure for the operation and replaced in the dynamic semantics in F3 by the action needed.

choiceOpDefs0(d :<data type definition>, nn: <name>):<operation definition>\* =def

< mk-<operation definition> (

mk-<operation heading>( operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>("Make"),

undefined, // no formal variable parameters

mk-<result>(mk-<sort>(mk-<type expression>(d.identifier0,empty)))

> // Make definition

⁀

< mk-<operation definition>(empty, // package use

mk-<operation heading>( operator,

mk-<operation preamble>(undefined, undefined), empty, //preamble, qualifier

c.s-<field of kind>-seq[1].s-<name>, // op name fn

< mk-<formal variable parameters>( // formal operation parameters is single item list

in, part, // kind, aggregation

mk-<parameters of sort>(

< mk-<name>("fieldValue") >, // list of one name for field value

c.s-<sort> // sort of field

) // parameters of sort

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result>(part, undefined, mk-<sort>(mk-<type expression>(d.identifier0,empty)))

), // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body> (undefined,)) // replaced by actual body in F3

) // statements

) // operation definition

: c in d.s-<data type definition body>.s-<choice definition>.s-<choice list>

> // list of operation definitions – all fn definitions

⁀

< mk-<operation definition> (empty, // package use

mk-<operation heading> (operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>(c.s-<field of kind>-seq[1].s-<name>.s-Token + "Modify"),

< mk-<formal variable parameters>(inout, part,

<parameters of sort>(< mk-<name>("choiceValue") >,

mk-<sort>(mk-<type expression>(d.identifier0,empty))

) // parameters of sort

), // formal variable parameters

mk-<formal variable parameters>(in, part,

<parameters of sort>(< mk-<name>("fieldValue") >, c.s-<sort>)

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result> (part, undefined, mk-<sort>(mk-<type expression>(d.identifier0,empty)))

) // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body> (undefined,)) // replaced by actual body in F3

) // statements

) // operation definition

: c in d.s-<data type definition body>.s-<choice definition>.s-<choice list>

> // list of Modify definitions for each choice

⁀

< mk-<operation definition> (empty, // package use

mk-<operation heading> (operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>(c.s-<field of kind>-seq[1].s-<name>.s-Token + "Extract"),

< mk-<formal variable parameters>(inout, part,

<parameters of sort>(< mk-<name>("choiceValue") >,

mk-<sort>(mk-<type expression>(d.identifier0,empty))

) // parameters of sort

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result> (part, undefined, c.s-<sort>),

) // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body> (undefined,)) // replaced by actual body in F3

) // statements

) // operation definition

: c in d.s-<data type definition body>.s-<choice definition>.s-<choice list>

> // list of Extract definitions for each choice

⁀

< mk-<operation definition> (empty, // package use

mk-<operation heading> (operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>(c.s-<field of kind>-seq[1].s-<name>.s-Token + "Present",

< mk-<formal variable parameters>(inout, part,

<parameters of sort>(< mk-<name>("choiceValue") >,

mk-<sort>(mk-<type expression>(d.identifier0,empty))

) // parameters of sort

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result>(part, undefined,

mk-<sort>(mk-<type expression>(predefinedId0("Boolean"),empty)))

) //operation result

) // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body> (undefined,)) // replaced by actual body in F3

) // statements

) // operation definition

: c in d.s-<data type definition body>.s-<choice definition>.s-<choice list>

> // list of Present definitions for each choice

⁀

<mk-<operation definition> (

mk-<operation heading> (operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>("PresentExtract"),

< mk-<formal variable parameters>(inout, part,

<parameters of sort>(< mk-<name>("choiceValue") >,

mk-<sort>(mk-<type expression>(d.identifier0,empty))

) // parameters of sort

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result>(part, undefined, mk-<sort>(

mk-<type expression>(mk-<identifier>(d.fullQualifier0, mk-<name>(nn.s-Token + “Present”)),

empty) // empty actual conext parameter list of type expression

)) // sort , operation result

) //operation result

) // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body>(undefined,)) // replaced by actual body in F3

) // statements

) // operation definition

> // PresentExtract operation definition

⁀

< mk-<operation definition> (

mk-<operation heading> (operator,

mk-<operation preamble> (virtual, undefined), empty, //preamble, qualifier

mk-<name>("Undefined"),

< mk-<formal variable parameters>(inout, part,

<parameters of sort>(< mk-<name>("choiceValue") >,

mk-<sort>(mk-<type expression>(d.identifier0,empty))

) // parameters of sort

) // formal variable parameters

>, // formal operation parameters = list of formal variable parameters

mk-<operation result>(part, undefined,

mk-<sort>(mk-<type expression>(predefinedId0("Boolean"),empty)))

) //operation result

) // operation heading

empty, // entity list in operator

mk-<statements> (undefined, // connector name, followed by terminating (dummy return) statement

mk-<return statement>(mk-<return body>(undefined,)) // replaced by actual body in F3

) // statements

) // operation definition

> // Undefined operation definition

Given a <data type definition>, the function getOpSigs0 delivers a list of <operation signature>s. The list is formed by concatenating the <operator list> from the <data type definition> with modified <operation signature>s from the <method list> of the <data type definition>. The modification entails adding the identifier of the parent <data type definition> as an additional argument to each <operation signature> in the <method list> (see section F2.2.9.5).

getOpSigs0(d :<data type definition>):<operation signature>\* =def

let ops = d.s-<data type definition body>.s-<operations>.s-<operation signatures>.s-<operator list> in

let meths = d.s-<data type definition body>.s-<operations>.s-<operation signatures>.s-<method list> in

ops⁀

< mk-<operation signature>(

m.s-<operation preamble>,

m.s-<operation name>,

mk-<arguments>(

< mk-<argument>(inout, mk-<sort>(mk-<type expression>(d.identifier0,empty))) > ⁀

if m.s-<arguments> = *undefined* then empty else m.s-<arguments> endif,

) // arguments

if m.s-<result>= *undefined*

then mk-<sort>(mk-<type expression>(d.identifier0,empty))

else m.s-<result>

endif

) // operation signature

: m in meths >

endlet // meths

endlet // ops

#### F2.2.9.3 Interface type

Abstract syntax

Interface-definition :: Sort

Null-literal-signature

Data-type-identifier-set

Signal-definition-set

Signal-identifier-set

Null-literal-signature = *Literal-signature*

Sort = Name

Concrete syntax

<interface definition> ::

<package use clause>\* [<virtuality>] <interface heading> [<interface specialization>]

<entity in interface>\*

| <signal list definition>

<interface heading> ::

<interface<name> [ <formal context parameters> ] [<virtuality constraint>]

<entity in interface> =

<signal definition list>

| <interface variable definition>

| <interface procedure definition>

| <interface use list>

<interface use list> :: [ <signal list> ]

<interface variable definition> :: <remote variable<name>+ <sort>

<interface procedure definition> :: <remote procedure<name> <procedure signature>

<as interface> :: <interface<identifier>

Further study is needed for the handling of <as interface>. See also Interface definition, which should implicitly define the choice type for <as interface>

Conditions on concrete syntax

∀ifd∈<interface definition>:

isRestrictedByInterface0(ifd.s-<entity in interface>.s-<interface use list>.s-<signal list item>-seq)

Each <signal list item> of the <signal list> in an <interface use list> of an <interface<definition> shall be a <signal<identifier> or an <interface<identifier> or a <remote procedure<identifier> or a <remote variable<identifier>. An <interface<identifier> that is part of the <signal list> shall also respect the restriction. See clause 12.1.2 *Concrete grammar* of [ITU‑T Z.101] and clause 10.4 *Concrete grammar* of [ITU‑T Z.102].

∀interfaceDef∈<interface definition>: ¬ isInterfaceDefContaining0(interfaceDef, interfaceDef)

The <interface definition> shall not contain the <interface<identifier> defined by the <interface definition> either directly or indirectly (via another <interface<identifier>). See clause 12.1.2 *Concrete grammar* of [ITU‑T Z.101].

∀ind∈<interface definition>:∀fcp∈<formal context parameter>:

fcp∈ind.localFormalContextParameterList0 ⇒

fcp∈<signal context parameter list> ∪<remote procedure context parameter> ∪

<remotevariable context parameter list> ∪ <sort context parameter>

The <formal context parameters> shall only contain <signal context parameter list>,   
<remote procedure context parameter>, <remotevariable context parameter list> or <sort context parameter>.

Mapping to abstract syntax

| idef = <interface definition>(\*, \*, <interface heading>(name, \*, \*), spec, entities) then

mk-Interface-definition(Mapping(name),

mk-Literal-signature(mk-Name("null"),

mk-Result(mk-Identifier(Mapping(idef.fullQualifier0), Mapping(name)), REF),

0), // null literal signature

Mapping(spec), // set of identities of inherited interface definitions

{ e ∈ Mapping(entities).toSet: (e ∈ Signal-definition)}, // signal definitions defined in the interface

{ sigid ∈ idef.usedSignalSet0 ∪ {sd.identifier0 | sd ∈ idef.definedSignalSet0} //ids of sigs def for i/f

)

Auxiliary functions

The function localInterfaceDefinitionSet0 gets the local defined interface definition list of a scope unit.

localInterfaceDefinitionSet0(su: ScopeUnit0): <interface definition>-set=def

{d∈<interface definition>:d.surroundingScopeUnit0 = su}

The function isRestrictedByInterface0 decides if each <signal list item> of a list is a <signal<identifier>, <remote procedure<identifier>, <remote variable<identifier> or <interface<identifier>. An <interface<identifier> that is part of the list shall also respect the 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} then isRestrictedByInterface0(sl.tail)

| {**interface**} then

let sl1=getEntityDefinition0(sl.head.s-<identifier>, interface).s-<signal list item>-seq in

isRestrictedByInterface0(sl1)∧isRestrictedByInterface0(sl.tail)

otherwise false

endcase

endif

The function isInterfaceDefContaining0 is used to determine if the first interface definition contains the second interface definition, either directly or indirectly.

isInterfaceDefContaining0(id1: <interface definition>, id2: <interface definition>): Boolean =def

(∃ useList ∈ <interface use list>:

useList.parentAS0 = id1

∧ (∃ident∈<identifier>:

ident.idKind0 = interface

∧ ident = useListItem.s-<signal list item>

∧ useListItem in useList

∧ ( getEntityDefinition0(ident, interface) = id2

∨ ( ∃ id3∈<interface definition>:

isInterfaceDefContaining0 (id1, id3) ∧ isInterfaceDefContaining0 (id3, id2)

) // id3

) // getEntityDefinition0 …

) // ident

) // useList

#### F2.2.9.4 Specialization of data types

Concrete syntax

<data type specialization> :: <data type<type expression> <renaming>

<renaming> :: <rename list>

<rename list> = <rename pair>\*

<rename pair> =

<rename pair operation name> | <rename pair literal name>

<rename pair operation name> :: <operation name> <base type<operation name>

<rename pair literal name> :: <literal name> <base type<literal name>

<interface specialization> :: <interface<type expression>+

Conditions on concrete syntax

∀dataDef∈<data type definition>:∀superTypeDef∈<data type definition>:

isSubtype0(dataDef, superTypeDef) ⇒

superTypeDef.s-<data type definition body>.s-<data type constructor>= undefined ∨

isSameConstructorKind0(

superTypeDef.s-<data type definition body>.s-<data type constructor>,

dataDef.s-<data type definition body>.s-<data type constructor>)

The <data type constructor> must be of the same kind as the <data type constructor> used in the   
<data type definition> of the sort referenced by <data type type expression> in the <data type specialization>.

∀rn∈<renaming>:

let lnl = << rp.s-<literal name>, rp.s2-<literal name>> | rp in rn.s-<rename pair>-seq > in

let onl = << rp.s-<operation name>, rp.s2-<operation name>> | rp in rn.s-<rename pair>-seq > in

(∀i,j∈1..lnl.length: i≠j⇒ lnl[i]≠ lnl[j])∧

(∀i,j∈1..onl.length: i≠j⇒ onl[i]≠ onl[j])

All <literal name>s and all <base type literal name>s in a <rename list> must be distinct. All <operation name>s and all <base type operation name>s in a <rename list> must be distinct.

∀sp∈<data type specialization>:

(let bt = sp.s-<type expression>.BaseType0 in

let onl = <rp.s2-<operation name> |

rp in sp.s-<renaming>.s-<rename pair>-seq: rp∈<rename pair operation name>> in

∀on∈<operation name>: on in onl⇒

(∃os∈<operation signature>: os.surroundingScopeUnit0 = bt ∧ on = os.name0)

endlet)

A <base type operation name> specified in a <rename list> must be an operation with   
<operation name> defined in the data type definition defining the <base type> of <data type type expression>.

∀sp∈<data type specialization>:∀te∈ sp.s-<type expression>:

¬isAbstractType0(te.BaseType0) ⇒∀te′∈ sp.s-<type expression>\{te}:isAbstractType0(te′.BaseType0)

At most one of the <data type<type expression>s is allowed to be not abstract.

Transformations

The model for specialization in clause 8.4 of [ITU-T Z.102] is used, augmented by clause 12.1.9 *Model of* [ITU-T Z.104] for data type specialization.

<data type definition>(use, preamble, heading, spec, undefined) // no data type definition body

=11=>

<data type definition>(use, preamble, heading, spec,

<data type definition body>(undefined, // entities

undefined, // data type constructor

inheritedOperations0(spec), // operations

undefined)) // default initialization

<data type definition>

( use, preamble, heading, spec, dtdb=<data type definition body>

(entities, constr, <operations>

(<operation signatures> (<operator list>(ops), <method list> (meths)), opDefsAndRefs), init) )

provided dtdb ≠ undefined // separate transformation

=11=>

<data type definition>(use, preamble, heading, spec,

<data type definition body>( entities, constr,

<operations>(<operation signatures>(

if dttb.s-<operation signatures>.s-<operator list> ≠ undefined

then <operator list>(ops)

else empty

endif ⁀

inheritedOperations0(spec).s-<operation signatures>.s-<operator list>,

if dttb.s-<operation signatures>.s-<method list> ≠ undefined

then <method list> (meths)

else empty

endif ⁀

inheritedOperations0(spec).s-<operation signatures>.s-<method list>),

opDefsAndRefs ⁀ inheritedOperations0(spec).s-<operation definitions>), init))

The following paragraphs are from clause 12.1.9 *Model of* [ITU-T Z.104].

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, unless the operator or method has been declared as private (see clause 12.1.8.4 of [ITU-T Z.104]) or operation renaming has taken place for that operator or method. Operation renaming has taken place for an operator or method if the inherited operation name appears as the second name in a <rename pair> in which case the operator or method is renamed to the first name in that pair.

When several operators or methods of the <base type> of <sort type expression> have the same name as the <base type operation name> in a <rename pair>, then all of these operators or methods are renamed.

In the following paragraphs ST is a subsort that is a specialization of a data type T, and Tid is an <identifier> for data type T.

For every occurrence of a <basic sort> of the form Tid in a specialization of T, the <basic sort> is replaced by the <basic sort> of the subsort: that is, the <identifier> for the subsort ST. Every occurrence of an <anchored sort> of the form **parent** T in a specialization of T, the <anchored sort> is replaced by an unambiguous qualified <basic sort> for the base type T: that is, the qualified <identifier> for T. Every occurrence of an <anchored sort> of the form **this** T in a specialization of T, the <anchored sort> is replaced by the <basic sort> of the subsort: that is, the <identifier> for the subsort ST.

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 every <argument> in T that is a <basic sort> T or an <anchored sort> of the form **this** T, in the inherited operator or method the <anchored sort> is replaced by the subsort ST.

NOTE – Anchored sorts are handled in the functions inheritedOperations0 and doRename.

Mapping to abstract syntax

| <data type specialization>(base, \*) then Mapping(base)

| <interface specialization>(bases, \*) then

{ Mapping(bases[i]) | i ∈ 1..bases.length }// set of inherited interface identifiers

Auxiliary functions

The function isRenamedBySpec0 determines if a <literal signature> or an <operation signature> is renamed by a <data type specialization>.

isRenamedBySpec0(sn:<literal signature>∪<operation signature>, spec:<data type specialization>):

Boolean=def

(∃rp∈<rename pair>:

(rp. parentAS0.parentAS0 = spec)∧

(rp.s2-<literal name> = sn.name0 ∨ rp.s2-<operation name> = sn.name0))

The function isSameConstructorKind0 is used to determine if two data type constructor items are of the same kind.

isSameConstructorKind0(c1:<data type constructor>, c2:<data type constructor>): Boolean =def

(c1∈<literal list>∧c2∈<literal list>)∨

(c1∈<structure definition>∧c2∈<structure definition>)∨

(c1∈<choice definition>∧c2∈<choice definition>)

Sort compatibility determines when a sort can be used in place of another sort, and when it cannot. The function isSortCompatible0 is used to determine if the first sort is sort compatible to the second one.

isSortCompatible0(sort1:<sort>, sort2:<sort>): Boolean =def

isSameSort0(sort1, sort2) ∨

isDirectlySortCompatible0(sort1, sort2) ∨

(isPidSort0(sort2) ∧

∃sort3∈<sort>: isSortCompatible0(sort1, sort3) ∧ isSortCompatible0(sort3, sort2))

The function isSameSort0 is used to determine if the given two sorts are the same. As well as being the same if they are equal, they are treated as if: they are the same if they have the same derived data type (as long as this is not *undefined*); or they both anchored sorts and *sortId0* is equal.

isSameSort0(sort1: <sort>, sort2: <sort>): Boolean =def

( sort1 = sort2 ) // simple case

∨

( getEntityDefinition0(sort1.sortId0,sort).derivedDataType0 =

getEntityDefinition0(sort2.sortId0, sort).derivedDataType0

∧ getEntityDefinition0(sort1.sortId0,sort).derivedDataType0 ≠ undefined

) // same sort when syntypes changed to sort derived from

∨

( sort1.s-<anchored sort> ≠ undefined ∧ sort2.s-<anchored sort> ≠ undefined ∧

sort1.sortId0 = sort2.sortId0

) // both anchored sorts that expand to the same sort id

∨

( let def1 = getEntityDefinition0(sort1.sortId0, interface) in

def1 ∈ <interface definition> ∧ def1 = getEntityDefinition0(sort2.sortId0, interface)

endlet // def1

)

Further study is needed to refine isSameSort0 for other alternatives of <sort> that are not handled here, to check that <basic sort> includes data type and syntype definitions, and an anchored sort that is the same as a data type.

Determine if two sort lists are the same.

isSameSortList0(sl, sl1: <sort>\*): Boolean =def

(sl.length = sl1.length) ∧

(∀i∈1.. sl.length: isSameSort0(sl[i], sl1[i]))

The function isDirectlySortCompatible0 is used to determine if the sort in the first argument is directly sort compatible to the one in the second.

isDirectlySortCompatible0(sort1:<sort>, sort2:<sort>): Boolean =def

(sort1 ∈ <anchored sort> ∧ sort1.**s**-<basic sort> = sort2) ∨

(sort1 ∈ <pid sort> ∧ isSubSort0(sort1, sort2))

The function isPidSort0 is used to determine if a sort is a pid sort.

isPidSort0(sort: <sort>): Boolean =def

getEntityDefinition0(sort.sortId0,interface) ∈ <interface definition>

∨ sort.sortId0 = predefinedId0("Pid")

The function isSubSort0 is used to determine if the sort given in the first argument is a super sort of the one in the second.

isSubSort0(sort1: <sort>, sort2: <sort>): Boolean =def

let td1 = getEntityDefinition0(sort1.sortId0, sort) in

let td2 = getEntityDefinition0(sort2.sortId0, sort) in

(td1∈<interface definition>⇒isSubtype0(td1, td2)) ∧

(td1∈<data type definition>∪<syntype definition>⇒

isSubtype0(td1.derivedDataType0, td2.derivedDataType0))

endlet

endlet

The function inheritedOperations0 computes the operations inherited from the base type.

inheritedOperations0(spec: <data type 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

mk-<operations>(

mk-<operation signatures>(

if ops ≠ undefined then mk-<operator list>(doRename(ops, spec)) endif, // sigs of operators

if meths ≠ undefined then mk-<method list>(doRename(meths, spec)) endif // sigs methods

),

undefined // Further study needed here to include operation definitions

// copied from base type, with name replaced if renamed,

// parameters/result updated and body updated (inherited sort changed).

)

endlet // meths

endlet // ops

The function doRename takes a set of operation signatures (in a data type definition) and produces a revised list of the operation signatures where the names are updated according to the renaming given in a data type specialization (of a data type definition) and arguments and results updated in the signatures to take account of anchored sorts and to use the subsort to replace the sort in the data type specialization.

doRename( sigsset: <operation signature>-set, spec: <data type specialization>): <operation signature>\*=def

let sig ∈ sigset in

< mk-<operation signature>( sig*.*s-<operation preamble>*,*

if isRenamedBySpec0(sig, spec) then

take({ n ∈ <name>: n = renamepair*.*s1-<name> ∧ renamepair in spec*.*s-<renaming>})

else sig*.*s-<name>

endif,

< reviseArgResSort0(a, spec): a in sig*.*s-<arguments> >,

reviseArgResSort0(sig*.*s-<result>, spec))

*>*

⁀

if | sigsset | = 1 then empty

else doRename(sigsset \ { sig }, spec )

endif

endlet // sig

The function reviseArgResSort0 takes an argument or result and updates the sort if necessary according to the given data type specialization. The function can also handle a sort parameter, but this is only for the function to call itself recursively. When called with an argument (or result or sort), an argument (or result or sort respectively) is returned. For sort that is an anchored sort with **this**, the function returns the subsort to the recursive call (except if the subsort is not a specialization of the basic sort given with the anchored sort in which case the basic sort is returned). For sort that is an anchored sort with **parent**, the function returns to the recursive call: the sort in the data type specialization to the recursive call if no basic sort is given; anchored sort unchanged if the subsort is not a specialization of the basic sort given with the anchored sort; otherwise the basic sort given. For a sort that is not an anchored sort, the function returns to the recursive call the sort any occurrence of the identifier of the sort in the data type specialization changed to the identifier of the subsort.

reviseArgResSort0(arg: <argument> ∪ <result> ∪ <sort>, spec: <data type specialization>):

<argument> ∪ <result> ∪ <sort> =def

let subsortId = spec.parentAS0.identifier0 in

let subsort = mk-<sort>(

mk-<type expression>(getEntityDefinition0(subsortId,sort).derivedDataType0,empty),undefined) in

let supersortId = spec.s-<type expression>.s-<identifier> in

let supersort =

mk-<sort>(mk-<type expression>(getEntityDefinition0(supersortId, sort),empty),undefined) in

case arg of

| <argument>( kind, sort) then

mk-<argument>( kind, reviseArgResSort0(sort, spec))

| <result>(sort) then mk-<result>( reviseArgResSort0(sort, spec))

| <anchored sort> (this) then subsort)

| <anchored sort> (this, basicsort) then

if isSubSort0(basicSort.sortId0, subsortId) then subsort else basicsort endif

| <anchored sort> (parent) then supersort

| <anchored sort> (parent, basicsort) then

if isSubSort0(basicSort.sortId0, subsortId) then arg else basicsort endif // remains anchored if subsort

otherwise

replaceInSyntaxTree0(supersortId*,* subsortId, arg) // change sort to subsort

endcase

endlet // supersort

endlet // supersortId

endlet // subsort

endlet // subsortId

#### F2.2.9.5 Operations

Abstract syntax

Static-operation-signature :: Operation-signature

Dynamic-operation-signature :: Operation-signature

Operation-signature :: Operation-name Formal-argument\*

[Operation-result] *Procedure-identifier*

Formal-argument = Argument

Operation-result = Sort-reference-identifier

Argument = Sort-reference-identifier

Conditions on abstract syntax

Concrete syntax

<operation signatures> :: [ <operator list> ] [ <method list> ]

<operator list> :: <operation signature>+

<method list> :: <operation signature>+

<operation signature> :: <operation preamble> <operation name> [<arguments>] [ <result> ]

<operation preamble> :: [<visibility> [<virtuality>]] | [<virtuality> [<visibility>]]

<arguments> ::<argument> +

<argument> = <formal parameter>

<formal parameter> :: <parameter kind> <sort>

<result> :: <sort>

Transformations

osigs = <operation signatures>((

<operator list>(operations),

<method list> (<operation signature>(pre, name, args, result) ⁀ rest))

=2=>

<operation signatures>(<operator list>(operations ⁀

<operation signature>(pre, name,

<argument>(inout, parentAS0ofKind(osigs, <data type definition>).identifier0) ⁀

if args = *undefined* then empty else args endif,

if result ≠ *undefined* then result

else parentAS0ofKind(osigs, <data type definition>).identifier0

endif ))),

<method list>(rest))

If <operation signature> is contained in a <method list> this is derived syntax and is transformed as follows: An <argument> is constructed from the <parameter kind> in/out, and the <sort identifier> of the sort being defined by the enclosing <data type definition>. If there are no <arguments> (that is the argument list was empty), 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 <result> was omitted, the <result> is the <sort<identifier> of the sort being defined by the enclosing <data type definition>. See clause 12.1.3 *Model* of [ITU‑T Z.104].

<formal parameter>(undefined, sort) =2=> <formal parameter>(in, sort)

A <formal parameter> with no explicit <parameter kind> has the implicit <parameter kind> **in**. See clause 9.4 *Model* of [ITU‑T Z.103].

Mapping to abstract syntax

| <operation signatures>(operators, \*)

then

{ Mapping(operators[i]) | i ∈ 1.. operators.length}

| os = <operation signature>( pre, name, arguments, result) then

let opsig = mk-Operation-signature(

uniqueOperationName0(name,

< arguments[i].s-<formal parameter>.s-<sort> | i ∈ 1..arguments.length >

result.s-<sort>,

mk-<sort>(mk-<type expression>(parentAS0ofKind(os, <data type definition>).identifier0 empty))),

Mapping(arguments),

Mapping(result),

Mapping(os.operatorProcedureId)) in

if pre.s-<virtuality> = virtual ∨ pre.s-<virtuality> = redefined

then

mk-Dynamic-operation-signature(opsig)

else

mk-Static-operation-signature(opsig)

endif

endlet // opsig

The following three text paragraphs are from clause 12.1.3 *Concrete grammar* of [ITU‑T Z.101].

An <operation signature> of an <operator list> represents a *Static-operation-signature*.

In an *Operation-signature*, each *Sort-reference-identifier* in *Formal-argument* is represented by an argument <sort>, and the *Operation-result* is represented by the result <sort>. The <sort> in the <formal parameter> of an <argument> of an operation represents the *Formal-argument*.

The *Operation-name* is unique within the defining scope unit in the abstract syntax even though the corresponding <operation name> is not necessarily unique. The unique *Operation-name* is derived from:

a) the <operation name>; plus

b) the (possible empty) list of argument sort identifiers; plus

c) the result sort identifier; plus

d) the sort identifier of the data type definition in which the <operation name> is defined.

An <operation signature> with <virtuality> **virtual** or **redefined** in the <operation preamble> represents a *Dynamic-operation-signature*. See clause 12.1.3 *Concrete grammar* of [ITU‑T Z.107].

Auxiliary functions

The function operatorProcedureId associates each operation signature with its implicit anonymous procedure identifier.

controlled operatorProcedureId: <operation signature> → <identifier>

initially ∀ o ∈ <operation signature>: o.operatorProcedureId = undefined

The function operationSignatureParameterList0 gets the list of the arguments of an operation signature.

operationSignatureParameterList0(os: <operation signature>):<formal parameter>\* =def

(os.**s**-<formal parameter>-seq)

The function uniqueOperationName0 generates a unique *Operation-name* from the <operation name>, argument sort list, result sort and sort of the data type in which the operation is defined. Any algorithm that generates a unique *Name* is suitable. The one given here turns the parameters into a Token string value that is made into a *Name*. The Token string is a concatenation of the Token string for the name and a list of Token strings representing <identifier> items. Each Token string for an <identifier> starts with separator ("#") that cannot appear in a <name>, therefore ensuring that every *Operation-name* is distinct any *Name* mapped simply from a <name>. The strings are in the following order: string for the result sort, string for the data type, and the strings for each argument sort (in order of the arguments).

NOTE – This algorithm is clearly inefficient and it is expected that analysis tools use a more efficient function to ensure unique *Operation-name* values.

uniqueOperationName0(n:<name>, argsorts:<sort>\*, ressort:<sort>, datatype:<sort>):

*Operation-name* =def

mk-*Name*(

n.**s**-Token ⁀ // character string for <name>

sortOpName0(ressort) ⁀ // string for the result sort

sortOpName0(datatype) ⁀ // string for the data type sort

< sortOpName0(argsorts[i]) | i in 1..argsorts.length > // strings for argument sorts

)

The function sortOpName0 generates the string for a <sort> to use in the functions uniqueOperationName0 and uniqueLiteralName0. The identifier for the sort is determined. The string starts with the separator ("#") followed by a string for the full qualifier of the identifier followed by the string for the name of the identifier. The string for the full qualifier is enclosed by "<<" and ">>" and each path item is the string for the scope unit kind, followed by a space, followed by the string for name of the path item with the character "/" separating path items. That is, the fully qualified identifier string is the same as in the SDL‑ 2010 concrete syntax.

sortOpName0(s:<sort>): Token =def

let id = s.sortId0 in

"#" ⁀

"<<" ⁀

< let pathitem **=** id.s-<path item>-seq[i] in

case pathitem.**s**-<scope unit kind> of

| package then "package"

| system type then "system type"

| system then "system"

| block then "block"

| block type then "block type"

| process then "process"

| process type then "process type"

| state then "state"

| state type then "state type"

| procedure then "procedure"

| signal then "signal"

| type then "type"

| operator then "operator"

| method then "method"

| interface then "interface"

| composition then "composition"

endcase // pathitem.**s**-<scope unit kind>

⁀ " "

⁀ pathitem**.s**-<name>.**s**-Token

⁀ if i **<** pathitem.length then "/" else empty endif

endlet // pathitem

| i in 1..id.s-<path item>-seq[i].length

>

⁀ ">>"

⁀ id.s-<name>.**s**-Token

endlet // id

#### F2.2.9.6 Data type constructors

Concrete syntax

<data type constructor> = <literal list> | <structure definition> | <choice definition>

##### F2.2.9.6.1 Literals

Abstract syntax

Literal-signature :: Literal-name

Result

Literal-natural

*Literal-natural* = *Nat*

Conditions on abstract syntax

∀ls1, ls2∈ Literal-signature: ls1.parentAS1=ls2.parentAS1∧ls1≠ls2⇒

ls1.s-Literal-natural ≠ ls2.Literal-natural

Each *Literal-signature* in the *Literal-signature* set of a *Value-data-type-definition* shall have a different *Literal-natural*. See clause 12.1.6.1 *Abstract grammar* of [ITU‑T Z.101].

Concrete syntax

<literal list> :: [<visibility>] <literal signature>+

<literal signature> = <literal name> | <named number>

<named number> :: <literal name> <Natural<simple expression>

Mapping to abstract syntax

| litlist = <literal list>(\*,literals)

then

let signatures = addNamedNumbers0(literals, 1) in

let nopreamble = <operation preamble>(undefined, undefined) in

let sort = litlist.parentAS0.parentAS0.identifier0 in

let arg = mk-<argument>( in, mk-<anchored sort>(sort)) in

let resbool = mk-<result>( predefinedId0("Boolean")) in

let reslit = mk-<result>( mk-<anchored sort>(sort)) in

{ Mapping(signatures[i]) | i ∈ 1..signatures.length }

∪ { Mapping(mk-<operation signature>(nopreamble, mk-<name>("num"), < arg, arg >,

mk-<result>(predefinedId0("Natural")))))

Mapping(mk-<operation signature>(nopreamble, mk-<name>("<"), < arg, arg >, resbool)),

Mapping(mk-<operation signature>(nopreamble, mk-<name>(">"), < arg, arg >, resbool)),

Mapping(mk-<operation signature>(nopreamble, mk-<name>(">="), < arg, arg >, resbool)),

Mapping(mk-<operation signature>(nopreamble, mk-<name>("<="), < arg, arg >, resbool)),

Mapping(mk-<operation signature>(nopreamble, mk-<name>("first"), < >, reslit)),

Mapping(mk-<operation signature>(nopreamble, mk-<name>("last"), < >, reslit)),

Mapping(mk-<operation signature>(nopreamble, mk-<name>("pred"), < arg >, reslit)),

Mapping(mk-<operation signature>(nopreamble, mk-<name>("succ"), < arg >, reslit)),

Mapping(mk-<operation signature>(nopreamble, mk-<name>("num"), < arg, arg >,

mk-<result>(predefinedId0("Natural")))))

} // operation signatures for generic operators

endlet // reslit

endlet // resbool

endlet // arg

endlet // sort

endlet // nopreamble

endlet // signatures

The following text is from clause 12.1.6.1 *Concrete grammar* of [ITU‑T Z.101].

Each <literal name> in a <literal list> is given the lowest possible Natural simple expression value for the *Literal-natural* of the *Literal-signature* not occurring for any other <literal signature>s of the same <literal list>, considering the <literal name>s one by one from left to right. The result is, for example,

**literals** B, A = 2, C, D;

has B < C , C < A , A < D , num(C) = 1, num(D) = 3

The following text is from clause 12.1.6.1 *Semantics* of [ITU‑T Z.101].

Additional generic operators exist for a sort defined by a constructor that creates a *Literal-signature* set, as follows:

a) an operator that gives the position of each data item in the ordering as the corresponding Natural simple expression value;

b) operators that compare two data items with respect to the established ordering; and

c) operators that return the first, last, next or previous data item in the ordering.

For a sort named S that is defined by a constructor that creates a *Literal-signature* set, there is a *Static-operation-signature* list equivalent to the following:

num ( S ) -> Natural;

"<" ( S, S ) -> Boolean;

">" ( S, S ) -> Boolean;

"<=" ( S, S ) -> Boolean;

">=" ( S, S ) -> Boolean;

first -> S;

last -> S;

succ ( S ) -> S;

pred ( S ) -> S;

where Boolean is the predefined Boolean sort and Natural is the predefined Natural sort, and the parameters and the results correspond to an *Aggregation-kind* of **PART**.

The operator num returns the Natural simple expression value corresponding to the *Literal-natural* of the literal.

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 values corresponding to each *Literal-natural* of the two literals. The operator first returns the first data item in the ordering (the literal with the lowest Natural simple expression value corresponding to the *Literal-natural*). The operator last returns the last data item in the ordering (the literal with the highest Natural simple expression value corresponding to the *Literal-natural*). The operator pred returns the preceding data item (that is, the literal with the highest *Literal-natural* that is less than the *Literal-natural* corresponding to the actual parameter), if one exists, or the same as the operator last, otherwise. The operator succ returns the successor data item (that is, the literal with the lowest *Literal-natural* that is greater than the *Literal-natural* corresponding to the actual parameter) in the ordering, if one exists, or the same as the operator first, otherwise.

| nn = <named number>(name, number) then

mk-Literal-signature

( uniqueLiteralName0 (name),

mk-Result(Mapping(

identifier0(parentAS0ofKind(nn, <data type definition>))))

mk-Literal-natural(Mapping(number)))

The following text is from clause 12.1.6.1 *Concrete grammar* of [ITU‑T Z.101].

The *Literal-name* is unique within the defining scope unit in the abstract syntax even if the corresponding <literal name> is not unique. The unique *Literal-name* is derived from:

a) the <literal name>; plus

b) the sort identifier of the data type definition in which the <literal name> is defined.

In a *Literal-signature*, the *Result* is the sort introduced by the <data type definition> defining the <literal signature>.

The Natural simple expression value of the <Natural simple expression> occurring in a <named number> represents the *Literal-natural* of the *Literal-signature.*

Auxiliary functions

The function uniqueLiteralName0 generates a unique *Literal-name* from the <literal name>, and sort of the data type in which the operation is defined. Any algorithm that generates a unique *Name* is suitable. The one given here turns the parameters into a Token string value that is made into a *Name*. The Token string is a concatenation of the Token string for the name and a Token string representing the sort <identifier>. The Token string for an <identifier> starts with separator ("#") that cannot appear in a <name>, therefore ensuring that every *Literal-name* is distinct from any *Name* mapped simply from a <name>. The result is distinct from an *Operation-name* without parameters of the same sort, because an *Operation-name* includes a result sort.

uniqueLiteralName0(n:<name>, datatype:<sort>):

*Literal-name* =def

mk-*Name*(

n.**s**-Token ⁀ // character string for <name>

sortOpName0(datatype) ⁀ // string for the data type sort

)

The function visibility0 provides the visibility of an operation or literal.

visibility0(s:<operation signature>∪<literal signature>):<visibility>=def

if s∈<operation signature> then s.s-<visibility>

else s.parentAS0.s-<visibility>

The function addNamedNumbers0 changes the <literal signature> list (of a <literal list>) so that each <literal name> in turn is changed into a <named number>.

addNamedNumbers0(literals: <literal signature>\*, i: Nat): <literal signature>\* =def

if i > literals.length then literals

else

addNamedNumbers0(

if literals[i] ∈ <literal name> then changeLitsig0(literals, i ) else literals endif,

i+1)

endif

The function changeLitsig0 changes the item i of a <literal signature> list from a <literal name> into a <named number>.

changeLitsig0(literals: <literal signature>\*, i: Nat): <literal signature>\* =def

headLitsigs0(literals, i) ⁀

< mk-<named number>(

literals[i], mk-<operand5>( mk-<identifier>( predefinedQual0("Integer"),

mk-<name>( nextNumber0(literals, 0))))

) > ⁀

tailLitsigs0(literals, i)

The function headLitsigs0 produces a list of the items before item i of a <literal signature> list.

headLitsigs0(literals: <literal signature>\*, i: Nat): <literal signature>\* =def

if i = 1 then empty

else literals.head ⁀ headLitsigs0(literals.tail, i-1)

The function tailLitsigs0 produces a list of the items after item i of a <literal signature> list.

tailLitsigs0(literals: <literal signature>\*, i: Nat): <literal signature>\* =def

if i = literals.length then empty

else literals[i+1] ⁀ tailLitsigs0(literals, i+1)

The function nextNumber0 produces a Token corresponding to the lowest available Natural number not occurring as a <simple expression> of any <named number> in the given <literal signature> list, and equal or higher than the initial value given. The function natToIntToken is in clause F2.2.1.6.5, Function definitions on AS0. The simpleMapping converts each <simple expression> into a Constant-expression that is evaluated by value1 to produce an integer Token compared with the current value of the counter i until one is found not already in the <literal signature> list.

nextNumber0(literals: <literal signature>\*, i: Nat): Token =def

if (¬∃ n ∈ <named number> : n in literals

∧ ( n.s-<simple expression>.simpleMapping.value1.intTokenToNat= i))

then

i.natToIntToken

else

nextNumber0(literals, i+1)

endif

##### F2.2.9.6.2 Structure data types

Concrete syntax

<structure definition> :: [<visibility>] <field list>

<field list> = <field>+

<field> =

<optional field> | <mandatory field>

<optional field> :: <fields of sort>

<mandatory field> :: <fields of sort> [<field default initialization>]

<field default initialization> :: <constant expression>

<fields of sort> :: [<visibility>] <field of kind>+ <field sort>

<field of kind> :: <aggregation kind> <field<name>

<field sort> = <sort>

Conditions on concrete syntax

∀sd∈<structure definition>: sd.fieldNameList0.length = |sd.fieldNameList0.toSet|

Each <field<name> of a structure sort shall be different from every other <field<name> of the same <structure definition>.

Transformations

< <optional field>(<fields of sort>(vis, < f > ⁀ rest, sort))> provided rest ≠ empty =5=>

< <optional field>(<fields of sort>(vis, < f >, sort)),

<optional field>(<fields of sort>(vis, rest, sort)) >

< <mandatory field>(<fields of sort>(vis, < f > ⁀ rest, sort), init)> provided rest ≠ empty =5=>

< <mandatory field>(<fields of sort>(vis, < f >, sort), init),

<mandatory field>(<fields of sort>(vis, rest, sort), init) >

A <field list> containing a <field> that has a <fields of sort> with an <field of kind> list is derived concrete syntax where this <field> is replaced by a list of <field>s separated by <end>, one for each <field of kind> in the order of occurrence of each <field of kind>. Each <field> in the replacement list has a <fields of sort> with same <visibility> and <field sort> as the original <fields of sort>. Each <field> in the replacement list is **optional** if the original <field> was **optional**, or has the <field default initialization> of the original <field> if there was one. See clause 12.1.6.2 *Model* of [ITU‑T Z.104].

<operator application>(ident, params = first ⁀ < param > ⁀ last)

provided ident.s-<operation name>.s-Token = "Make" ∧

param = undefined ∧

∃ sd(\*, fields) ∈ <structure definition>: sd.parentAS0. parentAS0 =

parentAS0ofKind(ident.refersto0, <data type definition>) ∧

expandfieldsSt0(fields)[first.length + 1] ∈ <mandatory field> ∧

expandfieldsSt0(fields)[first.length + 1].s-<field default initialization> ≠ undefined

=8=>

<method application>(<operator application>( ident, params),

<identifier>(parentAS0ofKind(ident.refersto0, <data type definition>).fullQualifier0,

<name>(sd.fieldNameList0[first.length + 1].s-Token + "Modify")),

< expandfieldsSt0(fields)[first.length + 1].s-<field default initialization>>)

A Make operation with a non-empty *Formal-argument* list creates a new structure and associates each field with the result of the corresponding formal parameter, or if no actual argument is given for the field, the default initialization for that field, or "undefined" if there is no default initialization for the field. See clause 12.1.6.2 *Semantics* of [ITU‑T Z.101].

Mapping to abstract syntax

The following paragraphs are from clause 12.1.6.2 *Concrete grammar* of [ITU‑T Z.101] and describe the mapping to abstract syntax for <structure definition> as part of the mapping is a <data type definition body> in clause F2.2.9.2 Data type definition.

The <structure definition> for a structure S represents (in the *Operation-signature* set of the   
*Data-type-definition* for S):

a) in the absence of data type specialization, if no operator named Make is given with an *Operation-result* that is the *Sort-reference-identifier* of the S structure sort (an S structure result), an *Operation-signature* for a generic operator named Make with:

i. a *Formal-argument* list where each item is the *Sort-reference-identifier* of the corresponding (in order) <field name> if the referenced <field> does not contain optional and does not contain a <field default initialization>;

ii. an *Operation-result* that is the S structure result;

iii. the procedure identified by the *Operation-signature* having each formal parameter of its *Parameter-aggregation* derived from the <aggregation kind> of the corresponding <field name>, and a *Result-aggregation* that is PART.

b) if S has a <data type specialization> and no constructor (an operator named Make with an *Operation-result* that is the *Sort-reference-identifier* of the S structure sort) is given, an *Operation-signature* for a generic operator named Make for each (normally one) inherited *Operation-signature* for an operator named Make with:

i. a *Formal-argument* list that begins with *Sort-reference-identifier* items of the   
*Formal-argument* list of the inherited Make operator and where each subsequent item is the *Sort-reference-identifier* of the corresponding (in order) <field name> if the referenced <field> does not contain optional and does not contain a <field default initialization>;

ii. an *Operation-result* that is the S structure result;

iii. the procedure identified by the *Operation-signature* having each formal parameter of its *Parameter-aggregation* derived from the <aggregation kind> of the corresponding <field name>, and a *Result-aggregation* that is PART.

c) for each field, if the <field name> is fn and the <field sort> is fs, an *Operation-signature* for the <operation signature>   
fnExtract ( S ) -> fs;   
for a generic operator where   
fnExtract is a *field-extract-name* formed from the concatenation of the field name and "Extract",   
S is an in/out parameter with an empty <aggregation kind>,   
and the result has the same <aggregation kind> as the field fn.

d) for each field, if the <field name> is fn and the <field sort> is fs, an *Operation-signature* for the <operation signature>   
fnModify ( S, fs ) -> S;   
for a generic operator where   
fnModify is a *field-modify-name* formed from the concatenation of the field name and "Modify",   
S is an in/out parameter with an empty <aggregation kind>,   
fs is an inparameter with the same <aggregation kind> as the field fn,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is PART.

e) for each field, if the <field name> is fn, an *Operation-signature* for the <operation signature>  
fnPresent ( S ) -> <<package Predefined>>Boolean;   
for a generic operator where

fnPresent is a *field-present-name* formed from the concatenation of the field name and "Present",   
S is an in/out parameter with an empty <aggregation kind>,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is PART.

f) an *Operation-signature* for a generic operator named Undefined based on the <operation signature>   
Undefined ( S )-> <<package Predefined>>Boolean;   
which is true if the structure is "undefined": that is, every field of the structure is "undefined",   
S is an in/out parameter with an empty <aggregation kind>,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is PART.

Auxiliary functions

The function implicitSignaturesAdded records whether the implicit signatures for structure operators have been added into a data type.

controlled implicitSignaturesAdded: <identifier> → Boolean

initially ∀ id ∈ <identifier>: (getEntityDefinition0(id, <data type definition>) ≠ undefined ∧

( getEntityDefinition0(id, <data type definition>).s-<data type definition body>.s-<data type constructor>

∈ <structure definition> ))

⇒

id.implicitSignaturesAdded = false

The function expandfieldsSt0 expands a list of structure fields to produce a new list where each field contains one name.

expandfieldsSt0(fl: {<optional field> ∪ <mandatory field>}\*):

{<optional field> ∪ <mandatory field>}\*=def

if fl.length = 0 then undefined

elseif fl.length = 1 then

if fl.head ∈ <optional field> then

< mk-<optional field>(

mk-<fields of sort>(

fl.head.s-<fields of sort>.s-<visibility>,

< fl.head.s-<fields of sort>.s-<field of kind>-seq.head >,

fl.head.s-<fields of sort>.s-<field sort>

) ) >

else

< mk-<mandatory field>(

mk-<fields of sort>(

fl.head.s-<fields of sort>.s-<visibility>,

< fl.head.s-<fields of sort>.s-<field of kind>-seq.head >,

fl.head.s-<fields of sort>.s-<field sort>),

fl.head.s-<field default initialization>

) >

endif ⁀

if fl.head.s-<fields of sort>.s-<field of kind>-seq.length = 1

then empty

else

expandfieldsSt0(

if fl.head ∈ <optional field> then

< mk-<optional field>(

mk-<fields of sort>(

fl.head.s-<fields of sort>.s-<visibility>,

fl.head.s-<fields of sort>.s-<field of kind>-seq.tail,

fl.head.s-<fields of sort>.s-<field sort>

) ) >

else

< mk-<mandatory field>(

mk-<fields of sort>(

fl.head.s-<fields of sort>.s-<visibility>,

fl.head.s-<fields of sort>.s-<field of kind>-seq.tail,

fl.head.s-<fields of sort>.s-<field sort>),

fl.head.s-<field default initialization>

) >

endif)

else

expandfieldsSt0(< fl.head >) ⁀ expandfieldsSt0(fl.tail)

endif

The function fieldNameList0 gives the list of structure field or choice names for a structure definition or choice definition in the order the fields appear in the definition.

fieldNameList0(d(\*,lst):<structure definition>∪<choice definition>): <name>\*=def

if d∈ <structure definition> then fieldNameListSt0(lst) else fieldNameListCh0(lst) endif

The function fieldNameListSt0 gives the list of structure field names from a list of fields in the order the fields appear in the list.

fieldNameListSt0(fl: {<optional field> ∪ <mandatory field>}\*): <name>\*=def

if fl.length = 0 then undefined

elseif fl.length = 1 then

< fl.head.s-<fields of sort>.s-<field of kind>-seq.head.s-<name> > ⁀

if fl.head.s-<fields of sort>.s-<field of kind>-seq.length = 1

then empty

else fieldNameListSt0(fl.head.s -<fields of sort>.s-<field of kind>-seq.tail)

endif

else

fieldNameListSt0(< fl.head >)⁀fieldNameListSt0(fl.tail)

endif

The function fieldNameListCh0 gives the list of choice field names from a list of fields in the order the fields appear in the list.

fieldNameListCh0(cl: <choice of sort>\*): <name>\*=def

if cl.length = 0 then undefined

elseif cl.length = 1 then

< cl.head.s-<choice of sort>.s-<field of kind>-seq.head.s-<name> > ⁀

if cl.head.s-<choice of sort>.s-<field of kind>-seq.length = 1

then empty

else fieldNameListCh0(cl.head.s -<choice of sort>.s-<field of kind>-seq.tail)

endif

else

fieldNameListCh0(< cl.head >)⁀fieldNameListCh0(cl.tail)

endif

The function fieldSortList0 gives the list of structure field sorts from a list of fields in the order the fields appear in the list.

fieldSortList0(d(\*,lst): <structure definition>∪<choice definition>):<sort>\*=def

if d∈ <structure definition> then fieldSortListSt0(lst) else fieldSortListCh0(lst) endif

The function fieldSortListSt0 gives the list of structure field sorts from a list of fields in the order the fields appear in the list.

fieldSortListSt0(fl: {<optional field> ∪ <mandatory field>}\*): <name>\*=def

if fl.length = 0 then undefined

elseif fl.length = 1 then

< fl.head.s-<fields of sort>.s-<field sort> > ⁀

if fl.head.s-<fields of sort>.s-<field of kind>-seq.length = 1

then empty

else fieldSortListSt0(fl.head.s -<fields of sort>.s-<field of kind>-seq.tail)

endif

else

fieldSortListSt0(< fl.head >)⁀ fieldSortListSt0(fl.tail)

endif

The function fieldSortListCh0 gives the list of choice sorts from a list of fields in the order the fields appear in the list.

fieldSortListCh0(cl: <choice of sort>\*): <name>\*=def

if cl.length = 0 then undefined

elseif cl.length = 1 then

< cl.head.s-<choice of sort>.s-<field sort> > ⁀

if cl.head.s-<choice of sort>.s-<field of kind>-seq.length = 1

then empty

else fieldSortListCh0(cl.head.s -<choice of sort>.s-<field of kind>-seq.tail)

endif

else

fieldSortListCh0(< cl.head >)⁀fieldSortListCh0(cl.tail)

endif

defaultValue(n: <name>): <constant expression> =def

case n.parentAS0 in

| <mandatory field>(\*, <field default initialization>(e)) then e

otherwise

undefined

endif

##### F2.2.9.6.3 Choice data types

Concrete syntax

<choice definition> :: [<visibility>] [ <choice list> ]

<choice list> = <choice of sort>+

<choice of sort> :: [<visibility>] <field of kind>+ <field sort>

Conditions on concrete syntax

∀cd∈<choice definition>: cd.fieldNameList0.length = |cd.fieldNameList0.toSet|

Each <field<name> of a choice sort shall be different from every other <field<name> of the same <choice definition>. See clause 12.1.6.3 *Concrete grammar* of [ITU‑T Z.101].

Transformations

< <choice of sort>(vis, < c > ⁀ rest, sort) >

provided rest ≠ empty

=5=>

< <choice of sort>(vis, < c >, sort), <choice of sort>(vis, rest, sort) >

A <choice list> containing a <choice of sort> with an <aggregation kind> <field name> pair list is derived concrete syntax where this <choice of sort> is replaced by a list of <choice of sort> items separated by <end>, one for each <aggregation kind> <field name> pair in the order of occurrence of each <aggregation kind> <field name> pair. Each <choice of sort> in the replacement list has the same <visibility> and <field sort> as the original <choice of sort>. See clause 12.1.6.3 *Model* of [ITU‑T Z.104].

Mapping to abstract syntax

The following paragraphs are from clause 12.1.6.3 *Concrete syntax* of [ITU‑T Z.101] and describe the mapping to abstract syntax for <choice definition> as part of the mapping is a <data type definition body> in clause F2.2.9.2 Data type definition.

The <choice definition> for a choice sort C represents (in the *Operation-signature* set of the   
*Data-type-definition* for C):

a) an *Operation-signature* for a generic operator named Make with an empty *Formal-argument* list and an *Operation-result* that is the *Sort-reference-identifier* of the C choice sort, and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is PART.

b) for each field, if the <field name> is fn and the <field sort> is fs, an *Operation-signature* for the operation signature   
fn (fs ) -> C;   
for a generic field association operator where   
fn is a *field-associate-name* which is the same as the field name,   
fs is an in parameter with the same <aggregation kind> as the field fn,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is PART.

c) for each field, if the <field name> is fn and the <field sort> is fs, an *Operation-signature* for the <operation signature>   
fnExtract ( C ) -> fs;   
for a generic operator where   
fnExtract is a *field-extract-name* formed from the concatenation of the field name and "Extract",   
C is an in/out parameter with an empty <aggregation kind>,  
and the result has the same <aggregation kind> as the field fn.

d) for each field, if the <field name> is fn and the <field sort> is fs, an *Operation-signature* for the <operation signature>   
fnModify (C, fs ) -> C;   
for a generic operator where   
fnModify is a *field-modify-name* formed from the concatenation of the field name and "Modify",   
C is an in/out parameter with an empty <aggregation kind>,   
fs is an **in** parameter with an empty <aggregation kind>,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is PART.

e) for each field, if the <field name> is fn, an Operation-signature for the <operation signature>   
fnPresent ( C ) -> <<package Predefined>>Boolean;   
for a generic operator where   
fnPresent is a *field-present-name* formed from the concatenation of the field name and "Present",   
C is an **in/out** parameter with an empty <aggregation kind>,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is PART.

f) an *Operation-signature* for a generic operator named PresentExtract based on the <operation signature>   
PresentExtract (C )-> AnonPresent;.   
where AnonPresent is defined as a literal constructor data type that uses the field names of the choice as literals as described below,   
C is an in/out parameter with an empty <aggregation kind>,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is PART.

g) an *Operation-signature* for a generic operator named Undefined based on the <operation signature>   
Undefined (C )-> <<package Predefined>>Boolean;   
which is true if the choice is "undefined",   
where C is an **in/out** parameter with an empty <aggregation kind>,   
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is PART.

The <choice definition> for a choice sort C also represents an additional (anonymous)   
*Data-type-definition*, that for description above is called AnonPresent. This *Data-type-definition* is placed in the context that the *Data-type-definition* for C, therefore both AnonPresent and its contained *Literal-signature-****set*** are visible where C is visible. This is defined with a *Literal-signature-****set*** where each <field name> of the choice sort C represents a *Literal-signature*. The order of the literals is the same as the order in which the <field name>s are specified left to right in the choice sort C. The purpose of this data type is to allow the operation PresentExtract with a result that corresponds to the field name. The name of this data type being unknown prevents it being used for other purposes.

#### F2.2.9.7 Behaviour of operations

Concrete syntax

<operation definition item> =

<operation definition> | <operation reference> | <external operation definition>

<operation reference> :: <operation kind> <operation signature>

<external operation definition> :: <operation kind> <operation signature>

<operation definition> ::

<package use clause>\* <operation heading> <entity in operation>\*

{ <operation body> | <statements> }

<operation heading> ::

<operation kind> <operation preamble> [ <qualifier> ] <operation name>

<formal operation parameters> [<operation result>]

<formal operation parameters> = <formal variable parameters>\*

<operation kind> :: **operator** | **method**

<operation identifier> :: [ <qualifier> ] <operation name>

<entity in operation> =

<data definition>

| <variable definition>

| <select definition>

<operation body> ::

<start> {<free action>}\*

<operation result> :: <result aggregation> [<variable<name>] <sort>

<operation definition> is an alternative textual concrete syntax to <operation diagram> and therefore defines a scope unit. See clause 6 of [ITU‑T Z.106].

Conditions on concrete syntax

∀opRef∈<operation reference>:

(opRef.s-<operation signature>.s-<arguments> = undefined ∨

opRef.s-<operation signature>.s-< result> = undefined)⇒

(∀opRef1∈<operation reference>:

opRef1 ≠ opRef ∧ opRef.parentAS0 = opRef1.parentAS0 ⇒ opRef1.name0≠ opRef.name0)

<arguments> and <result> of the <operation signature> in an <operation reference> are allowed to be omitted if there is no other <operation reference> within the same sort of data that has the same name. In this case, the referenced <operation definition> is identified simply by its name. The <operation reference> enables the referenced <operation definition> to be located, so that it is possible to map the concrete definition to the enclosing data type definition in the logical hierarchy in the abstract grammar. Modified from clause 8.2 *Concrete grammar* of [ITU‑T Z.101].

∀opDef∈<external operation definition>:

(opDef.s-<operation signature>.s-<arguments> = undefined ∧

opDef.s-<operation signature>.s-<result> = undefined)⇒

¬ (∃ opDef1 ∈ <external operation definition> \ { opDef }:

opDef.parentAS0 = opDef1.parentAS0 ∧

opDef.s-<operation signature>.s-<name> = opDef1.s-<operation signature>.s-<name>)

∧ (∃ opSig ∈ <operation signature> :

opDef.parentAS0 = opSig.parentAS0 ∧

opDef.s-<operation signature>.s-<name> = opSig.name0)

It is allowed to omit <arguments> and <result> in <external operation definition> if there is no other <external operation definition> within the same sort which has the same name, and an <operation signature> is present. In this case, the <arguments> and the <result> are derived from the <operation signature>. See clause 12.1.7 *Concrete grammar* of [ITU‑T Z.104].

∀od∈<operation definition>: ∃!os∈<operation signature>:

od.name0 = os.name0 ∧ od.parentAS0 = os.parentAS0 ⇒ isSameOperationAndSignature0(od, os)

In an <operation signature> of <operation signatures> there shall be one and only one corresponding definition (<operation reference> or <external operation definition>) in the <operation definitions> of the <operations>. See clause 12.1.1 *Concrete grammar* of [ITU‑T Z.104].

For each <operation definition> if there exists an <operation signature> in the same scope unit having the same <operation name>, then it must have the same <argument sort>s and <parameter kind>s as specified in the <formal operation parameters> (if present) and the same <result sort> as specified in <operation result> (if present).

∀os∈<operation signature>: ∃!od∈(<operation reference>∪<external operation definition>):

od.parentAS0= os.parentAS0∧ od.name0= os.name0∧ isSameOperationAndSignature0(od, os)

For an <operation signature> of <operation signatures> there shall be one and only one corresponding definition (<operation reference> or <external operation definition>) in the <operation definitions> of the <operations>. See clause 12.1.1 *Concrete grammar* of [ITU‑T Z.104].

∀bs∈<operation body>∪<statement>: parentAS0ofKind(bs, <operation definition>)≠undefined ⇒

(¬∃ie∈<imperative expression>: isAncestorAS0(bs, ie))∧

(∀id∈<identifier>: id.idKind0∉{synonym, procedure}∧ isAncestorAS0(bs, id)⇒

isDefinedIn0(getEntityDefinition0(id, id.idKind0),

parentAS0ofKind(bs, <operation definition>)))

<operation body> as well as the <statement>s in <operation definition> may contain neither an <imperative expression> nor an <identifier> defined outside the enclosing <operation definition>, except for <synonym identifier>s, <operation identifier>s, <literal identifier>s and <sort>s.

Transformations

<operations>(opsigs, opdefs)

provided

∃ opdef in opdefs: (

opdef ∈ <operation definition> ∧

if opdefs[i].s-<operation heading>.s-<operation kind> = **operator**

then

if opsigs.s-<operator list> ≠ undefined

**then** ¬∃ opsig in opsigs.s-<operator list>:

isSameOperationSignature0(operationSignature0(opdef), opsig)

**else** true

**endif** // operator

else // method

if opsigs.s-<method list> ≠ undefined

**then** ¬∃ opsig in opsigs.s-<method list>:

isSameOperationSignature0(operationSignature0(opdef), opsig)

**else** true

**endif** // method

**endif** true // operator or method

) // exists opdef in opdefs such that no corresponding signature in opsigs

=8=>

mk-<operations>(

mk-<operation signatures>(

mk-<operator list>(

if opsigs.s-<operator list> ≠ undefined then opsigs.s-<operator list> else empty endif ⁀

< mk-<operation signature>(operationSignature0(opdef[i])) :

i ∈ 1..opdefs.length ∧

if opsigs.s-<operator list> ≠ undefined

**then** ¬∃ opsig in opsigs.s-<operator list>:

isSameOperationSignature0(operationSignature0(opdef[i]), opsig)

**else** true

**endif**

**> //** list of new operator signatures

**), //** operator list

mk-<method list>(

if opsigs.s-<method list> ≠ undefined then opsigs.s-<operator list> else empty endif ⁀

< mk-<operation signature>(operationSignature0(opdef[i])) :

i ∈ 1..opdefs.length ∧

if opsigs.s-<method list> ≠ undefined

**then** ¬∃ opsig in opsigs.s-<operator list>:

isSameOperationSignature0(operationSignature0(opdef[i]), opsig)

**else** true

**endif**

**> //** list of new method signatures

**), //** method list

), // operation signatures

opdefs

)

For every <operation definition> which does not have a corresponding <operation signature>, an <operation signature> is constructed. See clause 12.1.7 *Model* of [ITU‑T Z.104].

eod = <external operation definition>(kind, <operation signature>(pre, name, undefined, undefined))

=8=>

let opSig = take({os ∈ <operation signature> : eod.parentAS0 = opSig.parentAS0 ∧ os.name0 = name}) in

mk-<external operation definition>( kind,

mk-<operation signature>(pre, name, opSig.s-<arguments>, opSig.s-<result>))

endlet

It is allowed to omit <arguments> and <result> in <external operation definition>. In this case, the <arguments> and the <result> are derived from the <operation signature>. See clause 12.1.7 *Concrete grammar* of [ITU‑T Z.104].

od.operatorProcedureId

provided

od ∈ <operation definition>

∧ od.operatorProcedureId = undefined

=??=>

(operatorProcedureId \{( od, undefined )} ) ∪ {( od, newName)}

od = <operation definition>(use, <operation heading>(kind, \*, \*, name, \*, params,

<operation result>(var, sort)), entities, body)

provided od.operatorProcedureId ≠ undefined

=8=>

od // the operation definition

and

od.getEntities0 // the procedure is placed in the data type definition

=> od.getEntities0 ⁀

< mk-<internal procedure definition>(use,

mk-<procedure heading>(undefined, undefined,

od.operatorProcedureId.s-<name>, *empty*, undefined, undefined,

(if kind = method then

< mk-<formal variable parameters>(inout,

mk-<parameters of sort>(od.operatorProcedureId,

parentAS0ofKind(od, <data type definition>).identifier0)) >

// has the same anonymous name as the procedure

else *empty* endif) ⁀

< mk-<formal variable parameters>(p.s-<parameter kind>, p.s-<parameters of sort>) |

p in params >,

mk-<procedure result>(var, sort)),

entities, makeProcedureBody(body))

>

Further study is needed for the above transform because in Z.101 to Z.107, a procedure definition is not allowed as <entity in data type>; therefore this needs to be changed to a *Mapping* according see text below from Z.101. Therefore this transform is not listed at the end of the document.

An <operation definition> represents a *Procedure-definition* in the *Procedure-definition-****set*** of the directly enclosing *Data-type-definition*. The *Procedure-name* of the *Procedure-definition* is an anonymous unique name, and the *Procedure-definition* is associated with the *Operation-signature* by the *Procedure-identifier* in the *Operation-signature*. The <formal operation parameters> list for the operator represents the *Procedure-formal-parameter* list of the *Procedure-definition* in the same way as the formal parameters for a procedure. The <operation result> represents the *Result* of the *Procedure-definition*, therefore the <result aggregation> represents the *Result-aggregation*. The components of the <operation body area> are used in the same way as the components of a <procedure body area> to represent the *Data-type-definition-****set*,** *Syntype-definition-****set***, *Variable-definition-****set***, *Procedure-definition-****set*** and *Procedure-graph* of the *Procedure-definition*. See clause 12.1.7 *Concrete grammar* of [ITU‑T Z.101].

If the <operation kind> is **operator**, each item in the <formal operation parameters> represents an item (in the same order) in the *Procedure-formal-parameter* list of the *Procedure-definition* for the operator. If the <operation kind> is **method**, the first item in the *Procedure-formal-parameter* list of the *Procedure-definition* for the method is an *Inout-parameter* with an anonymous *Variable-name* and a *Sort-reference-identifier* that identifies the sort for the data type in which the method is defined. Each item in the <formal operation parameters> of a method represents a subsequent item (in the same order) in the *Procedure-formal-parameter* list of the *Procedure-definition* for the method. See clause 12.1.7 *Concrete grammar* of [ITU‑T Z.104].

If the <operation heading> begins with the keyword **operator**, then <operation definition> defines the behaviour of an operator. If the <operation heading> begins with the keyword **method**, then <operation definition> defines the behaviour of a method. Whether an operation is an operator or method is part of the operation signature and therefore part of the identity of the operation. See clause 12.1.7 *Concrete grammar* of [ITU‑T Z.104].

If an operation contains informal text, the interpretation of expressions involving application of the corresponding operation is not formally defined by the Specification and Description Language but is determined from the informal text by the interpreter. If informal text is specified, a complete formal specification has not been given in the Specification and Description Language. See clause 12.1.7 *Model* of [ITU‑T Z.104].

Auxiliary functions

The Boolean function isSameOperationAndSignature0 is true if the given operation is an <operation definition> has the same result as the given signature, both have the same number of parameters and each the parameters of the operation has the same sort and parameter kind as the given signature.

isSameOperationAndSignature0(od: <operation definition>∪<external operation definition>,

os: <operation signature> ): Boolean =def

if od ∈ <operation definition> then

let seq1 = od.operationFormalparameterList0 in

letseq2 = os.operationSignatureParameterList0 in

(od.s-<operation heading>.s-<operation result> ≠ undefined ⇒

isSameResult0(od.s-<operation heading>.s-<operation result>, os.s-<result>) ∧

(seq1≠empty ⇒

seq1.length = seq2.length∧

(∀i∈1..seq1.length: isSameSort0(seq1[i].parentAS0.s-<sort>, seq2[i].s-<sort>) ∧

seq1[i].parentAS0.s-<parameter kind>= seq2[i].s-<parameter kind>)))

endlet // seq2

endlet // seq1

else // od ∈ <external operation definition>

od.s-<operation signature> = os

endif

The function operationFormalparameterList0 gets the list of names of the 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 variable parameters>-seq >

The function idKind0 determines the entity kind of an <identifier> according to its syntactic position. For an <identifier> in an <actual context parameter>, the result is the entity kind of the formal context parameter which matches the list position of the actual parameter in the formal context parameter list of the base type <identifier> of the enclosing <type expression>. For a base type <identifier> in an <type expression>, the result is determined by context of the <type expression> and in most cases is sort. For an <identifier> in an <encoding path>, the result is determined by number of items in the <encoding expression>: if there is one item it is a channel or gate, and if there are two items the <identifier> gives the literal in the Encoding data type.

idKind0(i: <identifier>): EntityKind0 =def

case i.parentAS0 of

| <active agents expression> then agent

| a=<actual context parameter> then

take({f ∈ <formal context parameter>:

isContextParameterCorresponded0(f, a)

∧ parentAS0ofKind(f, EntityDefinition0) = a.parentAS0.baseType0

}).entityKind0

| <agent constraint atleast> then agent type

| <agent constraint exactly> then agent type

| <agent type context parameter> then agent type

| <any expression> then sort

| <as channel> then channel

| <as gate> then gate

| <as signal> then signal

| <assignment> then variable

| <block type reference> then block type

| <channel endpoint> then   
 if getEntityDefinition0(i, agent) ≠ undefined then agent else state endif

| <channel to channel connection> then channel

| <composite state type reference> then state type

| <compositestate type context parameter> then state type

| <create body> then   
 if getEntityDefinition0(i, agent) ≠ undefined then agent   
 else agent type   
 endif

| <destination> then agent

| <encoding expression> then signal

| <encoding path> (i) then

if getEntityDefinition0(i, channel) ≠ undefined then channel else gate endif

| <encoding path> (\*,\*) then literal

| <exported> then remote procedure

| <export body> then variable

| <exported variable> then remote variable

| <external synonym definition item> then sort

| <formal parameter> then sort

| <import expression> then remote variable

| <indexed variable> then variable

| <interface constraint> then interface

| <interface variable definition> then sort

| <internal synonym definition item> then sort

| <local variables of sort> then sort

| <loop variable definition> then sort

| <loop variable indication identifier> then variable

| <operation result> then sort

| <output body item> then signal

| <package reference> then package

| <package use clause> then package

| <parameters of sort> then sort

| <procedure call body> then procedure

| <procedure context parameter> then procedure

| <procedure result> then sort

| <process type reference> then process type

| <range check constrained sort> then sort

| <range check expression> then sort

| <remote procedure call body> then remote procedure

| <reset clause> then timer

| <result> then sort

| <set clause> then timer

| <signal constraint> then signal

| <signal definition> then sort

| <signal list item> then stimulusKind(i)

| <sort> then sort

| <sort constraint> then sort

| <state partition connection entry> then state

| <state partition connection exit> then state

| <stimulus> then variable

| <synonym> then synonym

| <syntype definition syntype> then sort

| <syntype> then syntype

| <system type reference> then system type

| <textual endpoint constraint> then

if i.parentAS0.parentAS0 ∈ <textual interface gate definition> then interface

elseif getEntityDefinition0(i, block type) ≠ undefined then block type

elseif getEntityDefinition0(i, process type) ≠ undefined then process type

else state type

endif

| <textual interface gate definition> then interface

| <timer active expression> then timer

| <timer communication constraint> then

if i.parentAS0.s1-<identifier>= i then timer else variable endif

| te = <type expression> then

case te.parentAS0 of

| <interface specialization> then interface

| <procedure call body> then procedure

| sp = <specialization> then

case sp.parentAS0 of

| aah = <agent additional heading> then

case aah.parentAS0 of

| ai = <agent instantiation> then

if ai.parentAS0 ∈ <block heading> then block type else process type endif

| atah = <agent type additional heading> then

case atah.parentAS0 of

| <block type heading> then block type

| <process type heading> then process type

otherwise system type

endcase

| <system heading> then system type

endcase

| <composite state heading> then state type

| <composite state type heading> then state type

| <procedure heading> then procedure

| <signal definition> then signal

| <state aggregation type heading> then state type

endcase

| <typebased block heading> then block type

| <typebased composite state> then state type

| <typebased process heading> then process type

| <typebased state partition heading> then state type

| <typebased system heading> then system type

otherwise sort

endcase

| <valid input signal set> then stimulusKind(i)

| <variables of sort> then sort

| <virtuality constraint> then parentAS0ofKind(i, TypeDefinition0 ).kind0

| <via path> then

if getEntityDefinition0(i, channel) ≠ undefined then channel

else gate

endif

endcase

The function stimulusKind determines the entity kind of an <identifier> that is known to be a stimulus.

stimulusKind(i: <identifier>): EntityKind0 =def

if getEntityDefinition0(i, signal) ≠ undefined then signal   
 elseif getEntityDefinition0(i, timer) ≠ undefined then timer   
 elseif getEntityDefinition0(i, remote procedure) ≠ undefined then remote procedure   
 elseif getEntityDefinition0(i, remote variable) ≠ undefined then remote variable   
 elseif getEntityDefinition0(i, signallist) ≠ undefined then interface   
 else interface   
 endif

The function makeProcedureBody makes <operation body> or statement list into the <procedure body> or <compound statement> for the procedure called for the operation.

makeProcedureBody(b:<operation body> ∪ <statements>): <procedure body> ∪ <statements> =def

case b of

| <operation body>(onexc, start, actions) then mk-<procedure body>(onexc, start, actions)

| <statements> then b

endcase

The function operationSignature0 produces the operation signature for a given operation definition.

operationSignature0(opdef: <operation definition>): < operation signature > =def

let hdg = opdef.s-<operation heading> in

let pars = hdg.s-<formal operation parameters> in

mk-<operation signature>(

hdg.s-<operation preamble>,

hdg.s-<operation name>,

mk-<arguments>(

< mk-<formal parameter>(pars[i].s-<parameter kind>, pars[i].s-<sort>): i ∈ 1..pars.length > ),

opdef.s-<operation heading>.s-<operation result>.s-<sort>

) // operation signature

endlet // pars

endlet // hdg

#### F2.2.9.8 Additional data definition constructs

##### F2.2.9.8.1 Name class mapping

Name class mapping is defined in clause A.2.10 of [ITU-T Z.104] and is intended for use only to define SDL‑2010 language features and is not part of the SDL‑2010 language itself. It is no longer formally described in Annex F.

##### F2.2.9.8.2 Restricted visibility

Concrete syntax

<visibility> = public | protected | private

Conditions on concrete syntax

∀d∈<data type definition>: d.specialization0≠ undefined ⇒

(∀cons∈<data type constructor>:

cons.surroundingScopeUnit0 = d ⇒ cons.s-<visibility> = undefined)

<visibility> must not precede a <literal list>, <structure definition>, or <choice definition> in a <data type definition> containing <data type specialization>.

∀os∈<operation signature>:

os.virtuality0 ∈{redefined, finalized}⇒os.visibility0=undefined

<visibility> must not be used in an <operation signature> that redefines an inherited operation signature.

Transformations

<operation preamble>(virt, vis) provided vis ≠ undefined

=2=><operation preamble>(virt, undefined)

<literal list>(vis, sigs) provided vis ≠ undefined

=2=> <literal list>( undefined, sigs)

<structure definition>(vis, fields) provided vis ≠ undefined

=2=> <structure definition>( undefined, fields)

<fields of sort>(vis, fields, sort) provided vis ≠ undefined =2=> <fields of sort>( undefined, fields, sort)

<choice definition>(vis, fields) provided vis ≠ undefined =2=> <choice definition>( undefined, fields)

<choice of sort>(vis, fields, sort) provided vis ≠ undefined

=2=> <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. See clause 12.1.9 of [ITU‑T Z.104].

Auxiliary functions

The function isPrivate0 determines if a <literal signature> or an <operation signature> is private.

isPrivate0 (s: <literal signature> ∪ <operation signature>): Boolean =def

( s.visibility0 = private)

The function isPublic0 determines if a <literal signature> or an <operation signature> is public.

isPublic0 (s: <literal signature> ∪ <operation signature>): Boolean =def

( s.visibility0 ≠ private) ∧ ( s.visibility0 ≠ protected)

##### F2.2.9.8.3 Syntypes

Abstract syntax

Syntype-definition :: Syntype-name

Parent-sort-identifier

Range-condition

[ *Default-initialization* ]

Parent-sort-identifier = Sort-identifier

Concrete syntax

<syntype> = <syntype<identifier>

<syntype definition> ::

<package use clause>\*

{ <syntype definition syntype> | <syntype definition data type> }

<syntype definition syntype> ::

<syntype<name> <parent sort identifier>

[<default initialization>] [<constraint>]

<syntype definition data type> ::

<type preamble> <data type heading> [<data type specialization>]

<data type definition body> <constraint>

<parent sort identifier> = <sort>

Transformations

< <syntype definition>(uses,

<syntype definition data type>(preamble,

<data type heading>(kind, name, params, vconstr), spec, body, constr)) >

=8=>

let nn = newName in

< <syntype definition>(uses,

<syntype definition syntype>(name, <identifier>(*empty*, nn), undefined, constr)),

<data type definition>(uses, preamble, <data type heading>(kind, nn, params, vconstr),

spec, body) >

endlet // newName

A <syntype definition data type> is 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 without the <constraint>, followed by a <syntype definition> with <syntype definition syntype> based on this anonymously named sort and including <constraint>. See clause 12.1.8.1 *Model* of [ITU‑T Z.104].

Mapping to abstract syntax

| <syntype definition>(\*, <syntype definition syntype>(name, parent, \*, constr)) then

mk-Syntype-definition(Mapping(name), Mapping(parent), Mapping(constr))

##### F2.2.9.8.4 Constraint

Abstract syntax

Range-condition :: Condition-item-set

Condition-item = Open-range | Closed-range | Size-constraint

Open-range :: Operation-identifier Constant-expression

Closed-range :: Constant-expression Constant-expression

Size-constraint :: Operation-identifier

{ Open-range | Closed-range }\*

Concrete syntax

<constraint> = <range condition> | <size constraint>

<range condition> = <range>+

<range> = <closed range> | <open range>

<open range> = <constant> | <open range with operator>

<open range with operator> ::

{ <equals sign> | <not equals sign> | <less than sign> | <greater than sign> |

<less than or equals sign> | <greater than or equals sign> } <constant>

<constant> = <constant expression>

<closed range> :: <constant> <constant>

<size constraint> :: <range condition>

Conditions on concrete syntax

The text items below for conditions are from clause 12.1.8.2 of [ITU-T Z.101].

∀sd∈<syntype definition>: sd.s-implicit.s-<constraint>∈<range condition> ⇒

(let rc = sd.s-implicit.s-<constraint> in

(∀sym∈<less than sign>: isAncestorAS0(rc, sym)⇒ isDefinedSym0(sd, "<"))∧

(∀sym∈<greater than sign>: isAncestorAS0(rc, sym)⇒ isDefinedSym0(sd, ">"))∧

(∀sym∈<less than or equals sign>: isAncestorAS0(rc, sym)⇒ isDefinedSym0(sd, "<="))∧

(∀sym∈<greater than or equals sign>: isAncestorAS0(rc, sym)⇒ isDefinedSym0(sd, ">="))

endlet)

The symbol "<" shall only be used in the concretesyntax of the <range condition> if that symbol has been defined with an <operation signature>: "<" ( P, P ) -> <<**package** Predefined>>Boolean; where P is the sort of the syntype, and similarly for the symbols ("<=", ">", ">=", respectively).

∀sd∈<syntype definition>: sd.s-implicit.s-<constraint>∈<range condition>⇒

∀cr∈<closed range>:

isAncestorAS0(sd.s-implicit.s-<range condition>,cr)⇒ isDefinedSym0(sd, "<=")

A <closed range> shall only be used if the symbol "<=" is defined with an <operation signature>: "<=" ( P, P ) -> <<**package** Predefined>>Boolean; where P is the sort of the syntype.

∀sd∈<syntype definition>:∀rc∈<range condition>:∀ce∈<constant expression>:

isAncestorAS0(rc, ce)∧rc.surroundingScopeUnit0=sd⇒isSameSort0(ce.staticSort0, sd.identifier0)

A <constant expression> in a <range condition> shall have the same sort as the sort of the syntype.

∀sd∈<syntype definition>: ∀sc∈<size constraint>:

sc=sd.s-implicit.s-<constraint>⇒isDefinedSym0(sd, "length")

A <size constraint> shall only be used in the concrete syntax of the <range condition> if the symbol length has been defined with an <operation signature>: length (**in** P ) -> <<**package** Predefined>> Natural; where P is the sort of the syntype.

Mapping to abstract syntax

| r = <range condition> then

mk-Range-condition(

< if item ∈ Expression0 then mk-Open-range(rangeOperator(r, "="), Mapping(item))

else Mapping(item) endif : item in r >.toSet)

| r = <open range with operator>(<equals sign>(), const) then

mk-Open-range(rangeOperator(r, "="), Mapping(const))

| r = <open range with operator>(<not equals sign>(), const) then

mk-Open-range(rangeOperator(r, "/="), Mapping(const))

| r = <open range with operator>(<less than sign>(), const) then

mk-Open-range(rangeOperator(r, "<"), Mapping(const))

| r = <open range gen with operator>(<greater than sign>(), const) then

mk-Open-range(rangeOperator(r, ">"), Mapping(const))

| r = <open range with operator>(<less than or equals sign>(), const) then

mk-Open-range(rangeOperator(r, "<="), Mapping(const))

| r = <open range with operator>(<greater than or equals sign>(), const) then

mk-Open-range(rangeOperator(r, ">="), Mapping(const))

| <closed range>(c1, c2) then

mk-Closed-range(mk-Open-range(rangeOperator(r, ">="), Mapping(c1),

mk-Open-range(rangeOperator(r, "<="), Mapping(c2)) }

| sc=<size constraint>(rc) then

mk-Size-constraint(rangeOperator(sc,"length"), Mapping(rc))

Auxiliary functions

The function isPredefSort0 is used to determine if a sort is one of the predefined sorts.

isPredefSort0(s: <sort>): Boolean =def

getEntityDefinition0(s, sort)=undefined∧s.s-<name>∈PredefinedSort0

The function isDefinedSym0 is used to determine if the given symbol is defined and the each parameter's sort is the same as that of the specified syntype.

isDefinedSym0(sd:<syntype definition>, sym: Symbol0): Boolean =def

(let dtd=sd.derivedDataType0 in

if sym∈{"<",">","<=",">="} then

(∃ll∈<literal list>: ll.surroundingScopeUnit0 = dtd)∨

(∃os∈<operation signature>: (os.surroundingScopeUnit0=dtd)∧

(let fpl= os.operationSignatureParameterList0 in

os.entityName0= sym ∧

isPredefSort0(os.s-<result>.s-<sort>) ∧ os.s-<result>.s-<sort>.s-<name>= "Boolean" ∧

fpl.length = 2 ∧

getEntityDefinition0(fpl[1].s-<sort>, sort) =sd∧

getEntityDefinition0(fpl[2].s-<sort>, sort) =sd

endlet))

else // sym∈{"length"}

(∃os∈<operation signature>: os.surroundingScopeUnit0=dtd∧

(let fpl=os.operationSignatureParameterList0 in

os.name0= "length"∧

os.s-<result>.s-<sort>.sortId0 = predefinedId0("Natural")∧

fpl.length= 1∧

getEntityDefinition0(fpl[1].s-<sort>, sort).derivedDataType0=dtd

endlet))

endlet)

rangeOperator(rc: <range condition>, t: Token): Identifier

mk-Identifier(

Mapping( // qualifier within the sort defined by the construct enclosing rc

getEntityDefinition0(

case p =parentAS0ofKind(rc,

<inline syntype definition>

∪ <range check constrained sort>

∪ <size constraint>

∪ <sort>

∪ <syntype definition data type>

∪ < syntype definition syntype >

∪ <textual answer part>)

of

| <inline syntype definition> then p.s-<basic sort>.sortId0

| <range check constrained sort> then p.s-<identifier>

| <size constraint> then

case gp = parentAS0ofKind(p,

<range check constrined sort>

∪ <syntype definition data type>

∪ < syntype definition syntype >)

of

| <range check constrained sort> then gp.s-<identifier>

| <syntype definition data type> then gp.identifier0

| < syntype definition syntype > then gp.identifier0

endcase // sort id from size constraint

| <sort> then p.sortId0

| <syntype definition data type> then p.identifier0

| <syntype definition syntype > then p.identifier0

| <textual answer part> then

let decision = p.parentAS0.parentAS0 in

let question = take({quest ∈ Expression0 : quest.parentAS0 = decision }) in

let questionsorts = staticSortSet0(question) in // possible question sorts

let answerconsts = { const ∈ Expression0 : const.parentAS0.parentAS0.parentAS0 = decision ∧ const.parentAS0 ∈ <textual answer part> } in // possible answer sorts

let answersorts = **U** { staticSortSet0(const) : const ∈ answerconsts } in

take(questionsorts ∩ answersorts) // one of the sorts in both sets – there should be exactly one

endlet // answersorts

endlet // answerconsts

endlet // questionsorts

endlet // question

endlet // decision

endcase, // sort id

sort).derivedDataType0.fullQualifierWithin0 // sort def - derived type – qualifier within

), // Mapping – for qualifier

mk-Name(t))

##### F2.2.9.8.5 Synonym definition

Concrete syntax

<synonym definition> :: <synonym definition item>+

<synonym definition item> =

<internal synonym definition item> | <external synonym definition item>

<internal synonym definition item> ::

<synonym<name> [<sort>] <constant expression>

<external synonym definition item> ::

<synonym<name> <predefined<sort>

Conditions on concrete syntax

The conditions below are from clause 12.1.8.3 *Concrete grammar* of [ITU‑T Z.104].

∀syno∈<internal synonym definition item>:

¬ isContainedInConsExp0(syno, syno.s-<constant expression>)

The <constant expression> shall not refer to the synonym defined by the <synonym definition> either directly or indirectly (via another synonym).

∀sdi∈<internal synonym definition item>:sdi.s-<sort> ≠ undefined⇒

∃s∈<sort>: s∈sdi.s-<constant expression>.staticSortSet0∧isSameSort0(s, sdi.s-<sort>)

If a <sort> is specified, the sort of the <constant expression> has to be compatible with this sort.

∀ sdi∈<internal synonym definition item>:

|sdi.s-<constant expression>.staticSortSet0|>1⇒ sdi.s-<sort> ≠ undefined

If the sort of the <constant expression> is unique (that is, the expression belongs to only one sort), it is allowed to omit the <sort> in the <synonym definition>. and the Sort-reference-identifier is derived from the constant expression sort.

Transformations

<synonym definition> (< sd > ⁀ rest) provided rest ≠ empty =5=>

mk-<synonym definition> (< sd >) ⁀ mk-<synonym definition> (rest)

A <synonym definition> that defines multiple synonyms is a shorthand for a sequence of <synonym definition> items, each defining one synonym. See clause 12.1.8.3 *Model* of [ITU‑T Z.104].

<external synonym definition item> ( name, sort) =7=>

mk-<internal synonym definition item>( name, sort, take({s ∈ sort}))

An <external synonym definition item> defines a <synonym> whose result is not defined in a specification. See clause 12.1.8.3 *Concrete grammar* of [ITU‑T Z.104].

Mapping to abstract syntax

| <synonym definition>(sdi)

then

mk-Variable-definition(Mapping(sdi.s-<name>),

if sdi ∈ <internal synonym definition item> ∧ |sdi.s-<constant expression>.staticSortSet0|=1

then Mapping(sdi.s-<constant expression>.staticSortSet0)

else Mapping(sdi.s-<sort>)

endif, // sort

PART, // aggregation

if sdi ∈ <internal synonym definition item>

then Mapping(sdi.s-<constant expression>)

else undefined

endif // constant expression

)

The paragraphs below are from clause 12.1.8.3 *Concrete grammar* of [ITU‑T Z.104].

A <synonym definition> represents a Variable-definition in the context in which the synonym definition appears with the special property that the variable is read-only. The <synonym<name> represents the Variable-name.

If a <sort> is specified, this determines the Sort-reference-identifier of the Variable-definition.

If the <sort> in the <synonym definition> is omitted the *Sort-reference-identifier* of the Variable-definition is derived from the constant expression sort.

The Variable-definition has a PART has Aggregation-kind.

The <constant expression> in the concrete syntax denotes a *Constant-expression* in the abstract syntax.

Auxiliary functions

The function isContainedInConsExp0 is used to determine if a <constant expression> refers to the synonym defined by the enclosing <synonym definition> either directly or indirectly.

isContainedInConsExp0(def: <internal synonym definition item>, exp: <constant expression>):

Boolean =def

∃synoId∈<synonym>: isAncestorAS0(exp, synId) ∧

(def = getEntityDefinition0(synoId, synonym)∨

isContainedInConsExp0

(def, getEntityDefinition0(synoId, synonym).**s**-<constant expression>))

#### F2.2.9.9 Use of data

##### F2.2.9.9.1 Expressions and expressions as actual parameters

Abstract syntax

Expression = Constant-expression

| Active-expression

Constant-expression = Literal

| Conditional-expression

| Equality-expression

| Operation-application

| Range-check-expression

| Type-check-expression

| Type-coercion

| Agent-instance-pid-value

Active-expression = Variable-access

| Conditional-expression

| Operation-application

| Equality-expression

| Imperative-expression

| Range-check-expression

| Type-check-expression

| Type-coercion

| Value-returning-call-node

| Encoding-expression

| Decoding-expression

| Agent-instance-pid-value

Agent-instance-pid-value :: Agent-instance\*

Agent-instance = Agent-name Instance-number

Instance-number = Expression

Imperative-expression = Now-expression

| Pid-expression

| Timer-active-expression

| Timer-remaining-duration

| Active-agents-expression

| Any-expression

| State-expression

| Signal-expression

| Signallist-expression

Actual-parameters :: { *Expression* | UNDEFINED }\*

Please note that the above definition could be simplified. This can be done by omitting the difference between active expressions and constant expressions. This difference does not show up at any place, so it could be simply dropped.

Conditions on abstract syntax

Further study needed to formulate the condition.

The leftmost item in the Agent-instance list of an Agent-instance-pid-value shall have an Agent-name that is the Agent-name of an Agent-definition directly contained by the Agent-definition for the system. The Agent-definition for the system has an Agent-type-identifier for an Agent-type-definition with the Agent-kind **SYSTEM**.

Except the leftmost item, each other item in the list Agent-instance list of an Agent-instance-pid-value shall have an Agent-name that is the Agent-name of an Agent-definition directly contained by the Agent-instance item immediately to the left.

The Instance-number of an Agent-instance in the Agent-instance-pid-value of a Constant-expression shall be a positive Natural Constant-expression less than or equal to the Initial-number of the Number-of-instances of the Agent-definition for the Agent-name in the same Agent-instance.

See clause 12.2.1 *Abstract grammar* of [ITU‑T Z.104].

Concrete syntax

<expression> =

<expression0>

| <range check expression>

| <type coercion>

<expression0> =

<binary expression>

| <operand5>

| <equality expression>

| <create expression>

| <value returning procedure call>

| <decoding expression>

| <encoding expression>

<simple expression> = <constant expression>

<actual parameters> = <actual parameter list>

<actual parameter list> = [ <actual parameter> ]+

<actual parameter> = <expression>

<constant expression> = <constant<expression0>

<binary expression> ::

<expression> <infix operation name> <expression>

<operand5> :: [ <hyphen> | **not** ] <primary>

<primary> =

<operator application>

| <literal>

| <expression>

| <conditional expression>

| <extended primary>

| <active primary>

| <synonym>

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

<expression list> = <expression>+

<imperative expression> =

<now expression>

| <pid expression>

| <timer active expression>

| <timer remaining duration>

| <active agents expression>

| <any expression>

| <state expression>

| <signal expression>

| <signallist expression>

<agent instance pid value> :: [ <system<name> ] [ <agent instance> ]

<agent instance> :: <agent instance>\* <agent<name> [ <Natural<expression> ]

Conditions on concrete syntax

∀consExp∈<expression0>: consExp.parentAS0 ∈

<alternative question>∪

<closed range>∪

<default initialization>∪

<exported variables of sort>∪

<field default initialization> ∪

<internal synonym definition item>∪

<named number>∪

<number of instances>∪

<open range>∪

<open range with operator>∪

<timer default initialization>∪

<transition option>∪

<variables of sort>

⇒

¬∃activePri∈<active primary>: isAncestorAS0(consExp, activePri) ∧

¬∃vrpc∈<value returning procedure call>: isAncestorAS0(consExp, vrpc)

An <expression0> that does not contain any <active primary>, or a <value returning procedure call> is a <constant expression>. In the contexts where a <constant expression> is required, there shall not exist an <active primary> or <value returning procedure call> within an <expression0>.

∀id∈<identifier>: ∀expr∈<expression>:

( expr.parentAS0 ∈

<alternative question> ∪

<named number> ∪

<number of instances> ∪

<transition option> )

∧ isAncestorAS0(expr, id)∧id.idKind0∈{literal, operator, method}⇒

getEntityDefinition0(id, id.idKind0)∈PredefinedDefinition0

A <simple expression> shall contain only literals, and operations defined within the package Predefined, as defined in clause 14 of [ITU-T Z.104]. In the contexts where a <simple expression> is required, there shall only be identifiers for literals, operators or methods that are defined in package Predefined.

<agent instance pid value>.s-<name> ≠ undefined ⇒

<agent instance pid value>.s-<name> = take({s ∈ Sdl-specification}).s-Agent-definition.s-Agent-name

If a <name> is given in an <agent instance pid value>, it shall be the name of the system.

Transformations

<binary expression>(x,<implies sign>,y) =8=> <operator application>("=>",<x,y>)

<binary expression>(x,or,y) =8=> <operator application>("or",<x,y>)

<binary expression>(x,xor,y) =8=> <operator application>("xor",<x,y>)

<binary expression>(x,and**,**y) =8=> <operator application>("and",<x,y>)

<binary expression>(x,<greater than sign>,y) =8=> <operator application>(">",<x,y>)

<binary expression>(x,<greater than or equals sign>,y) =8=> <operator application>(">=",<x,y>)

<binary expression>(x,<less than sign>,y) =8=> <operator application>("<",<x,y>)

<binary expression>(x,<less than or equals sign>,y) =8=> <operator application>("<=",<x,y>)

<binary expression>(x,in**,**y) =8=> <operator application>("in",<x,y>)

<binary expression>(x,<plus sign>,y) =8=> <operator application>("+",<x,y>)

<binary expression>(x,<hyphen>,y) =8=> <operator application>("-",<x,y>)

<binary expression>(x,<concatenation sign>,y) =8=> <operator application>("//",<x,y>)

<binary expression>(x,<asterisk>,y) =8=> <operator application>("\*",<x,y>)

<binary expression>(x,<solidus>,y) =8=> <operator application>("/",<x,y>)

<binary expression>(x,mod,y) =8=> <operator application>("mod",<x,y>)

<binary expression>(x,rem,y) =8=> <operator application>("rem",<x,y>)

<operand5>(<hyphen>,x) =8=> <operator application>("-",<x>)

<operand5>(not,x) =8=> <operator application>("not",<x>)

An expression of the form

<expression> <infix operation name> <expression>

is derived syntax for

<quotation mark> <infix operation name> <quotation mark> (<expression>, <expression>)

where <quotation mark> <infix operation name> <quotation mark> represents an Operation-name.

Similarly,

<monadic operation name> <expression>

is derived syntax for

<quotation mark> <monadic operation name> <quotation mark> ( <expression> )

where <quotation mark> <monadic operation name> <quotation mark> represents an Operation-name.

If the <Natural<expression> of an <agent instance pid value> is omitted, this represents the Natural number "1". If the <Natural<expression> value of an <agent instance pid value> is zero or greater than the number of active instances of the agent instance set, the <agent instance pid value> represents the Pid value Null. See clause 12.2.1 of [ITU‑T Z.104].

Special visibility rules apply for an <agent<name> within an <agent instance>. The <agent instance> list before the <agent<name> in an <agent instance> identifies the heirarchical context for the <agent<name>. The <agent<name> is visible if it is visible in the last item of the <agent instance> list to the immediate left of the <agent<name>, or if it visible in the system. The <agent instance> list items to the left can be omitted provided the agent instance is uniquely identifiable (in the same way as omitting parts of a <qualifier> of an <identifier>) and each omitted <agent instance> list item represents the correponding Agent-name with an Instance-number for the Natural number "1". See clause 12.2.1 of [ITU‑T Z.104].

##### F2.2.9.9.2 Literal

See clause 12.2.2 *Concrete grammar* of [ITU‑T Z.101].

Abstract syntax

Literal :: Literal-identifier

Concrete syntax

<literal> = <literal identifier>

<literal identifier> :: <qualifier> <literal name>

Conditions on concrete syntax

Further study needed to formulate the condition.

It shall be possible to bind each unqualified <literal identifier> to exactly one defined *Literal-identifier* that satisfies the conditions in the construct in which the <literal identifier> is used.

Mapping to abstract syntax

| <literal identifier>(qual, name) then mk-Literal-identifier(Mapping(qual), Mapping(name))

Whenever a <literal identifier> is specified, the unique *Literal-name* in *Literal-identifier* is derived in the same way, with the result sort derived from context. A *Literal-identifier* is derived from context (see clause 6.2) so that if the <literal identifier> is overloaded (that is, the same name is used for more than one literal or operation), then the *Literal-name* identifies a visible literal with the same name and result sort consistent with the literal. If there are two literals with the same <name> but differing by result sorts, each has a different *Literal-name*.

Wherever a <qualifier> of a <literal identifier> ends with a <path item> with the keyword **type**, then the <sort name> is used as the result sort to derive the unique *Name* of the *Identifier*. The *Qualifier* is formed in the usual way from <qualifier>, therefore the <sort name> after the keyword **type** is used for both the *Qualifier* and deriving the *Name* of the *Identifier*.

Further study needed to distinguish the case of the qualifier being empty (and the sort determined by context) and qualifier specifying the sort.

##### F2.2.9.9.3 Synonym

Concrete syntax

<synonym> :: <synonym<identifier>

Transformations

<synonym>(ident)

provided ident.refersto0 ∈ <internal synonym definition item>

=8=>

ident.refersto0.s-<constant expression>

A <synonym> represents the <constant expression> defined by the <synonym definition> identified by the <synonym identifier>. An <identifier> used in the <constant expression> represents an Identifier in the abstract syntax according to the context of the <synonym definition>.

Further study needed because synonym semantics has changed.

##### F2.2.9.9.4 Extended primary

Concrete syntax

<extended primary> =

<indexed primary>

| <field primary>

| <composite primary>

<indexed primary> :: <primary> <actual parameter list>

<field primary> :: [<primary>] { <field name> | <field number> | <as signal> }

<field name> = <name>

<field number> = <Natural<name>

<composite primary> :: <qualifier> <actual parameter list>

Conditions on concrete syntax

∀fp∈<field primary>:

(fp.s-<primary> = undefined ⇒ fp.s-implicit ∈ <field name>)

The condition is needed because AS0 is simplified with respect to clause 12.2.3 *Concrete grammar* of [ITU‑T Z.104] where <primary> is only optional when a <field name> is given.

Transformations

<indexed primary>(prim, params)

=8=> mk-<method application>(prim, mk-<identifier>(*empty*, "Extract"), params)

An <indexed primary> is derived concrete syntax for

Extract (<primary>, <actual parameter list>)

See clause 12.2.3 *Model* of [ITU‑T Z.101]. The abstract syntax is determined from this concrete expression according to clause 12.2.1 of [ITU‑T Z.101].

<field primary>(prim, field)

provided prim ≠ undefined ∧ field ∉ (<field number> ∪ <as signal> )

=8=> mk-<method application>(prim, modifyExtractName(field, "Extract"), *empty*)

A <field primary> is derived concrete syntax for

*field-extract-name*(<primary> )

where the *field-extract-name* is formed from the concatenation of the field name and "Extract" in that order. The abstract syntax is determined from this concrete expression according to clause 12.2.1 of [ITU‑T Z.101]. See clause 12.2.3 *Model* of [ITU‑T Z.101]. The transformation according to this model is performed before the modification of the signature of methods in clause 12.1.3 of [ITU‑T Z.104].

<field primary>(prim, fieldnum)

provided prim ≠ undefined ∧ fieldnum ∈ <field number>

=8=>

let td = mk-<operand5>(undefined, prim).staticSort0.refersto0 in // type definition

mk-<field primary>(prim,

fieldNameList0( // list of field names for the data type definition

if td ∈ <data type definition> then td

else td.parentDataType0 // td is syntype definition, get the parent data type

endif.s-<data type definition body>.s-implicit

)[intTokenToNat(fieldnum)] // field number selects field name from field name list

) // revised field primary

endlet // td

A <field number> is a shorthand for the <field name> of a field of the sort of <field primary> or <field variable>, where the value 1 represents the first field, the value 2 the second field and the value n the nth field. See clause 12.2.3 *Model* of [ITU‑T Z.104].

<field primary>(prim, asSig)

provided prim ≠ undefined ∧ asSig ∈ <as signal>

=8=>

<field primary>(prim, asSig.s-<identifier>.asSignal.s-<name>)

An <as signal> in a <field primary> is a shorthand for the <field name> of a field of the sort of the <field primary>, where the <field name> is the same as the unique anonymous name of the structure data type defined by the <signal definition> identified by the <signal identifier> of <as signal>. See clause 12.2.3 *Model* of [ITU‑T Z.104].

<field primary>(undefined, field)

=8=> mk-<field primary>(mk-<varible access>(this), field)

When the <field primary> has the form <field name>, this is derived syntax for:

this ! <field name>

See clause 12.2.3 *Model* of [ITU‑T Z.104].

<composite primary>(qual, params)

=8=>

mk-<operator application>( mk-<identifier>(qual, "Make"), params)

A <composite primary> is derived concrete syntax for:

<qualifier> Make ( <actual parameter list> )

if any actual parameters were present, or

<qualifier> Make

otherwise, and where the <qualifier> is inserted only if it was present in the <composite primary>. See clause 12.2.3 *Model* of [ITU‑T Z.101]. The abstract syntax is determined from this concrete expression according to clause 12.2.1 of [ITU‑T Z.101].

##### F2.2.9.9.5 Equality expression

Abstract syntax

Equality-expression = Positive-equality-expression | Negative-equality-expression

Positive-equality-expression :: First-operand Second-operand

Negative-equality-expression :: First-operand Second-operand

First-operand = Expression

Second-operand = Expression

Concrete syntax

<equality expression> ::

<expression> { <equals sign> | <not equals sign> } <expression>

Conditions on concrete syntax

∀ee∈<equality expression>:

(let set1 = ee.s-<expression>.staticSortSet0 in

let set2 = ee.s2-<expression>.staticSortSet0 in

∃s1∈set1:∃s2∈set2:isSortCompatible0(s1, s2)∨ isSortCompatible0(s2, s1)

endlet)

An <equality expression> is legal concrete syntax only if the sort of one of its operand is sort compatible to the sort of the other operand.

Mapping to abstract syntax

| <equality expression>(first, <equals sign>, second) then

mk-Positive-equality-expression(Mapping(first), Mapping(second))

| <equality expression>(first, <not equals sign>, second) then

mk-Negative-equality-expression(Mapping(first), Mapping(second))

##### F2.2.9.9.6 Conditional expression

Abstract syntax

Conditional-expression :: Boolean-expression

Consequence-expression

Alternative-expression

Consequence-expression = Expression

Alternative-expression = Expression

Conditions on abstract syntax

∀c ∈ Conditional-expression:

(c.s-Consequence-expression*.*staticSort1= c.s-Alternative-expression.staticSort1) ∧

(c.s-Boolean-expression.staticSort1= predefinedId1("Boolean"))

For any Conditional-expression, the sort of the Consequence-expression must be the same as that of the Alternative-expression, and the sort of a Boolean-expression must be Boolean.

Concrete syntax

<conditional expression> ::

<Boolean<expression> <consequence expression> <alternative expression>

<consequence expression> = <expression>

<alternative expression> = <expression>

Conditions on concrete syntax

∀ce∈<conditional expression>:

let set1=ce.s-<consequence expression>.staticSortSet0 in

let set2=ce.s-<alternative expression>.staticSortSet0 in

|set1| =1 ∧ |set2| =1∧ isSameSort0(set1.take, set2.take)

endlet

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) then

mk-Conditional-expression(Mapping(e1), Mapping(e2), Mapping(e3))

##### F2.2.9.9.7 Operation application

Abstract syntax

Operation-application :: Operation-identifier *Actual-parameters*

Conditions on abstract syntax

∀oa ∈ Operation-application:

let os = getEntityDefinition1(oa, operation) in

isActualAndFormalParameterMatched1(oa.s-Expression-seq, os.formalParameterSortList1)

endlet

The Operation-identifierin the Operation-applicationmust be visible. Each Expression in the list of Expressions after the Operation-identifiermust be sort compatible to the corresponding (by position) sort in the list of Formal-argument*s* of the Operation-signature.

Concrete syntax

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

<operator application> :: <operation identifier> [ <actual parameters> ]

<method application> :: <primary> <operation identifier> [ <actual parameters> ]

Conditions on concrete syntax

∀methodApp∈<method application>:

getEntityDefinition0(methodApp.s-<identifier>, method) ≠ undefined

A <method application> is legal concrete syntax only if <operation identifier> represents a method.

Transformations

<method application>(prim, ident, params) =8=>

<operator application>(ident, < prim > ⁀ params)

The concrete syntax form

<expression> <full stop> <operation identifier> [<actual parameters>]

is derived concrete syntax for

<operation identifier> new-actual-parameters

where new-actual-parameters is <actual parameters> containing only <expression>, if <actual parameters> was not present; otherwise new-actual-parameters is obtained by inserting <expression> before the first optional expression in <actual parameters>.

Mapping to abstract syntax

<operator application>(ident, params) then

mk-Operation-application(Mapping(ident),Mapping(params))

Auxiliary functions

Get the actual parameter list associated with the Operation-identifier.

actualParameterListOfOpId1(id: Operation-identifier): [Expression]\*=def

case id.parentAS1 of

| Open-range then <id.parentAS1.s-Constant-expression>

| Operation-application then < exp | exp in id.parentAS1.s-Expression-seq>

endcase

##### F2.2.9.9.8 Range check expression

Abstract syntax

Type-check-expression :: Expression Parent-sort-identifier

Range-check-expression :: Expression Parent-sort-identifier Range-condition

Conditions on abstract syntax

∀rce ∈ Range-check-expression:

( isCompatibleTo1(rce.s-Expression.staticSort1, rce.s-Parent-sort-identifier) )

∨ ( getEntityDefinition1(rce.s-Parent-sort-identifier, interface) ≠ undefined

∧ isCompatibleTo1(rce.s-Expression.staticSort1, Mapping(predefinedId0("Pid"))))

The sort of the *Expression* of a Range-check-expression shall be sort compatible with the sort identified by the *Parent-sort-identifier*, or the sort Pid if the *Parent-sort-identifier* is a pid sort. See clause 12.1.7 *Abstract grammar* [ITU‑T Z.101].

Concrete syntax

<range check expression> ::

<expression>

{ <range check constrained sort> | <syntype> | <pid sort> | <datatype<type expression> }

<range check constrained sort> :: <sort<identifier> <constraint>

Conditions on concrete syntax

∀rccs ∈<range check constained sort>:

rccs.s-<identifier>.fullIdentifier0 ≠ predefinedId0("Pid")

∨ getEntityDefinition0(rccs.s-<identifier>,interface) ≠ undefined

The <sort<identifier> of a <range check constrained sort> shall not be Pid or a pid sort. See clause 12.1.7 *Concrete grammar* [ITU‑T Z.101].

Mapping to abstract syntax

| <range check expression>(expr, srt) then

case srt of

| <range check constrained sort> then

mk-Range-check-expression(

Mapping(expr),

Mapping(srt.s-<identifier>),

Mapping(srt.s-<constraint>)

)

| <syntype> then

mk-Range-check-expression(

Mapping(expr),

Mapping(srt),

Mapping(srt.refersto0.s-implicit.s-<constraint>)

)

| <pid sort> then

mk-Range-check-expression(

Mapping(expr),

Mapping(srt),

mk-Range-check-expression(∅)

)

| <datatype<type expression> then

mk-Type-check-expression(

Mapping(expr),

Mapping(srt),

)

endcase // srt

The <expression> represents the *Expression*. If the form <range check constrained sort> is used, the <sort<identifier> represents the *Parent-sort-identifier* that applies to the <constraint>, which represents the *Range-condition* as described in clause 12.1.8.2. If the <sort<identifier> in <range check constrained sort> identifies a syntype, this is the same as specifying the parent sort identifier of the syntype, and the <constraint> is not restricted to the range of the syntype. If the form <syntype> is used, *Parent-sort-identifier* and *Range-condition* are the *Parent-sort-identifier* and *Range-condition* (respectively) of the identified syntype. If the form <pid sort> is used, it determines the *Parent-sort-identifier* and the *Range-condition* is empty. See clause 12.1.7 *Concrete grammar* [ITU‑T Z.101] (with <operand2> replaced by <expression>).

If the form <datatype<type expression> is used, the <range check expression> represents a *Type-check-expression*. The <expression> represents the *Expression*. The <datatype type expression> determines the *Parent-sort-identifier*. See clause 12.1.7 *Concrete grammar* [ITU‑T Z.107] (with <operand2> replaced by <expression>)

##### F2.2.9.9.9 Variable definition

Abstract syntax

Variable-definition :: Variable-name

Sort-reference-identifier

Aggregation-kind

[ Constant-expression ]

Aggregation-kind = PART | REF

Conditions on abstract syntax

∀d ∈ Variable-definition: d*.*s-Constant-expression≠ undefined ⇒

d.s-Constant-expression. staticSort1= d.s-Sort-reference-identifier

If the Constant-expression is present, it must be of the same sort as the one denoted by   
Sort-reference-identifier.

Concrete syntax

<variable definition> :: <variables of sort>+ | <exported variables of sort>+

<variables of sort> ::

<aggregation kind> <variable<name>+ <sort> [<constant expression>]

<exported variables of sort> ::

<aggregation kind> <exported variable>+ <sort> [<constant expression>]

<exported variable> :: <variable<name> <remote variable<identifier>

<aggregation kind> :: [ part | ref ]

Further study is needed for <aggregation kind> and all the places it is used (not just variable definition). The <aggregation kind> has been added for <variables of sort> but there are many other places where it needs to be added. If <aggregation kind> is empty or part the Aggregation-kind is PART. If <aggregation kind> is ref the Aggregation-kind is REF. Needs to be written below as a Mapping.

Conditions on concrete syntax

∀d∈<agent definition> ∪<agent type definition>:

¬(∃v1,v2∈<exported variable>: v1 ≠ v2 ∧ v1.s-<identifier> = v2.s-<identifier> ∧

v1. parentAS0.parentAS0∈<variable definition> ∧

v2. parentAS0.parentAS0∈<variable definition>∧

v1.surroundingScopeUnit0 = d ∧ v2.surroundingScopeUnit0 = d)

Two exported variables in an agent cannot mention the same <remote variable identifier>.

Transformations

<variables of sort> (ak, < n > ⁀ rest, sort, expr) provided rest ≠ empty =5=>

<variables of sort> (ak, < n >, sort, expr) ⁀ <variables of sort> (ak, rest, sort, expr)

A <variables of sort> that defines multiple variables is a shorthand for a sequence of <variables of sort> items, each defining one variable.

<exported variables of sort> (ak, < ev > ⁀ rest, sort, expr) provided rest ≠ empty =5=>

<exported variables of sort> (ak, < ev >, sort, expr) ⁀ <exported variables of sort> (ak, rest, sort, expr)

An <exported variables of sort> that defines multiple variables is a shorthand for a sequence of <exported variables of sort> items, each defining one variable.

<variable definition>(< v > ⁀ rest) provided rest ≠ empty =5=>

<variable definition> (< v >) ⁀ <variable definition> (rest)

A <variable definition> that defines multiple <variables of sort> or <exported variables of sort> items is a shorthand for a sequence of <variable definition>s, each defining one item. Combined with the transforms for <variables of sort> and <exported variables of sort>, the consequence is each <variable definition> defines one variable.

Mapping to abstract syntax

| <variable definition>(\*, < var >) then Mapping(var)

| <variables of sort>(ak, <name>, sort, const) then

mk-Variable-definition(Mapping(name), Mapping(sort), Mapping(ak), Mapping(const))

| <exported variables of sort>(ak, <ev>, sort, const) then

mk-Variable-definition(Mapping(ev.s-<name>), Mapping(sort), Mapping(ak), Mapping(const))

##### F2.2.9.9.10 Variable access

Abstract syntax

Variable-access :: Variable-identifier

Concrete syntax

<variable access> :: { <variable<identifier> | **this** | <import expression> }

Conditions on concrete syntax

∀va∈<variable access>: va.s-implicit = this ⇒

(parentAS0ofKind(*va*, <operation definition>) ≠ undefined ∧

parentAS0ofKind(*va*, <operation definition>).kind0**=method**) // before transformation to a procedure

∨ (∃*od* ∈ <operation definition> :

*od*.operatorProcedureId ≠ undefined ∧ *od*.kind0**=method**

∧ *od*.operatorProcedureId.refersto0 **=** parentAS0ofKind(*va*, <internal procedure definition >)

) // after method body transformed into an anonymous procedure

this must only occur in method definitions.

Transformations

va = <variable access>(this)

provided parentAS0ofKind(*va*, <internal procedure definition >) ≠ undefined // anon proc for method

=8=> <variable access>(parentAS0ofKind(va, <internal procedure definition >).s-<procedure heading>.

s-<formal variable parameters>.head.s-<parameters of sort>.s-<name>.head)

A <variable access> using the keyword **this** is replaced by the anonymous name introduced as the name of the leading parameter in <arguments> according to clause 12.1.3 *Model* of [ITU-T Z.104]. See clause 12.3.2 *Model* of [ITU-T Z.104].

Mapping to abstract syntax

| <variable access>(identifier) then mk-Variable-identifier(Mapping(identifier))

##### F2.2.9.9.11 Assignment

Abstract syntax

Assignment :: Variable-identifier Expression

Conditions on abstract syntax

∀a ∈ Assignment: ∃d ∈ Variable-definition:

(d=getEntityDefinition1(a.s-Variable-identifier,variable)) ∧

isCompatibleTo1(a.**s**-Expression.staticSort1, d.**s**-Sort-reference-identifier)

In an Assignment, the sort of the Expression must be sort compatible to the sort of the Variable-identifier.

Concrete syntax

<assignment> :: <variable> <expression>

<variable> = <variable<identifier> | <extended variable>

Mapping to abstract syntax

| <assignment>(var,expr) then mk-Assignment(var,expr)

##### F2.2.9.9.12 Extended variable

Concrete syntax

<extended variable> = <indexed variable> | <field variable>

<indexed variable> :: <variable> <actual parameter list>

<field variable> :: <variable> { <field name> | <field number> | <as signal> }

Transformations

<assignment>(<indexed variable>(var, params), expr)

=8=>

<assignment>(var, <method application>(var, <identifier>(*empty*, "Modify"), expr))

<indexed variable> is derived concrete syntax for

<variable> <is assigned sign> <variable> <full stop> Modify ( expressionlist )

where expressionlist is constructed by appending <expression> to the <actual parameter list>. The abstract grammar is determined from this concrete expression according to clause 12.2.1 of [ITU‑T Z.101]. The same model applies to the second form of <indexed variable>.

<assignment>(<field variable>(var, fieldname), expr)

=8=>

<assignment>

(var, <method application>(var, modifyExtractName(fieldname, "Modify"), expr))

The concrete syntax form

<variable> <exclamation mark> <field name> <is assigned sign> <expression>

is derived concrete syntax for

<variable> <full stop> field-modify-operation-name ( <expression> )

where the field-modify-operation-name is formed from the concatenation of the field name and "Modify". The abstract syntax is determined from this concrete expression according to clause 12.2.1 of [ITU-T Z.101]. The same model applies to the second form of <field variable>.

Auxiliary functions

modifyExtractName(name: <identifier>, suffix: Token ):<identifier> =def

mk-<identifier>(name.s-<qualifier>, name.s-<name> + suffix)

##### F2.2.9.9.13 Default initialization

Abstract syntax

Default-initialization = Constant-expression

Concrete syntax

<default initialization> :: [<virtuality>] [<constant expression>]

Conditions on concrete syntax

∀init∈<default initialization>:

init.s-<constant expression> = undefined ⇒init.virtuality0 ∈{redefined, finalized}

The <constant expression> shall only be omitted if <virtuality> is **redefined** or **finalized**.

Transformations

<variables of sort>(ak, names, sort, undefined)

provided

getEntityDefinition0(sort, sort).getDefaultInitialization0 ≠ undefined

=8=>

<variables of sort>( ak, names, sort,

getEntityDefinition0(sort, sort). getDefaultInitialization0)

Further study needed because in SDL-2010 there should be a mapping to *Default-initialization*, which should be used to determine the *Constant-expression* of a *Variable-definition*, and the range initialization of a syntype should be checked.

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 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 getDefaultInitialization0 computes the default initialization for a data type or syntype.

getDefaultInitialization0(type: <data type definition> ∪ <syntype definition>): <default initialization> =def

case type in

| <data type definition> then

let tb = type.s-<data type definition body> in

if tb ≠ undefined then

let di = tb.s-<default initialization> ∧ di.s-<constant expression> in

if di ≠ undefined then

if di.s-<constant expression> ≠ undefined then di else undefined endif

else

type.s-<data type specialization>.s-<base type>-seq.baseInitialization0

endif

endlet

else sds.s-<parent sort identifier>.getDefaultInitialization0

endif

endlet

| <syntype definition> then

let sds = type.s-<syntype definition syntype> in

let di = sds.s-<default initialization> in

if di ≠ undefined then di

else sds.s-<parent sort identifier>.getDefaultInitialization0

endif

endlet

endlet

endcase

The function baseInitialization0 returns the <default initialization> for the first <base type> (in the list of a <data type specialization>) that has a <default initialization> or undefined if there is no <default initialization>.

baseInitialization0(bl: <base type>\*): <default initialization> =def

if bl ≠ undefined

then

let di = bl.head.getDefaultInitialization0 in

if di ≠ undefined then di

else

let blt = bl.tail in

if blt.length ≠ 0 then blt.baseInitialization0 else undefined endif

endif

else undefined

endif

##### F2.2.9.9.14 Now expression

Abstract syntax

Now-expression :: ()

Concrete syntax

<now expression> :: ()

Mapping to abstract syntax

| <now expression> then mk-Now-expression()

##### F2.2.9.9.15 Import expression

Concrete syntax

<import expression> :: <remote variable<identifier> <communication constraints>

Conditions on concrete syntax

∀exp∈<import expression>: exp.s-<identifier>∈

(let d=parentAS0ofKind(exp, <agent definition> ∪<agent type definition> ) in

exp.s-<identifier>∈d.validOutputSignalSet0

endlet)

A remote variable mentioned in an <import expression> shall be in the complete output set of an enclosing agent type or agent set. See clause 10.6 *Concrete grammar* of [ITU‑T Z.102].

NOTE – The <destination> in the <communication constraints> of an <import expression> has to be compatible with the remote variable, which is described in more detail in clause F2.2.7.6, Remote variable.

Transformations

The import expression has implied syntax for the importing of the result as defined in clause 10.6 of [ITU-T Z.102] and it also has an implied Variable-access of the implied variable for the import in the context where the <import expression> appears. See clause 12.3.4.5 *Model* of [ITU-T Z.104].

NOTE – The transformation of <import expression> is described in clause F2.2.7.6, Remote variable.

##### F2.2.9.9.16 Pid expression

Abstract syntax

Pid-expression = Self-expression

| Parent-expression

| Offspring-expression

| Sender-expression

Self-expression :: ()

Parent-expression :: ()

Offspring-expression :: ()

Sender-expression :: ()

Concrete syntax

<pid expression> =

<self expression>

| <parent expression>

| <offspring expression>

| <sender expression>

<self expression> :: ()

<parent expression> :: ()

<offspring expression> :: ()

<sender expression> :: ()

<create expression> = <create body>

Transformations

The use of <create expression> in an expression is a shorthand for inserting a create request just before the action where the <create expression> occurs followed by an assignment of offspring to an implicitly declared anonymous variable of the same sort as the static sort of the <create expression>. The implicit variable is then used in the expression. If <create expression> occurs several times in an expression, one distinct variable is used for each occurrence. In this case the order of the inserted create requests and variable assignments is the same as the order of the <create expression>s.

If the <create expression> contains an <agent type identifier> then the transformations that are applied to a create statement that contains an <agent type identifier> are also applied to the implicit create statements resulting from the transformation of a <create expression> (see clause 11.13.2 of [ITU‑T Z.103]).

Mapping to abstract syntax

| <self expression> then mk-Self-expression()

| <parent expression> then mk-Parent-expression()

| <offspring expression> then mk-Offspring-expression()

| <sender expression> then mk-Sender-expression()

##### F2.2.9.9.17 Timer active expression and timer remaining duration

Abstract syntax

Timer-active-expression :: Timer-identifier Expression\*

Timer-remaining-duration :: Timer-identifier Expression\*

Conditions on abstract syntax

∀t ∈ Timer-active-expression ∪ Timer-remaining-duration:

let d = getEntityDefinition1(t.s-Timer-identifier, timer) in

t.s-Expression.length = d.s-Sort-reference-identifier. length ∧

(∀i ∈ 1.. t.s-Expression. length:

isCompatibleTo1( t.s-Expression[i].staticSort1, d.s-Sort-reference-identifier[i]))

endlet

The sorts of the Expression list in the Timer-active-expression or Timer-remaining-duration shall correspond by position to the Sort-reference-identifier list directly following the Timer-name identified by the Timer-identifier.

Concrete syntax

<timer active expression> :: <timer<identifier> [ <expression list> ]

<timer remaining duration> :: <timer<identifier> [ <expression list> ]

Mapping to abstract syntax

| <timer active expression>(id,l) then

mk-Timer-active-expression(Mapping(id), Mapping(l))

| <timer remaining duration>(id,l) then

mk-Timer-remaining-duration(Mapping(id), Mapping(l))

##### F2.2.9.9.18 Active agents expression

Abstract syntax

Active-agents-expression :: { Agent-identifier | **THIS** }

Concrete syntax

<active agents expression> ::{ <agent<identifier> | **this** }

Conditions on abstract syntax

∀exp∈<create body>: (exp.s-implicit = this) ⇒

(exp.surroundingScopeUnit0∈<agent type definition> ) ∧

(exp.surroundingScopeUnit0.surroundingScopeUnit0∈<agent type definition> )

this shall only be specified in an <agent type diagram> and in scopes enclosed by an <agent type diagram> and represents **THIS**.

Mapping to abstract syntax

| <active agents expression>(id=<identifier>) then mk-Active-agents-expression(Mapping(id))

| <active agents expression>( **this**) then mk-Active-agents-expression(**THIS**)

##### F2.2.9.9.19 Any expression

Abstract syntax

Any-expression :: Sort-reference-identifier

Concrete syntax

<any expression> :: <sort>

Conditions on abstract syntax

∀exp∈<any expression>: isContainingElements0(exp.s-<sort>)

The <sort> must contain elements.

Mapping to abstract syntax

| <any expression>() then mk-Any-expression

Auxiliary functions

isContainingElements0(s:<sort>):Boolean=def

let d = getEntityDefinition0(s, sort) in

(d∈PredefinedSort0)∨

(∃cons∈<data type constructor>: cons.surroundingScopeUnit0 = d ∧

(cons∈<structure definition>∪<choice definition>⇒

∀sort∈<sort>:sort in cons.fieldSortList0⇒isContainingElements0(sort)))∨

(d.specialization0≠undefined∧

isContainingElements0(d.specialization0.s-<type expression>.baseType0)∧

(∀acp∈<actual context parameter>:

acp in

d.actualContextParameterList0∧acp.idKind0 = sort⇒ isContainingElements0(acp))

endlet

##### F2.2.9.9.20 State expression

Abstract syntax

State-expression :: ()

Concrete syntax

<state expression> :: ()

Mapping to abstract syntax

| <state expression>() then mk-State-expression

##### F2.2.9.9.21 Signal expression

Abstract syntax

Signal-expression :: ()

Concrete syntax

<signal expression> :: ()

Mapping to abstract syntax

| <signal expression>() then mk-Signal-expression

##### F2.2.9.9.22 Signallist expression

Abstract syntax

Signallist-expression :: ()

Concrete syntax

<signallist expression> :: ()

Mapping to abstract syntax

| <signallist expression>() then mk-Signallist-expression

##### F2.2.9.9.23 Value returning procedure call

Abstract syntax

Value-returning-call-node :: [ **THIS** ]

Procedure-identifier

*Actual-parameters*

Concrete syntax

<value returning procedure call> =

<procedure call body>

| <remote procedure call body>

Conditions on concrete syntax

∀exp∈<continuous expression>: exp.parentAS0∈<continuous signal> ⇒

¬∃procCall∈<value returning procedure call>: isAncestorAS0(exp, procCall)

∀exp∈<provided expression>: exp. parentAS0∈<input part> ⇒

¬∃procCall∈<value returning procedure call>: isAncestorAS0 (exp, procCall)

A <value returning procedure call> must not occur in the <Boolean expression> of a   
<continuous signal> or <enabling condition>.

∀procId∈<identifier>: procId. parentAS0∈<value returning procedure call>⇒

getEntityDefinition0(procId, procedure).s-<procedure heading>.s-<procedure result> ≠ undefined

The <procedure identifier> in a <value returning procedure call> must identify a procedure having a <procedure result>.

∀procCall∈<procedure call body>:

procCall ∈<value returning procedure call> ∧ procCall.**s-**this ≠ undefined ⇒

getEntityDefinition0(procId, procedure) =

parentAS0ofKind(procCall, <procedure definition> )

If **this** is used, <procedure identifier> must denote an enclosing procedure.

Transformations

let nn= newName in

p=<value returning procedure call>(<procedure call body>(id, params) )

provided parentAS0ofKind(id.refersto0, <agent type definition>) ≠

parentAS0ofKind(p, <agent type definition>)

=8=>

let par=parentAS0ofKind(p, <agent type definition>) in

<value returning procedure call>(<procedure call body>(

<identifier>(par.fullQualifier0⁀<path item>(par.entityKind0, par.entityName0), nn),

params))

endlet

and // add the new definition

let defs=

parentAS0ofKind(p, <agent type definition>).s-<agent structure>.s-<entity in agent>-seq in

defs => defs ⁀

<internal procedure definition>(empty,

<procedure heading>(

<procedure preamble>(undefined, undefined),

empty, nn, empty, undefined, undefined, empty, undefined, empty),

empty,

<procedure body>(undefined, undefined, empty))

**endlet** // defs

**endlet** // nn

If the <procedure identifier> is not defined within the enclosing agent, the procedure call is transformed into a call of a local, implicitly created, subtype of the procedure.

The keyword **this** implies that when the procedure is specialized, the <procedure identifier> is replaced by the identifier of the specialized procedure.

Mapping to abstract syntax

| <value returning procedure call>(<procedure call body>(t, id, params)) then

mk-Value-returning-call-node(Mapping(t), Mapping(id), Mapping(params))

##### F2.2.9.9.24 Type coercion

Abstract syntax

Type-coercion :: Expression Sort-reference-identifier

Conditions on abstract syntax

The static sort of Expressionshall be the sort identified by Sort-reference-identifier, or a parent sort of that sort.

∀ t ∈ Type-coercion:isCompatibleTo1(staticSort1(t.s-Expression), t.s-Sort-reference-identifier)

Concrete syntax

<type coercion> :: <expression> <sort>

Mapping to abstract syntax

| <type coercion>(e, s) then mk-Type-coercion(Mapping(e), Mapping(s))

##### F2.2.10 Computing constants

This clause contains auxiliary functions used to compute values for constants.

Auxiliary functions

value1 returns a member of Value1 (a Literal-signature).

value1(e: Constant-expression):Value1 =def

case e of

| Literal(lid) then computeLiteral1(lid)

| Conditional-expression(bool, consequence, alternative) then

if bool.value1.semvalueBool1 then consequence.value1 else alternative.value1 endif

| Positive-equality-expression(a, b) then computeEquality1(a.value1, b.value1)

| Negative-equality-expression(a, b) then computeInEquality1(a.value1, b.value1)

| Operation-application(op, values) then computeConstant1(op, < v.value1: v in values>)

| Range-check-expression(range, expr) then expr.value1 ∈ range.range1

| Type-check-expression(expr, parent) then

isCompatibleTo1(getEntityDefinition1(expr.value1.s-Result.s-Sort-reference-identifier, sort),

getEntityDefinition1(parent.s-Result.s-Sort-reference-identifier, sort))

| Type-coercion(expr, \*) then expr.value1

| Agent-instance-pid-value(ail) then computeAgentPid1(ail)

endcase

computeLiteral1 returns the Literal-signature corresponding to the literal.

computeLiteral1(id: Literal-identifier): Value1 =def

getEntityDefinition1(id, idKind1(id) ).s-Literal-signature

semvalueBool1 returns the Boolean value true if the Literal-signature is the Boolean value true, otherwise it returns the Boolean value false.

semvalueBool1(b: Literal-signature): Boolean =def

if b= mk-Literal-signature(mk-Name("true"), mk-Result(predefinedId1("Boolean"), PART), 0)

then true else false endif // Nat value for true is 0, Nat value for false is 1

computeEquality1 returns the Boolean Literal-signature for true if the value of both expressions is the same, and the Boolean Literal-signature for false otherwise.

computeEquality1(a: Literal-signature, b: Literal-signature): Literal-signature =def

mk-Literal-signature (

if a = b then mk-Name("true") else mk-Name("false") endif,

mk-Result(predefinedId1("Boolean"), PART),

if a = b then 0 else 1 endif)

computeInEquality1 returns the Boolean Literal-signature for false if the value of computeEquality1 is the Boolean Literal-signature for true, and the Boolean Literal-signature for true otherwise.

computeInEquality1(a: Literal-signature, b: Literal-signature): Literal-signature =def

let eq = computeEquality1 (a, b).s-Name.s-Token = "true" in

mk-Literal-signature (

if eq then mk-Name("false") else mk-Name("true") endif,

mk-Result(predefinedId1("Boolean"), PART),

if eq then 1 else 0 endif)

endlet

computeConstant1 returns the Literal-signature for the value of applying the operation to the constant expression argument values for the operation. The name of the data type is derived from the Operation-identifier: the operation is defined within a data type; therefore the last Path-item of the Qualifier of the Operation-identifier is the Data-type-name of this data type. If the part of the Qualifier of the Operation-identifier before the Data-type-name is the package Predefined, the data type is one of the predefined data types there is select ion of the appropriate compute function.

computeConstant1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study needed for the otherwise case.

if op.s-Qualifier.length = 2

∧ op.s-Qualifier.head ∈ Package-qualifier

∧ op.s-Qualifier.head.s-Package-name = "Predefined"

∧ op.s-Qualifier.last ∈ Data-type-qualifier

then // predefined, unparameterised data type

case op.s-Qualifier.last.s-Data-type-name in

| "Boolean" then computeBoolean1(op, values)

| "Character" then computeChar1(op, values) // alias "IA5Char"

| "Charstring" then computeCharstring1(op, values) // alias "IA5String"

| "Integer" then computeInteger1(op, values)

| "Real" then computeReal1(op, values)

| "Duration" then computeDuration1(op, values)

| "Time" then computeTime1(op, values)

| "Bit" then computeBit1(op, values)

| "Bitstring" then computeBitstring1(op, values)

// Octet is a syntype of Bitstring constrained to length 8 bits

| "Octetstring" then computeOctetstring1(op, values)

| "Pid" then computePid1(op, values)

// NumericChar is a syntype of Character constrained to numerical characters and SPACE

// NumericCharString is a syntype of Charstring constrained to numerical characters and SPACE

// PrintableChar is a syntype of Character constrained to printable characters and SPACE

// PrintableCharString is a syntype of Charstring constrained to printable characters and SPACE

| "TeletexChar" then computeTeletexChar1(op, values)

| "TeletexCharString" then computeTeletexCharString1(op, values)

| "VideotexChar" then computeVideotexChar1(op, values)

| "VideotexCharString" then computeVideotexCharString1(op, values)

| "UniversalChar" then computeUniversalChar1(op, values) // ISO/IEC 10646 character (32 bits)

| "UniversalCharString" then computeUniversalCharString1(op, values)

// UTF8String is a syntype of UniversalCharString

// GeneralChar is a syntype of UniversalChar constrained to G set, C set, SPACE and DEL

// GeneralCharString is a syntype of UniversalCharString constrained to G set, C set, SPACE and DEL

// GraphicChar is a syntype of UniversalChar constrained to G set and SPACE

// GraphicCharString is a syntype of UniversalCharString constrained to G set and SPACE

// VisibleChar is a syntype of Character constrained to visible characters and SPACE

// VisibleString is a syntype of Charstring constrained to visible characters and SPACE

// BMPChar is a syntype of UniversalChar constrained to the Basic Multilingual Plane (16 bit characters)

// BMPCharString is a syntype of UniversalCharString constrained to the Basic Multilingual Plane

| "NULL" then

mk-Literal-signature(mk-Name("null"), mk-Result(predefinedId1("NULL"), PART), 0)

otherwise undefined // should not occur

endcase

else // not predefined, unparameterised data type

if isSpecialLiteralOp1(op) then computeLiteralOp1(op, values)

elseif isSpecialStructOp1(op) then computeStructOp1(op, values)

elseif isSpecialChoiceOp1 (op) then computeChoiceOp1(op, values)

else undefined // This covers all other operations that do not have an active primary

// as a parameter, but in general it is not practical to interpret the operation to determine the result.

endif

endif // is it predefined, unparameterised data type

computeBoolean1 returns (to computeConstant1) the Literal-signature for the value of applying the Boolean operation to the constant expression argument values.

computeBoolean1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Nat value for true is 0, Nat value for false is 1

let truelt=mk-Literal-signature(mk-Name("true"),mk-Result(predefinedId1("Boolean"),PART),0) in

let falslt=mk-Literal-signature(mk-Name("false"),mk-Result(predefinedId1("Boolean"), PART),1)in

case op.s-Name in

| """not""" then //

if values.length = 1

then

if values.head = truelt then falslt else truelt endif

else undefined //should not occur

endif

| """and""" then

if values.length = 2

then

if values.head = truelt ∧ values.last = truelt then truelt

else falslt

endif

else undefined //should not occur

endif

| """or""" then

if values.length = 2

then

if values.head = truelt ∨ values.last = truelt then truelt

else falslt

endif

else undefined //should not occur

endif

| """xor""" then

if values.length = 2

then

if values.head ≠ values.last = true then truelt

else falslt

endif

else undefined //should not occur

endif

| """=>""" then

if values.length = 2

then

if values.head = truelt ∧ values.last = falslt then falslt

else truelt

endif

else undefined //should not occur

endif

otherwise //should not occur

endcase

endlet // falslt

endlet // truelt

computeCharacter1 returns (to computeConstant1) the Literal-signature for the value of applying the Character operation to the Character constant expression argument values.

computeCharacter1(op: Operation-identifier, values: Value1\*): Literal-signature =def

case op.s-Name in

| "chr" then // the Character for any given integer value

if values.length = 1 ∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

then // parameter is an Integer literal for which the literal natural is 2\*n for n>0 and -2\*n-1 for n<0

case intval = if (values.head.s-Nat/2)\*2 = values.head.s-Nat // is it even

then values.head.s-Nat/2 // yes divide by 2

else -(values.head.s-Nat +1) /2

endif

in

| < 0 then computeCharacter1(op,

Literal-signature(natToIntToken(intval + 128),

mk-Result(predefinedId1("Integer"),PART),

values.head.s-Nat - if intval + 128 > 0 then 257 else 256 endif))

| > 128 then computeCharacter1(op,

Literal-signature(natToIntToken(intval - 128),

mk-Result(predefinedId1("Integer"),PART),

values.head.s-Nat-256))

// Each name is enclosed in a QUOTATION MARK pair

| 0 then Literal-signature(mk-Name("NUL"),mk-Result(predefinedId1("Character"),PART),0)

| 1 then Literal-signature(mk-Name("SOH"),mk-Result(predefinedId1("Character"),PART),1)

| 2 then Literal-signature(mk-Name("STX"),mk-Result(predefinedId1("Character"),PART),2)

| 3 then Literal-signature(mk-Name("ETX"),mk-Result(predefinedId1("Character"),PART),3)

| 4 then Literal-signature(mk-Name("EOT"),mk-Result(predefinedId1("Character"),PART),4)

| 5 then Literal-signature(mk-Name("ENQ"),mk-Result(predefinedId1("Character"),PART),5)

| 6 then Literal-signature(mk-Name("ACK"),mk-Result(predefinedId1("Character"),PART),6)

| 7 then Literal-signature(mk-Name("BEL"),mk-Result(predefinedId1("Character"),PART),7)

| 8 then Literal-signature(mk-Name("BS"),mk-Result(predefinedId1("Character"),PART),8)

| 9 then Literal-signature(mk-Name("HT"),mk-Result(predefinedId1("Character"),PART),9)

| 10 then Literal-signature(mk-Name("LF"),mk-Result(predefinedId1("Character"),PART),10)

| 11 then Literal-signature(mk-Name("VT"),mk-Result(predefinedId1("Character"),PART),11)

| 12 then Literal-signature(mk-Name("FF"),mk-Result(predefinedId1("Character"),PART),12)

| 13 then Literal-signature(mk-Name("CR"),mk-Result(predefinedId1("Character"),PART),13)

| 14 then Literal-signature(mk-Name("SO"),mk-Result(predefinedId1("Character"),PART),14)

| 15 then Literal-signature(mk-Name("SI"),mk-Result(predefinedId1("Character"),PART),15)

| 16 then Literal-signature(mk-Name("DLE"),mk-Result(predefinedId1("Character"),PART),16)

| 17 then Literal-signature(mk-Name("DC1"),mk-Result(predefinedId1("Character"),PART),17)

| 18 then Literal-signature(mk-Name("DC2"),mk-Result(predefinedId1("Character"),PART),18)

| 19 then Literal-signature(mk-Name("DC3"),mk-Result(predefinedId1("Character"),PART),19)

| 20 then Literal-signature(mk-Name("DC4"),mk-Result(predefinedId1("Character"),PART),20)

| 21 then Literal-signature(mk-Name("NAK"),mk-Result(predefinedId1("Character"),PART),21)

| 22 then Literal-signature(mk-Name("SYN"),mk-Result(predefinedId1("Character"),PART),22)

| 23 then Literal-signature(mk-Name("ETB"),mk-Result(predefinedId1("Character"),PART),23)

| 24 then Literal-signature(mk-Name("CAN"),mk-Result(predefinedId1("Character"),PART),24)

| 25 then Literal-signature(mk-Name("EM"),mk-Result(predefinedId1("Character"),PART),25)

| 26 then Literal-signature(mk-Name("SUB"),mk-Result(predefinedId1("Character"),PART),26)

| 27 then Literal-signature(mk-Name("ESC"),mk-Result(predefinedId1("Character"),PART),27)

| 28 then Literal-signature(mk-Name("IS4"),mk-Result(predefinedId1("Character"),PART),28)

| 29 then Literal-signature(mk-Name("IS3"),mk-Result(predefinedId1("Character"),PART),29)

| 30 then Literal-signature(mk-Name("IS2"),mk-Result(predefinedId1("Character"),PART),30)

| 31 then Literal-signature(mk-Name("IS1"),mk-Result(predefinedId1("Character"),PART),31)

| 127 then Literal-signature(mk-Name("DEL"),mk-Result(predefinedId1("Character"),PART),127)

//Each of the following names starts and ends with an APOSTROPHE

| 32 then Literal-signature(mk-Name("' '"),mk-Result(predefinedId1("Character"),PART),32)

| 33 then Literal-signature(mk-Name("'!'"),mk-Result(predefinedId1("Character"),PART),33)

| 34 then Literal-signature(mk-Name("'"'"),mk-Result(predefinedId1("Character"),PART),34)

// the name character for 34 is APOSTROPHE QUOTATION-MARK APOSTROPHE

// and is enclosed in a QUOTATION-MARK pair

| 35 then Literal-signature(mk-Name("'#'"),mk-Result(predefinedId1("Character"),PART),35)

| 36 then Literal-signature(mk-Name("'$'"),mk-Result(predefinedId1("Character"),PART),36)

| 37 then Literal-signature(mk-Name("'%'"),mk-Result(predefinedId1("Character"),PART),37)

| 38 then Literal-signature(mk-Name("'&'"),mk-Result(predefinedId1("Character"),PART),38)

| 39 then Literal-signature(mk-Name("''''"),mk-Result(predefinedId1("Character"),PART),39)

// the name character for 39 is four APOSTROPHE items, representing the apostrophe character

// and is enclosed in a QUOTATION-MARK pair

| 40 then Literal-signature(mk-Name("'('"),mk-Result(predefinedId1("Character"),PART),40)

| 41 then Literal-signature(mk-Name("')'"),mk-Result(predefinedId1("Character"),PART),41)

| 42 then Literal-signature(mk-Name("'\*'"),mk-Result(predefinedId1("Character"),PART),42)

| 43 then Literal-signature(mk-Name("'+'"),mk-Result(predefinedId1("Character"),PART),43)

| 44 then Literal-signature(mk-Name("','"),mk-Result(predefinedId1("Character"),PART),44)

| 45 then Literal-signature(mk-Name("'-'"),mk-Result(predefinedId1("Character"),PART),45)

| 46 then Literal-signature(mk-Name("'.'"),mk-Result(predefinedId1("Character"),PART),46)

| 47 then Literal-signature(mk-Name("'/'"),mk-Result(predefinedId1("Character"),PART),47)

| 48 then Literal-signature(mk-Name("'0'"),mk-Result(predefinedId1("Character"),PART),48)

| 49 then Literal-signature(mk-Name("''1'"),mk-Result(predefinedId1("Character"),PART),49)

| 50 then Literal-signature(mk-Name("'2'"),mk-Result(predefinedId1("Character"),PART),50)

| 51 then Literal-signature(mk-Name("'3'"),mk-Result(predefinedId1("Character"),PART),51)

| 52 then Literal-signature(mk-Name("''4'"),mk-Result(predefinedId1("Character"),PART),52)

| 53 then Literal-signature(mk-Name("'5'"),mk-Result(predefinedId1("Character"),PART),53)

| 54 then Literal-signature(mk-Name("'6'"),mk-Result(predefinedId1("Character"),PART),54)

| 55 then Literal-signature(mk-Name("'7'"),mk-Result(predefinedId1("Character"),PART),55)

| 56 then Literal-signature(mk-Name("'8'"),mk-Result(predefinedId1("Character"),PART),56)

| 57 then Literal-signature(mk-Name("'9'"),mk-Result(predefinedId1("Character"),PART),57)

| 58 then Literal-signature(mk-Name("':'"),mk-Result(predefinedId1("Character"),PART),58)

| 59 then Literal-signature(mk-Name("';'"),mk-Result(predefinedId1("Character"),PART),59)

| 60 then Literal-signature(mk-Name("'<'"),mk-Result(predefinedId1("Character"),PART),60)

| 61 then Literal-signature(mk-Name("'='"),mk-Result(predefinedId1("Character"),PART),61)

| 62 then Literal-signature(mk-Name("''>'"),mk-Result(predefinedId1("Character"),PART),62)

| 63 then Literal-signature(mk-Name("'?'"),mk-Result(predefinedId1("Character"),PART),63)

| 64 then Literal-signature(mk-Name("'@'"),mk-Result(predefinedId1("Character"),PART),64)

| 65 then Literal-signature(mk-Name("'A'"),mk-Result(predefinedId1("Character"),PART),65)

| 66 then Literal-signature(mk-Name("'B'"),mk-Result(predefinedId1("Character"),PART),66)

| 67 then Literal-signature(mk-Name("'C'"),mk-Result(predefinedId1("Character"),PART),67)

| 68 then Literal-signature(mk-Name("'D'"),mk-Result(predefinedId1("Character"),PART),68)

| 69 then Literal-signature(mk-Name("'E'"),mk-Result(predefinedId1("Character"),PART),69)

| 70 then Literal-signature(mk-Name("'F'"),mk-Result(predefinedId1("Character"),PART),70)

| 71 then Literal-signature(mk-Name("'G"),mk-Result(predefinedId1("Character"),PART),71)

| 72 then Literal-signature(mk-Name("''H'"),mk-Result(predefinedId1("Character"),PART),72)

| 73 then Literal-signature(mk-Name("'I'"),mk-Result(predefinedId1("Character"),PART),73)

| 74 then Literal-signature(mk-Name("'J'"),mk-Result(predefinedId1("Character"),PART),74)

| 75 then Literal-signature(mk-Name("'K'"),mk-Result(predefinedId1("Character"),PART),75)

| 76 then Literal-signature(mk-Name("'L'"),mk-Result(predefinedId1("Character"),PART),76)

| 77 then Literal-signature(mk-Name("'M'"),mk-Result(predefinedId1("Character"),PART),77)

| 78 then Literal-signature(mk-Name("'N'"),mk-Result(predefinedId1("Character"),PART),78)

| 79 then Literal-signature(mk-Name("'O'"),mk-Result(predefinedId1("Character"),PART),79)

| 80 then Literal-signature(mk-Name("'P'"),mk-Result(predefinedId1("Character"),PART),80)

| 81 then Literal-signature(mk-Name("'Q"),mk-Result(predefinedId1("Character"),PART),81)

| 82 then Literal-signature(mk-Name("''R'"),mk-Result(predefinedId1("Character"),PART),82)

| 83 then Literal-signature(mk-Name("'S'"),mk-Result(predefinedId1("Character"),PART),83)

| 84 then Literal-signature(mk-Name("'T'"),mk-Result(predefinedId1("Character"),PART),84)

| 85 then Literal-signature(mk-Name("'U'"),mk-Result(predefinedId1("Character"),PART),85)

| 86 then Literal-signature(mk-Name("'V'"),mk-Result(predefinedId1("Character"),PART),86)

| 87 then Literal-signature(mk-Name("'W'"),mk-Result(predefinedId1("Character"),PART),87)

| 88 then Literal-signature(mk-Name("'X'"),mk-Result(predefinedId1("Character"),PART),88)

| 89 then Literal-signature(mk-Name("'Y'"),mk-Result(predefinedId1("Character"),PART),89)

| 90 then Literal-signature(mk-Name("'Z'"),mk-Result(predefinedId1("Character"),PART),90)

| 91 then Literal-signature(mk-Name("'["),mk-Result(predefinedId1("Character"),PART),91)

| 92 then Literal-signature(mk-Name("''\'"),mk-Result(predefinedId1("Character"),PART),92)

| 93 then Literal-signature(mk-Name("']'"),mk-Result(predefinedId1("Character"),PART),93)

| 94 then Literal-signature(mk-Name("'^'"),mk-Result(predefinedId1("Character"),PART),94)

| 95 then Literal-signature(mk-Name("'\_'"),mk-Result(predefinedId1("Character"),PART),95)

| 96 then Literal-signature(mk-Name("'`'"),mk-Result(predefinedId1("Character"),PART),96)

// the name character for 96 is APOSTROPHE GRAVE-ACCENT APOSTROPHE

// and is enclosed in a QUOTATION-MARK pair

| 97 then Literal-signature(mk-Name("'a'"),mk-Result(predefinedId1("Character"),PART),97)

| 98 then Literal-signature(mk-Name("'b'"),mk-Result(predefinedId1("Character"),PART),98)

| 99 then Literal-signature(mk-Name("'c'"),mk-Result(predefinedId1("Character"),PART),99)

| 100 then Literal-signature(mk-Name("'d'"),mk-Result(predefinedId1("Character"),PART),100)

| 101 then Literal-signature(mk-Name("'e"),mk-Result(predefinedId1("Character"),PART),101)

| 102 then Literal-signature(mk-Name("''f'"),mk-Result(predefinedId1("Character"),PART),102)

| 103 then Literal-signature(mk-Name("'g'"),mk-Result(predefinedId1("Character"),PART),103)

| 104 then Literal-signature(mk-Name("'h'"),mk-Result(predefinedId1("Character"),PART),104)

| 105 then Literal-signature(mk-Name("'i'"),mk-Result(predefinedId1("Character"),PART),105)

| 106 then Literal-signature(mk-Name("'j'"),mk-Result(predefinedId1("Character"),PART),106)

| 107 then Literal-signature(mk-Name("'k'"),mk-Result(predefinedId1("Character"),PART),107)

| 108 then Literal-signature(mk-Name("'l'"),mk-Result(predefinedId1("Character"),PART),108)

| 109 then Literal-signature(mk-Name("'m'"),mk-Result(predefinedId1("Character"),PART),109)

| 110 then Literal-signature(mk-Name("'n'"),mk-Result(predefinedId1("Character"),PART),110)

| 111 then Literal-signature(mk-Name("'o"),mk-Result(predefinedId1("Character"),PART),111)

| 112 then Literal-signature(mk-Name("''p'"),mk-Result(predefinedId1("Character"),PART),112)

| 113 then Literal-signature(mk-Name("'q'"),mk-Result(predefinedId1("Character"),PART),113)

| 114 then Literal-signature(mk-Name("'r'"),mk-Result(predefinedId1("Character"),PART),114)

| 115 then Literal-signature(mk-Name("'s'"),mk-Result(predefinedId1("Character"),PART),115)

| 116 then Literal-signature(mk-Name("'t'"),mk-Result(predefinedId1("Character"),PART),116)

| 117 then Literal-signature(mk-Name("'u'"),mk-Result(predefinedId1("Character"),PART),117)

| 118 then Literal-signature(mk-Name("'v'"),mk-Result(predefinedId1("Character"),PART),118)

| 119 then Literal-signature(mk-Name("'w'"),mk-Result(predefinedId1("Character"),PART),119)

| 120 then Literal-signature(mk-Name("'x'"),mk-Result(predefinedId1("Character"),PART),120)

| 121 then Literal-signature(mk-Name("'y"),mk-Result(predefinedId1("Character"),PART),121)

| 122 then Literal-signature(mk-Name("''z'"),mk-Result(predefinedId1("Character"),PART),122)

| 123 then Literal-signature(mk-Name("'{'"),mk-Result(predefinedId1("Character"),PART),123)

| 124 then Literal-signature(mk-Name("'|'"),mk-Result(predefinedId1("Character"),PART),124)

| 125 then Literal-signature(mk-Name("'}'"),mk-Result(predefinedId1("Character"),PART),125)

| 126 then Literal-signature(mk-Name("'~'"),mk-Result(predefinedId1("Character"),PART),126)

endcase // integer value parameter of chr

else undefined //should not occur – more than one parameter for chr

endif

| """<""" then //

if values.length = 2

then

if values.head.s-*Nat* < values.last.s-*Nat*

then mk-Literal-signature(mk-Name("true"),mk-Result(predefinedId1("Boolean"),PART),0)

else mk-Literal-signature(mk-Name("false"),mk-Result(predefinedId1("Boolean"), PART),1)

endif

else undefined //should not occur – exactly two parameters for <

endif

| """<=""" then //

if values.length = 2

then

if values.head.s-*Nat* <= values.last.s-*Nat*

then mk-Literal-signature(mk-Name("true"),mk-Result(predefinedId1("Boolean"),PART),0)

else mk-Literal-signature(mk-Name("false"),mk-Result(predefinedId1("Boolean"), PART),1)

endif

else undefined //should not occur – exactly two parameters for <=

endif

| """>""" then //

if values.length = 2

then

if values.head.s-*Nat* > values.last.s-*Nat*

then mk-Literal-signature(mk-Name("true"),mk-Result(predefinedId1("Boolean"),PART),0)

else mk-Literal-signature(mk-Name("false"),mk-Result(predefinedId1("Boolean"), PART),1)

endif

else undefined //should not occur – exactly two parameters for >

endif

| """>=""" then //

if values.length = 2

then

if values.head.s-*Nat* >= values.last.s-*Nat*

then mk-Literal-signature(mk-Name("true"),mk-Result(predefinedId1("Boolean"),PART),0)

else mk-Literal-signature(mk-Name("false"),mk-Result(predefinedId1("Boolean"), PART),1)

endif

else undefined //should not occur – exactly two parameters for >=

endif

| "num" then // natural value of a Character

if values.length = 1

then

Literal-signature(natToIntToken({i : values.head.s-*Nat* = i.chr.s-*Nat* ∧ i > 0 ∧ i < 128}.take),

mk-Result(predefinedId1("Integer"),PART),

2 \* values.head.s-Nat)

else undefined //should not occur – more than one parameter for num

endif

otherwise //should not occur – not one of the operators for Character

endcase

computeCharstring1 returns (to computeConstant1) the Literal-signature for the value of applying the Charstring operation to the Charstring constant expression argument values.

computeCharstring1(op: Operation-identifier, values: Value1\*): Literal-signature =def

case op.s-Name in

| "''" then // emptystring - APOSTROPHE APOSTROPHE enclosed in a QUOTATION-MARK pair

if values.length = 0

then

mk-Literal-signature(mk-Name("''"),mk-Result(predefinedId1("Charstring"),PART),0)

else undefined // should not occur – some parameters given

endif

| "mkstring" ∨ "Make" then // make a string from a Character

if values.length = 1 ∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Character"

then

mk-Literal-signature(

mk-Name("'" // APOSTROPHE

+ values.head.s-Nat.chr // character that has this numerical value

+ "'" ) // APOSTROPHE, // same Token for Name as character for most characters

mk-Result(predefinedId1("Charstring"),PART),

values.head.s-Nat+1) // one more than the corresponding Character natural

else undefined // should not occur – wrong number of parameters, or parameter not a Character literal

endif

| """//""" then // concatentate two Charstrings

if values.length = 2

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Charstring"

∧ values.last.s-Result.s-Sort-reference-identifier.s-Name = "Charstring"

then

if values.head.s-Nat = 0 // empty first string

then values.last // result is second string

elseif values.last.s-Nat = 0 // empty second string

then values.head // result is first string

else

let revisedChStr = // concatenate characters for Names

substring(values.head.s-Name ,2,values.head.s-Name.length-1) // omit apostrophes

+ substring(values.last.s-Name,2,values.last.s-Name.length-1)) // omit apostrophes

in

mk-Literal-signature(

mk-Name("'" + revisedChStr + "'" ), // APOSTROPHE + revisedChStr + APOSTROPHE

mk-Result(predefinedId1("Charstring"),PART),

computeCharstringNat1(revisedChStr) )

endlet // revisedChStr

endif // neither string empty

else undefined // should not occur – wrong number of parameters, or parameters not Charstring

endif

| "length" then // length of a Charstring – derived from the name of the charstring

if values.length = 1 ∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Charstring"

then

mk-Literal-signature((values.head.s-Name.length-2).natToIntToken,

mk-Result(predefinedId1("Integer"),PART),

2\*(values.head.s-Name.length-2)) // nat value for integer literal

else undefined // should not occur – wrong number of parameters, or parameter not a Charstring

endif

| "first" then // first character of a Charstring

if values.length = 1

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Charstring"

∧ values.head.s-Nat ≠ 0 // not emptystring

then

mk-Literal-signature(

mk-Name("'" // APOSTROPHE

+ substring(values.head,2,1)

+ "'" ) // APOSTROPHE,

mk-Result(predefinedId1("Character"),PART),

substring(values.head.s-Name,2,1).num) // nat for character

else undefined // should not occur – wrong number of parameters, or not a Charstring, or emptystring

endif

| "last" then // last character of a Charstring

if values.length = 1

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Charstring"

∧ values.head.s-Nat ≠ 0 // not emptystring

then

mk-Literal-signature(

mk-Name("'" // APOSTROPHE

+ substring(values.head.s-Name,values.head.s-Name.length-1,1)

+ "'" ) // APOSTROPHE,

mk-Result(predefinedId1("Character"),PART),

substring(values.head.s-Name,values.head.s-Name.length-1,1).num) // nat for character

else undefined // should not occur – wrong number of parameters, or not a Charstring, or emptystring

endif

| "Extract" then // extract a character of a Charstring

if values.length = 2

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Charstring"

∧ values.head.s-Nat ≠ 0 // not emptystring

∧ values.last.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

∧ values.last.s-Nat ≠ 0 // not zero

∧ (values.last.s-Nat /2) \* 2 = values.last.s-Nat // is positive integer

∧ values.last.s-Nat /2 < values.head.s-Name.length -1 // in range of Charstring

then

mk-Literal-signature(

mk-Name("'" // APOSTROPHE

+ substring(values.head.s-Name,(values.last.s-Nat /2)+1,1).chr

+ "'" ) // APOSTROPHE,

mk-Result(predefinedId1("Character"),PART),

substring(values.head.s-Name, (values.last.s-Nat /2)+1,1)+1) // nat value for character literal

else undefined // should not occur

endif

| "Modify" then // change a character of a Charstring

if values.length = 3

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Charstring"

∧ values.head.s-Nat ≠ 0 // not emptystring

∧ values.tail.head.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

∧ values.tail.head.s-Nat ≠ 0 // not zero

∧ (values.tail.head.s-Nat /2) \* 2 = values.last.s-Nat // is positive integer

∧ values.tail.head.s-Nat /2 < values.head.s-Name.length -1 // in range of Charstring

∧ values.last.s-Result.s-Sort-reference-identifier.s-Name = "Character"

then

let revisedChStr =

< if i = values.tail.head.s-Nat /2

then values.last.s-Nat.chr // character to add to name

else values.head.s-Name[i+1]

endif:

i ∈ 1..(values.head.s-Name.length-2)

>

in

mk-Literal-signature(

mk-Name("'" + revisedChStr + "'" ), // APOSTROPHE + revisedChStr + APOSTROPHE

mk-Result(predefinedId1("Charstring"),PART),

computeCharstringNat1(revisedChStr) )

endlet // revisedChStr

else undefined // should not occur

endif

| "substring" then // substring of a Charstring

if values.length = 3

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Charstring"

∧ values.head.s-Nat ≠ 0 // not emptystring

∧ values.tail.head.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

∧ values.tail.head.s-Nat ≠ 0 // not zero

∧ (values.tail.head.s-Nat /2) \* 2 = values.last.s-Nat // is positive integer – starting element

∧ values.tail.head.s-Nat /2 < values.head.s-Name.length -1 // in range of Charstring

∧ values.last.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

∧ (values.last.s-Nat /2) \* 2 = values.last.s-Nat // is positive integer – length

∧ values.tail.head.s-Nat/2+values.last.s-Nat /2 < values.head.s-Name.length-2 // in range

then

let revisedChStr =

< values.head.s-Name[i+1] :

i ∈ values.tail.head.s-Nat/2..( values.tail.head.s-Nat/2+values.last.s-Nat/2) >

in

mk-Literal-signature(

mk-Name("'" + revisedChStr + "'" ), // APOSTROPHE + revisedChStr + APOSTROPHE

mk-Result(predefinedId1("Charstring"),PART),

computeCharstringNat1(revisedChStr) )

endlet // revisedChStr

else undefined // should not occur

endif

| "remove" then // remove substring from a Charstring

if values.length = 3

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Charstring"

∧ values.head.s-Nat ≠ 0 // not emptystring

∧ values.tail.head.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

∧ values.tail.head.s-Nat ≠ 0 // not zero

∧ (values.tail.head.s-Nat /2) \* 2 = values.tail.head.s-Nat // is positive integer – starting element

∧ values.tail.head.s-Nat /2 < values.head.s-Name.length -1 // in range of Charstring

∧ values.last.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

∧ (values.last.s-Nat /2) \* 2 = values.last.s-Nat // is positive integer - length

∧ values.tail.head.s-Nat/2+values.last.s-Nat/2 < values.head.s-Name.length-2 // in range

then

let revisedChStr =

< values.head.s-Name[i+1] :

i ∈ (if values.tail.head.s-Nat/2=1 then ∅ else 1..values.tail.head.s-Nat/2-1 endif ∪

if values.tail.head.s-Nat/2+values.last.s-Nat/2 > values.head.s-Name.length-2 then ∅

else values.tail.head.s-Nat/2+values.last.s-Nat/2..values.head.s-Name.length-1

endif ) >

in

mk-Literal-signature(

mk-Name("'" + revisedChStr + "'" ), // APOSTROPHE + revisedChStr + APOSTROPHE

mk-Result(predefinedId1("Charstring"),PART),

computeCharstringNat1(revisedChStr) )

endlet // revisedChStr

else undefined // should not occur

endif

otherwise //should not occur – not one of the operators for Charstring

endcase

computeCharstringNat1 returns (to computeCharstring1) the Literal-natural for the Literal-signature for the Name given as its Literal-name.

computeCharstringNat1(nm: Name): Nat =def

if nm = "''" then 0 // emptystring

else (nm.head.num+1)\*128^(nm.length-1) + computeCharstringNat1(nm.tail)

endif

computeInteger1 returns (to computeConstant1) the Literal-signature for the value of applying the Integer operation to the Integer constant expression argument values. There are no integer literals for negative integers in the concrete grammar: instead negative values are represented by the application of the unary minus operation. However, in the formal model *Literal-signature* items are defined for negative values with the integer *Name* preceded by a "-". The *Literal-natural* value for an integer for positive integers is twice the integer value. For a negative integer n the *Literal-natural* value   
is -2\*n‑1.

computeInteger1(op: Operation-identifier, values: Value1\*): Literal-signature =def

let isIntIntOp = (values.length = 2

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

∧ values.last.s-Result.s-Sort-reference-identifier.s-Name = "Integer")

in

let a =

if (values.head.s-Nat/2) \* 2 = values.head.s-Nat

then values.head.s-Nat/2

else –(values.head.s-Nat/2+1)

endif

in

let b =

if (values.last.s-Nat/2) \* 2 = values.last.s-Nat

then values.last.s-Nat/2

else –(values.last.s-Nat/2+1)

endif

in

let truelt=mk-Literal-signature(mk-Name("true"),mk-Result(predefinedId1("Boolean"),PART),0) in

let falslt=mk-Literal-signature(mk-Name("false"),mk-Result(predefinedId1("Boolean"), PART),1)in

case op.s-Name in

| ""-"" then // unary minus of an Integer, or subtraction of Integers

if values.length = 1

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

then // unary minus of an Integer

mk-Literal-signature(

if (values.head.s-Nat /2) \* 2 = values.head.s-Nat

then "-" + values.head.s-Name

else substring(values.head.s-Name,2, values.head.length-1)

endif ,

mk-Result(predefinedId1("Integer"),PART),

values.head.s-Nat + if (values.head.s-Nat /2) \* 2 = values.head.s-Nat then -1 else +1 endif

) // Literal-signature

elseif isIntIntOp then // subtraction of Integers

mk-Literal-signature(

if a - b < 0

then "-" + *natToIntToken*(-a + b)

else *natToIntToken*(a - b)

endif ,

mk-Result(predefinedId1("Integer"),PART),

if a - b < 0

then (-a + b) \* 2 -1

else (a - b) \* 2

endif

) // Literal-signature

else undefined // should not occur

endif

| ""+"" then // addition of Integers

if isIntIntOp then

mk-Literal-signature(

if a + b < 0

then "-" + *natToIntToken*(-a - b)

else *natToIntToken*(a + b)

endif ,

mk-Result(predefinedId1("Integer"),PART),

if a + b < 0

then (-a - b) \* 2 -1

else (a + b) \* 2

endif

) // Literal-signature

else undefined // should not occur

endif

| ""\*"" then // multiplication of Integers

if isIntIntOp then

mk-Literal-signature(

if a \* b < 0

then "-" + *natToIntToken*(-a \* b)

else *natToIntToken*(a \* b)

endif ,

mk-Result(predefinedId1("Integer"),PART),

if a \* b < 0

then (-a \* b) \* 2 -1

else a \* b \* 2

endif

) // Literal-signature

else undefined // should not occur

endif

| ""/"" then // division of Integers

if isIntIntOp ∧ values.last.s-Nat ≠ 0 then

mk-Literal-signature(

if a / b < 0

then "-" + *natToIntToken*(-a / b)

else *natToIntToken*(a / b)

endif ,

mk-Result(predefinedId1("Integer"),PART),

if a / b < 0

then (-a / b) \* 2 -1

else (a / b) \* 2

endif

) // Literal-signature

else undefined // should not occur

endif

| ""mod"" then // modulus of Integers

if isIntIntOp ∧ values.last.s-Nat ≠ 0 then

if a >= 0 ∧ b >= 0 then computeInteger1(""rem"", values) // that is – a rem b

elseif b < 0 then computeInteger1(""mod"", < a, computeInteger1(""-"", < b >)>)

elseif a < 0 ∧ b > 0 ∧ computeInteger1(""rem"", values).s-Nat =0

then mk-Literal-signature("0", mk-Result(predefinedId1("Integer"),PART), 0)

elseif a < 0 ∧ b > 0 ∧ computeInteger1(""rem"", values).s-Name.head = "-")

then computeInteger1(""+"", < b, computeInteger1(""rem"", values)>)

else undefined // should not occur

endif

else undefined // should not occur

endif

| ""rem"" then // remainder of Integer division

if isIntIntOp ∧ values.last.s-Nat ≠ 0

then

if a – (b \*( a / b)) < 0

mk-Literal-signature(

if a – (b \*( a / b)) < 0

then "-" + *natToIntToken*(-a + (b \*( a / b)))

else *natToIntToken*(-a – (b \*( a / b)))

endif ,

mk-Result(predefinedId1("Integer"),PART),

if a – (b \*( a / b)) < 0

then (-a + (b \*( a / b)) \* 2 -1

else (a / b) \* 2

endif

) // Literal-signature

else undefined // should not occur

endif

| ""<"" then // less than for Integers

if isIntIntOp then

if a < b then truelt else falselt endif

else undefined // should not occur

endif

| ""<="" then // less than or equal to for Integers

if isIntIntOp then

if a <= b then truelt else falselt endif

else undefined // should not occur

endif

| "">"" then // greater than for Integers

if isIntIntOp then

if a > b then truelt else falselt endif

else undefined // should not occur

endif

| "">="" then // greater than or equal to for Integers

if isIntIntOp then

if a >= b then truelt else falselt endif

else undefined // should not occur

endif

| "power" then // a to the power b

if isIntIntOp then

if b = 0 then 1

elseif a = 1 then mk-Literal-signature("1", mk-Result(predefinedId1("Integer"),PART), 2)

elseif a = -1 then

if ((b/2) \* 2 ≠ b)

then mk-Literal-signature("1", mk-Result(predefinedId1("Integer"),PART), 2)

else mk-Literal-signature("-1", mk-Result(predefinedId1("Integer"),PART), 2-1)

endif

elseif (a < -1 ∨ a > 1) ∧ b < 0

then mk-Literal-signature("0", mk-Result(predefinedId1("Integer"),PART), 0)

elseif a < 0 ∧ (b/2) \* 2 ≠ b // a negative and b odd

then computeInteger1(""-"",

< computeInteger1("power", < computeInteger1("-", < values.head >), values.last >)

>) // - power(-a,b)

else // b > 0

computeInteger1(""\*"",

< values.head,

computeInteger1("power",

values.head,

computeInteger1("-",

< values.last,

mk-Literal-signature("1",

mk-Result(predefinedId1("Integer"),PART),

2)

>)

>) // a\*power(a,b-1)

endif

else undefined // should not occur

endif

| "integer" then // integer operation – used to turn an integer into a subtype of integer

if values.length = 1

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

then

values.head

| "num" then // num operation – used to turn a subtype of integer into an integer

if values.length = 1

∧ values.head.s-Result.s-Sort-reference-identifier.s-Name = "Integer"

then

values.head

// the formalisation of the equivalent integer literal for bitstring and hex literals is left for further study

otherwise undefined // should not occur – not one of the operators for Integer

endcase

endlet // falselt

endlet // truelt

endlet // b

endlet // a

endlet // isIntIntOp

computeReal1 returns (to computeConstant1) the Literal-signature for the value of applying the Real operation to the constant expression argument values. Each Real is actually a rational number that can be written as a/b, and these are counted as in the following table, where the rational is given before the = sign and the number after the sign is the count. The shaded cells with an X, each denote a rational that has the same value as one already counted: for example, 8/6 and 4/3. The count is doubled for the literal natural value, so that if the count for positive real r is n its literal natural is the even number 2 \* n and the literal natural for –r is the odd number 2 \* n -1.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |
| Divisor/ Dividend | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 1/1 = 1 | 1/2 = 3 | 1/3 = 4 | 1/4 = 9 | 1/5 = 10 | 1/6 = 17 | 1/7 = 18 | 1/8 = 27 |
| 2 | 2/1 = 2 | 2/2 = X | 2/3 = 8 | 2/4 = X | 2/5 = 16 | 2/6 = X | 2/7 = 26 | 2/8 = X |
| 3 | 3/1 = 5 | 3/2 = 7 | 3/3 = X | 3/4 = 15 | 3/5 = 19 | 3/6 = X | 3/7 = 29 | 3/8 = 39 |
| 4 | 4/1 = 6 | 4/2 = X | 4/3 = 14 | 4/4 = X | 4/5 = 25 | 4/6 = X | 4/7 = 48 | 4/8 = X |
| 5 | 5/1 = 11 | 5/2 = 13 | 5/3 = 20 | 5/4 = 24 | 5/5 = X | 5/6 = 37 | 5/7 | 5/8 |
| 6 | 6/1 = 12 | 6/2 = X | 6/3 = X | 6/4 = X | 6/5 = 36 | 6/6 = X | 6/7 | 6/8 = X |
| 7 | 7/1 = 21 | 7/2 = 23 | 7/3 = 30 | 7/4 = 35 | 7/5 | 7/6 | 7/7 = X | 7/8 |
| 8 | 8/1 = 22 | 8/2 = X | 8/3 = 34 | 8/4 = X | 8/5 | 8/6 = X | 8/7 | 8/8 = X |

computeReal1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computeDuration1 returns (to computeConstant1) the Literal-signature for the value of applying the Duration operation to the constant expression argument values. Duration inherits from Real and uses the same method for deriving literal natural values.

computeDuration1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function

computeTime1 returns (to computeConstant1) the Literal-signature for the value of applying the Time operation to the constant expression argument values. Time inherits from Real and uses the same method for deriving literal natural values.

computeTime1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computeBit1 returns (to computeConstant1) the Literal-signature for the value of applying the Bit operation to the constant expression argument values.

computeBit1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computeBitstring1 returns (to computeConstant1) the Literal-signature for the value of applying the Bitstring operation to the constant expression argument values.

computeBitstring1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computeOctetstring1 returns (to computeConstant1) the Literal-signature for the value of applying the Octetstring operation to the constant expression argument values.

computeOctetstring1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computePid1 returns (to computeConstant1) the Literal-signature for the value of applying the Pid operation to the constant expression argument values.

computePid1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study needed to determine whether this function is needed and its formulation (if it is needed).

computeTeletexChar1 returns (to computeConstant1) the Literal-signature for the value of applying the TeletexChar operation to the constant expression argument values.

computeTeletexChar1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computeTeletexCharString1 returns (to computeConstant1) the Literal-signature for the value of applying the TeletexCharString operation to the constant expression argument values.

computeTeletexCharString1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computeVideotexChar1 returns (to computeConstant1) the Literal-signature for the value of applying the VideotexChar operation to the constant expression argument values.

computeVideotexChar1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computeVideotexCharString1 returns (to computeConstant1) the Literal-signature for the value of applying the VideotexCharString operation to the constant expression argument values.

computeVideotexCharString1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computeUniversalChar1 returns (to computeConstant1) the Literal-signature for the value of applying the UniversalChar operation to the constant expression argument values.

computeUniversalChar1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function.

computeUniversalCharString1 returns (to computeConstant1) the Literal-signature for the value of applying the UniversalCharString operation to the constant expression argument values.

computeUniversalCharString1(op: Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function

isSpecialLiteralOp1 determines if the operator is one of the implicit literal operators (num, "<", ">", "<=", ">=", first, last, succ, pred) of a data type created from a literals list constructor (that is, with a non-empty Literal-signature-set).

isSpecialLiteralOp1( op : Operation-identifier ): Boolean=def

// Further study required for the formulation of this function

computeLiteralOp1 returns (to computeConstant1) the Literal-signature for the value of applying the special literal operation (see isSpecialLiteralOp1) to the constant expression argument values.

computeLiteralOp1( op : Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function

isSpecialStructOp1 determines if the operator is one of the operators (Make, fnExtract, fnModify, fnPresent, undefined) of a data type created from a structure definition, which is distinguished from a choice definition by the presence of parameters to Make, and the absence of PresentExtract.

isSpecialStructOp1( op : Operation-identifier ): Boolean=def

// Further study required for the formulation of this function

computeStructOp1 returns (to computeConstant1) the Literal-signature for the value of applying the special struct operation (see isSpecialStructOp1) to the constant expression argument values.

computeStructOp1( op : Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function

isSpecialChoiceOp1 determines if the operator is one of the operators (Make, fn, fnExtract, fnModify, fnPresent, PresentExtract, undefined) of a data type created from a choice definition.

isSpecialChoiceOp1( op : Operation-identifier ): Boolean=def

// Further study required for the formulation of this function

computeChoiceOp1 returns (to computeConstant1) the Literal-signature for the value of applying the special choice operation (see isSpecialChoiceOp1) to the constant expression argument values.

computeChoiceOp1( op : Operation-identifier, values: Value1\*): Literal-signature =def

// Further study required for the formulation of this function

computeAgentPid1 returns (to computeConstant1) the Pid value identified by the Agent-instance list.

computeAgentPid1(ail: Agent-instance\*): Literal-signature =def

// Further study required for the formulation of this function

## F2.3 Transformation of SDL‑2010 shorthands

This clause details the transformation of the SDL‑2010 constructs, whose dynamic semantics are given after a transformation to the subset of SDL‑2010 for which *Abstract Grammar* exists. These shorthand notations are constructs for which a *Model* section exists.

The properties of a shorthand notation are derived from the way it is modelled in terms of (or transformed to) the primitive concepts. In order to ensure easy and unambiguous use of the shorthand notations, and to reduce side effects when several shorthand notations are combined, these concepts are transformed in a specified order as detailed in this clause.

The specified order of transformation means that in the result of the transformation of a shorthand notation of order *m*, another shorthand notation of order *n* may be used, provided *n*>*m*. The order of the transformation is given as a number inside the transformation arrow, e.g., =5=> for a transformation of order 5.

The transformations are described as a number of enumerated steps. One step may describe the transformation of several concepts and thus consist of a number of sub-steps, either because these concepts must be transformed as a group or because the transformation order between these concepts is not significant.

If entities are moved to different scopes during the subsequent transformation steps, the <qualifier>s in every <identifier> bound to such an entity are updated to reflect this change. In fact, this case should not happen in the new version of SDL‑2010.

The following enumeration details the transformation steps to be performed in order.

1) Lexical transformations:

a) <macro definition> items and <macro call> items (see clause 6.7 of [ITU‑T Z.102]) are identified lexically and <macro call> items are expanded;

b) <macro definition> items are removed (also in <package definition> items).

– These transformations are not described formally, i.e., no macros are considered in the formal semantics.

– This step also includes simple transformations (such as removal of separators) that just adapt the AS0.

2) operations are handled, public removed from <operation signature> and <literal signature> items:

– omitted parameter kind replaced by in. See clause 9.4 *Model* of [ITU‑T Z.103];

– the <operation signature> of each method is transformed to an <operation signature> of the equivalent operator. See clause 12.1.3 of [ITU‑T Z.104];

– <operation definition> items are transformed into procedures having anonymous names and having the result as <procedure result>. See clause 12.1.7 of [ITU‑T Z.101] and clause 12.1.7 of [ITU‑T Z.104].

– <inline data type definition> (see clause 12.1 of [ITU‑T Z.104]);

– <inline syntype definition> (see clause 12.1 of [ITU‑T Z.104]);

– remove public (see clause 12.1.9 of [ITU‑T Z.104]).

3) <task> items and <statements> lists are transformed as defined in clause 11.13.1 of [ITU‑T Z.103], clause 11.14 of [ITU‑T Z.102] and clause 11.14 of [ITU‑T Z.103].

4) Definition references are replaced by <referenced definition>s (see clause 7.3 of [ITU‑T Z.101]).

5) The graphs are normalized, name lists are expanded to one name per item, omitted items with defaults set to the default:

– non-terminating decisions and non-terminating transition options are transformed into terminating decisions and terminating transition options respectively (clause 5.7.12.5 [ITU‑T Z.106]);

– the actions and/or terminator statement following the decisions and transition options are moved to appear as <free action>s. Those generated <free action>s which have no label attached are given anonymous labels;

– action lists (including the terminator statement which follows) where the first action (if any, otherwise the following terminator statement) has a label attached, are replaced by a join to the label and the action list appears as a <free action>;

– a system specification that is a process or block is replaced by a system enclosing the process or block as in clause 7.1 of [ITU‑T Z.103];

– each <signal definition list> that has multiple <signal definition> items is replaced by multiple <signal definition list> items each containing one <signal definition> as described in clause 10.3 of [ITU‑T Z.101];

– each <signal definition> has a (usually anonymous) name defined for the associated data type as in clause 10.3 of [ITU‑T Z.104];

– each <variables of sort>/<exported variables of sort> that has multiple <name>/<exported variable> (respectively) items is replaced by multiple <variables of sort>/<exported variables of sort> (respectively) items each containing one <name>/<exported variable> (respectively), and each <variable definition> that has multiple <variables of sort>/<exported variables of sort> items is replaced by multiple <variable definition> items each containing one <variables of sort>/<exported variables of sort>, so that each replacement <variable definition> defines one variable;

– each <fields of sort> that has multiple <field of kind> items is replaced by multiple <fields of sort> items each containing one <field<name>, so that each <field> contains only one <field<name> (clause 12.1.6.2 of [ITU‑T Z.104]);

– each <synonym definition> that has more than one <synonym definition item> is replaced by multiple <synonym definition> items each containing one <internal synonym definition item>/<external synonym definition item>;

– the local variables of compound and loop statements are expanded so that each <variable definition statement> defines one variable (clause 11.14.1 of [ITU‑T Z.102]);

– multiple timer set/reset clauses in a <set>/<set statement>/<reset>/<reset statement> are replaced by multiple <set>/<set statement>/<reset>/<reset statement> items each containing one set/reset clause, and multiple timer definition items in a timer definition are replaced by multiple timer definitions (clause 11.15 of [ITU‑T Z.103]);

– formal variable parameters have omitted parameter kind items set to in (clause 9.5 of [ITU‑T Z.103]);

– procedure formal parameters are expanded so that each <formal variable parameters> defines the name of only one parameter (derived from clause 8.1.1.1 of [ITU‑T Z.101]);

– add unique implicit connector name for a compound statement without a <connector name>(clause 11.14.1 of [ITU‑T Z.102]);

– add unique implicit names for unnamed channels (clause 10.1 [ITU‑T Z.103]).

6) Package use is transformed:

– the package Predefined is included in the <sdl specification> as in clause 7.1 of [ITU‑T Z.101];

– if the same <package<identifier> occurs in multiple <package use clause> items associated with a definition, these <package use clause> items are replaced by a single <package use clause> for that <package<identifier> as in clause 7.2 *Model* of [ITU‑T Z.101].

7) Transformation of:

– generic system (see clause 13 of [ITU‑T Z.102]) and external data (an <external synonym definition item> – see clause 12.1.8.3 of [ITU‑T Z.104], an operation defined by an <external operation definition> – see clause 12.1.7 of [ITU‑T Z.104], a procedure defined by an <external procedure definition> – see clause 9.4 of [ITU‑T Z.103], or <informal text> in a transition option):

– identifiers in <simple expression> items contained in the <sdl specification> are bound to definitions. During this binding, only <data definition> items defined in the predefined package Predefined and <external synonym definition> items are considered (that is, all other <data definition> items are ignored);

– <simple expression> items are evaluated and <select definition> items <transition option> items are replaced by the selected items.

8) Transformation of:

– <synonym> to a <constant expression> (see clause 12.1 of [ITU‑T Z.104]);

– <basic sort> with a <range condition> (see clause 12.1 of [ITU‑T Z.104]);

– non-deterministic decision (see clause 11.13.5 of [ITU‑T Z.102]);

– operations involving <infix operation name>s and their operands transformed to the prefix form (see clause 12.2.1 of [ITU‑T Z.104]);

– for every <operation definition> which does not have a corresponding <operation signature>, construct an <operation signature>: see clause 12.1.7 of [ITU‑T Z.104];

– structure data type (<structure definition>: see clause 12.1.6.2 of [ITU‑T Z.101] and clause 12.1.6.2 of [ITU‑T Z.104]);

– state list (<state list>: see clause 11.2 of [ITU‑T Z.103]);

– more than one <stimulus> or asterisk in an <input list> (see clause 11.3 of [ITU‑T Z.103]);

– more than one <priority stimulus> or asterisk in an <priority input> (see clause 11.4 of [ITU‑T Z.103]);

– more than one <save item> or asterisk in a <save list> (see clause 11.7 of [ITU‑T Z.103]);

– more than one <state exit point> or asterisk in a <connect list> of a <connect part> (see clause 11.11.4 of [ITU‑T Z.103]);

– field primary (see clause 12.2.3 of [ITU‑T Z.101] and clause 12.2.3 of [ITU‑T Z.104]);

– composite primary (for structure or array values: or clause 12.2.3 of [ITU‑T Z.101]);

– the range constraint for <syntype definition>s (see clause 12.1.8.2 of [ITU‑T Z.101]);

– multiple signals and multiple destinations in <output body> (see clause 11.13.4 of [ITU‑T Z.103]);

– multiple timers in <set body> and <reset body> (see clause 11.15 of [ITU‑T Z.103]);

– channel to channel connections replacing them with gates on the (implicit) agent type (see clause 10.2 of [ITU‑T Z.103]);

– default duration value for timer set (see clause 11.15 of [ITU‑T Z.101]);

– initialization of variables of sorts with default initialization (see clause 12.3.1 of [ITU‑T Z.101], clause 12.3.3.2 of [ITU‑T Z.101] and clause 12.3.3.2 of [ITU‑T Z.104]);

– <stimulus> containing <indexed variable>s and <field variable>s are transformed (see clause 11.3 of [ITU‑T Z.103]);

– replacing signallist definitions by interface definitions (see clause 12.1.2 Model of [ITU‑T Z.103]);

– replacing interface gate definitions by gate definitions (see clause 8.1.4 Model of [ITU‑T Z.103]);

– replacing interface identifiers from interface definitions by a list of signal identifiers (see clause 10.4 Semantics of [ITU‑T Z.101]);

– indexed primary (see clause 12.2.3 of [ITU‑T Z.101] and clause 12.2.3 of [ITU‑T Z.104]);

– field variable (see clause 12.3.3.1 of [ITU‑T Z.101]);

– indexed variable (see clause 12.3.3.1 of [ITU‑T Z.101]);

– <return body> with <expression> (see clause 11.12.2.4 of [ITU‑T Z.101]) ;

– <procedure call body> that calls a procedure defined outside the agent type, transformed into a call of an implicitly defined local procedure (see clause 11.13.3 of [ITU‑T Z.101]);

– <value returning procedure call> containing a <procedure call body> so that an implicit procedure is added (see clause 11.13.3 of [ITU‑T Z.101]);

– implicit name for state timer with an expression (rather than a set clause);

– implicitly named parameterless timer definition for state timer with an expression;

– anonymous names for choice data types defined by gate definitions for use by <as gate> (see clause 8.1.4 of [ITU‑T Z.104];

– empty parameters of Make (see clause 12.1.6.2 of [ITU‑T Z.101]);

– <field number> of <field primary> to a <field name> (see clause 12.2.3 of [ITU‑T Z.104]);

– <as signal> of <field primary> to a <field name> (see clause 12.2.3 of [ITU‑T Z.104]);

– <syntype definition data type> (see clause 12.1.8.1 of [ITU‑T Z.104]).

9) Insertion of implicit channels as described in clause 10.1 of [ITU‑T Z.103].

10) Insertion of implicit signal lists as described in clauses 10.5 and 10.6 of [ITU‑T Z.102].

11) Handle context parameters, specialize data types and generate anonymous names

– expand list context parameters so that only one name in each list (clauses 8.3.5 to 8.3.9 and 8.3.13 of [ITU‑T Z.102];

– replacement of context parameters described in clause 8.1.2 of [ITU‑T Z.102] and clause 8.3 of [ITU‑T Z.102];

– specialize data types as described in clause 8.4 of [ITU‑T Z.102] augmented by clause 12.1.9 of [ITU‑T Z.104];

– allocate anonymous type name for type expressions as in clause 8.1.2 of [ITU‑T Z.102];

– allocate anonymous names for unnamed default start and unnamed return of composite state types for default entry and exits clause 11.11.2 of [ITU‑T Z.102].

12) Transformation of shorthand notations <agent definition>, <agent body> and <composite state definition>:

– each <agent definition> into an <agent type definition> and a <textual typebased agent definition> as described in clause 9 of [ITU‑T Z.103];

– each <agent body> of an <agent type definition> into a <composite state type definition> and a <textual typebased composite state> as described in clause 8.1.1.1 of [ITU‑T Z.103];

– each <composite state definition> into a <composite state type definition> and a <typebased composite state> as described in clauses 9, 9.5 and 11.2 of [ITU‑T Z.103];

– each <via path> in a resulting <composite state type definition> that is a <channel identifier> into a <gate identifier> as described 11.3 of [ITU‑T Z.103].

13) Derivation of signal list for any channel path with omitted signal list:

– If an associated <signal list> is omitted from a <channel definition>, the <signal list> is replaced by a <signal list> derived from the channel connection (see clause 10.1 of [ITU‑T Z.103]).

14) Full qualifiers are inserted:

– According to the visibility rules and the rules for resolution by context (see clause 6.6 of [ITU‑T Z.101]), qualifiers are extended to denote the full path. This binding takes into account specialization such as: the replacement of an <anchored sort> by the <basic sort> of the **parent** base type or **this** local subsort (see clause 12.1 and 12.1.9 of [ITU‑T Z.104]).

15) Transformation of asterisk state:

– A body originating from an agent definition or procedure definition has its asterisk states expanded according to the model defined in clause 11.2 of [ITU‑T Z.103].

– Multiple appearance of state is merged (see clause 11.2 of [ITU‑T Z.103]).

16) Transformation of global block variables:

– A block with a global variable definition (a <variable definition> as an <entity in agent> in the <agent structure> of the block/system type of the block/system) has a state machine that is interpreted concurrently with agents in the block/system. Access from contained agents in the block/system to a global variable of the block is covered by two implicitly defined remote procedures for setting and getting the data item associated with the variable. See clause 9.2 of [ITU-T Z.102].

17) Implicit declarations for remote procedures and remote variables (see clauses 10.5 and 10.6 of [ITU‑T Z.102]) are generated. Imported and exported values (see clause 10.6 of [ITU‑T Z.102])) are transformed. Then remote procedures (see clause 10.5 of [ITU‑T Z.102])) are transformed.

NOTE – Text in [ITU‑T Z.101], [ITU‑T Z.102], [ITU‑T Z.103] and [ITU‑T Z.104] referenced above describes SDL‑GR, therefore needs modifying for the equivalent SDL-PR and AS0 (for example: <agent diagram> becomes <agent definition>).

Unused items (bookmarked text)

FunctionName: defaultValue – currently not used. Should it be deleted?

FunctionName: isConstantExpression0 – replaced by a static condition in F2.2.9.9.1

FunctionName: isSameNode1 – not used – imported from F1

FunctionName: parentSortId0 – currently not used. Should it be deleted?

|  |  |
| --- | --- |
| **SERIES OF ITU-T RECOMMENDATIONS** | |
| Series A | Organization of the work of ITU-T |
| Series D | Tariff and accounting principles and international telecommunication/ICT economic and policy issues |
| Series E | Overall network operation, telephone service, service operation and human factors |
| Series F | Non-telephone telecommunication services |
| Series G | Transmission systems and media, digital systems and networks |
| Series H | Audiovisual and multimedia systems |
| Series I | Integrated services digital network |
| Series J | Cable networks and transmission of television, sound programme and other multimedia signals |
| Series K | Protection against interference |
| Series L | Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant |
| Series M | Telecommunication management, including TMN and network maintenance |
| Series N | Maintenance: international sound programme and television transmission circuits |
| Series O | Specifications of measuring equipment |
| Series P | Telephone transmission quality, telephone installations, local line networks |
| Series Q | Switching and signalling, and associated measurements and tests |
| Series R | Telegraph transmission |
| Series S | Telegraph services terminal equipment |
| Series T | Terminals for telematic services |
| Series U | Telegraph switching |
| Series V | Data communication over the telephone network |
| Series X | Data networks, open system communications and security |
| Series Y | Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities |
| **Series Z** | **Languages and general software aspects for telecommunication systems** |
|  |  |

1. \* To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, <http://handle.itu.int/11.1002/1000/11830-en>. [↑](#footnote-ref-2)