International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.104

(10/2019)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

Specification and Description Language – Data and action language in SDL-2010

Recommendation ITU-T Z.104

International Telecommunication Union

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

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

Recommendation ITU-T Z.104

Specification and Description Language – Data and action language in SDL-2010

Summary

Recommendation ITU-T Z.104 defines the data features of the Specification and Description Language so that data definitions and expressions are well defined. Together with Recommendations ITU-T Z.100, ITU-T Z.101, ITU-T Z.102, ITU-T Z.103, ITU-T Z.105, ITU-T Z.106 and ITU-T Z.107, this Recommendation is part of a reference manual for the language. The language defined in this Recommendation partially overlaps features of the language included in Basic SDL-2010 in Recommendation ITU-T Z.101 and used in Comprehensive SDL-2010 in Recommendation ITU-T Z.102 and the features of Recommendation ITU-T Z.103.

Coverage

The Specification and Description Language has concepts for behaviour, data description and (particularly for larger systems) structuring. The basis of behaviour description is extended finite state machines communicating by messages. Data description is based on data types for values. The basis for structuring is hierarchical decomposition and type hierarchies. Though a distinctive feature of the Specification and Description Language is the graphical representation, the data and expression language is textual. This Recommendation covers the features of the language used to encode and decode data communicated by channels, define data types with values and operations and variables (including parameters) based on data types and expression actions that use the data types. This Recommendation does not always provide a canonical syntax, but by applying the *Model* descriptions given, a specification is transformed to Basic SDL-2010 defined in ITU-T Z.101 except in those cases where additional abstract syntax is added in this Recommendation. Object-oriented data in ITU-T Z.107 is an extension to this Recommendation.

Applications

The Specification and Description Language is applicable within standards bodies and industry. The main application areas for which the Specification and Description Language has been designed are stated in Recommendation ITU-T Z.100, but the language is generally suitable for describing reactive systems. The range of applications is from requirement description to implementation. The features of the language defined in Recommendation ITU-T Z.104 are essential for the data within a system.

History

Edition	Recommendation	Approval	Study Group	Unique ID*
1.0	ITU-T Z.104	2004-10-07	17	11.1002/1000/2203
2.0	ITU-T Z.104	2011-12-22	17	11.1002/1000/11391
2.1	ITU-T Z.104 (2011) Amd. 1	2012-10-14	17	11.1002/1000/11758
3.0	ITU-T Z.104	2016-04-29	17	11.1002/1000/12858
4.0	ITU-T Z.104	2019-10-14	17	11.1002/1000/14055

Keywords

Abstract data types, action language, C programming language binding, data, data types, encoding rules, predefined package, SDL-2010, Specification and Description Language.

* 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.

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

Table of Contents

Introduction

Status/Stability

This Recommendation is part of the ITU-T Z.100 to ITU-T Z.107 series of Recommendations that give the complete language reference manual for SDL-2010. The text of this Recommendation is stable. For more details see Recommendation ITU-T Z.100.

Recommendation ITU-T Z.104

Specification and Description Language – Data and action language in SDL-2010

1 Scope and objective

This Recommendation defines the data and action language features of the Specification and Description Language. The features defined in this document include: encoding and decoding of data communicated over channels, the definition of data types (including interfaces), the definition of variables (including parameters), the definition and application of operations, the evaluation of expressions and assignment of values to variables. Without these features the language would be of limited use. For this reason, even very simple specifications in the Specification and Description Language use ITU-T Z.104 features. Together with [ITU-T Z.100], [ITU-T Z.101], [ITU-T Z.102], [ITU-T Z.103], [ITU-T Z.105], [ITU-T Z.106] and [ITU-T Z.107], this Recommendation forms a reference manual for the language.

1.1 Objective

The objective of this Recommendation is to fully define the data related features of the Specification and Description Language (except [ITU-T Z.105] and [ITU-T Z.107] features). Some of the ITU-T Z.104 material is also in [ITU-T Z.100] and [ITU-T Z.101]. These definitions should be entirely consistent. The difference between the descriptions in [ITU-T Z.100] and [ITU-T Z.101] and the definition in Recommendation ITU-T Z.104 is that the former descriptions are overviews to enable data to be used with other features of SDL-2010, whereas the ITU-T Z.104 definition is intended to be complete (with the exception of definition of data types in ASN.1 modules and object-oriented – see [ITU-T Z.105] and [ITU-T Z.107] respectively).

Recommendation ITU-T Z.104 also includes a mechanism for binding data definitions and actions expressed in the concrete syntax from other notations as described in Annex C to the abstract grammar and semantics of SDL-2010 and data binding in clause 7.2. By default the concrete grammar is as given in the main body of ITU-T Z.104 or [ITU-T Z.101], [ITU-T Z.102], [ITU-T Z.103], [ITU-T Z.105], [ITU-T Z.106] and [ITU-T Z.107].

1.2 Application

This Recommendation is part of the reference manual for the Specification and Description Language.

The use of data is so fundamental, that even the simplest specifications need some data language features. Therefore, it is almost certain that some features defined in Recommendation ITU-T Z.104 will appear in any specification in SDL-2010.

2 References

The following ITU-T Recommendations and other references contain provisions which, through reference in this text, constitute provisions of this Recommendation. At the time of publication, the editions indicated were valid. All Recommendations and other references are subject to revision; users of this Recommendation are therefore encouraged to investigate the possibility of applying the most recent edition of the Recommendations and other references listed below. A list of the currently valid ITU-T Recommendations is regularly published. The reference to a document within this Recommendation does not give it, as a stand-alone document, the status of a Recommendation.

[ITU-T T.50] Recommendation ITU-T T.50 (1992), *International Reference Alphabet (IRA) (Formerly International Alphabet No. 5 or IA5) – Information technology – 7-bit coded character set for information interchange*.

[ITU-T X.680] Recommendation ITU-T X.680 (2015) | ISO/IEC 8824-1:2015, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*.

[ITU-T X.681] Recommendation ITU-T X.681 (2015) | ISO/IEC 8824-2:2015, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification*.

[ITU-T X.682] Recommendation ITU-T X.682 (2015) | ISO/IEC 8824-3:2015, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification*.

[ITU-T X.683] Recommendation ITU-T X.683 (2015) | ISO/IEC 8824-4:2015, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*.

[ITU-T X.690] Recommendation ITU-T X.690 (2015) | ISO/IEC 8825-1:2015, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*.

[ITU-T X.691] Recommendation ITU-T X.691 (2015) | ISO/IEC 8825-2:2015, *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*.

[ITU-T X.693] Recommendation ITU-T X.693 (2015) | ISO/IEC 8825-4:2015, *Information technology – ASN.1 encoding rules: XML Encoding Rules (XER)*.

[ITU-T X.696] Recommendation ITU-T X.696 (2015) | ISO/IEC 8825-7:2015, *Information technology – ASN.1 encoding rules: Specification of Octet Encoding Rules (OER)*.

[ITU-T Z.100] Recommendation ITU-T Z.100 (2019), *Specification and Description Language 2008 – Overview of SDL-2010*.

[ITU-T Z.101] Recommendation ITU-T Z.101 (2019), *Specification and Description Language – Basic SDL-2010*.

[ITU-T Z.102] Recommendation ITU-T Z.102 (2019), *Specification and Description Language – Comprehensive SDL-2010*.

[ITU-T Z.103] Recommendation ITU-T Z.103 (2019), *Specification and Description Language – Shorthand notation and annotation in SDL-2010*.

[ITU-T Z.105] Recommendation ITU-T Z.105 (2019), *Specification and Description Language – SDL-2010 combined with ASN.1 modules*.

[ITU-T Z.106] Recommendation ITU-T Z.106 (2019), *Specification and Description Language – Common interchange format for SDL-2010*.

[ITU-T Z.107] Recommendation ITU-T Z.107 (2019), *Specification and Description Language – Object-oriented data in SDL-2010*.

[ITU-T Z.111] Recommendation ITU-T Z.111 (2016), *Notations and guidelines for the definition of ITU-T languages*.

[ISO/IEC 10646] ISO/IEC 10646:2017, *Information technology – Universal Coded Character Set (UCS)*.

3 Definitions

3.1 Terms defined elsewhere

This Recommendation uses the following terms defined elsewhere:

The definitions of [ITU-T Z.100] apply.

3.2 Terms defined in this Recommendation

3.2.1 decode (process): The process to construct an SDL-2010 data value from a text string or bit-pattern that is assumed to be an encoding of the SDL-2010 data value using the same encoding rules as those used in the decode.

3.2.2 decoding: The result of a decode process.

3.2.3 encode (process): The process of producing an encoding.

3.2.4 encoding: The text string or bit-pattern resulting from the application of a set of encoding rules to an SDL-2010 data value.

3.2.5 set of encoding rules: One of the sets of (ASN.1) encoding rules (defined in [ITU-T X.690], [ITU-T X.691], [ITU-T X.693] and [ITU-T X.696]) or the set of text encoding rules defined in clause 10.7.1 and Annex B of this Recommendation, or an implementation-dependant or application-dependant set of encoding rules defined by a procedure invoked according to this Recommendation (see clause 10.7).

4 Abbreviations and acronyms

This Recommendation uses the following abbreviations and acronyms: the abbreviations defined in [ITU-T Z.100] apply. In addition the following abbreviation applies:

ASN.1 Abstract Syntax Notation One (as defined in [ITU-T X.680], [ITU-T X.681], [ITU-T X.682] and [ITU-T X.683]).

5 Conventions

The conventions defined in [ITU-T Z.100] apply, which includes the conventions defined in [ITU-T Z.111].

Where an abstract or concrete syntax rule is defined in this Recommendation with the same name as a rule in [ITU-T Z.101], [ITU-T Z.102] or [ITU-T Z.103], the rule given here replaces the rule in [ITU-T Z.101], [ITU-T Z.102] or [ITU-T Z.103]. Any *Abstract grammar* or *Concrete grammar* conditions, *Semantics* and *Model* defined on a named rule in [ITU-T Z.101], [ITU-T Z.102] or [ITU-T Z.103] apply to the redefined rule, unless specifically defined otherwise in this Recommendation. Any contradiction between [ITU-T Z.101] or [ITU-T Z.102] or [ITU-T Z.103] and this Recommendation is an error in the definition of SDL-2010 that needs to be resolved by further study.

Some numbered clauses are included so that this Recommendation has the general structure and numbering of [ITU-T Z.101], and in the text the feature is as described in [ITU-T Z.101]. Any extensions described in [ITU-T Z.102] or [ITU-T Z.103] are also allowed.

6 General rules

The general rules of [ITU-T Z.101], [ITU-T Z.102] and [ITU-T Z.103] apply.

However, <operation definition> is an alternative textual concrete syntax to <operation diagram> and therefore defines a scope unit.

7 Organization of Specification and Description Language specifications

7.1 Framework

The framework of specifications is as defined in [ITU-T Z.101].

7.2 Package

Concrete grammar

<package use clause> ::=
[<data binding>]
use <package identifier> [/ <definition selection list>] <end>

<data binding> ::=
default <package identifier>

The <package use clause> of Basic SDL-2010 is extended to include an optional <data binding>.

Each diagram that uses the native SDL-2010 defined concrete syntax for data and actions implicitly or explicitly includes **`default`** `Predefined` in its <package use clause>. If the language <data binding> implicitly or explicitly includes a package defined in Annex C for language binding, each rule identified in Annex C as a concrete syntax variation replaces the rule of the same name wherever it is used, and the rule represents the abstract grammar as defined in Annex C. Each diagram that uses a particular syntax linked to a package name by a language <data binding>, implicitly or explicitly includes a <data binding> for that package name in its <package use clause>. A referenced diagram that is logically included in another diagram implicitly includes the <data binding> of the enclosing diagram, unless a different <data binding> is given. Therefore it is only usually necessary to give the <data binding> for the package for the system type or for the <system specification>.

If the language <data binding> is omitted for the diagram and enclosing diagrams, the implicit language <data binding> is **`default`** `Predefined` if not otherwise specified. It is allowed to specify the implicit <data binding> by means such as analysis directives or tool features not defined by SDL-2010, but the concrete grammar including how it represents the SDL-2010 abstract grammar shall be one of the concrete grammars defined in Annex C.

Model

When an ASN.1 module is used as a package, and a <definition selection> in the <package use clause> referring to the ASN.1 has the <selected entity kind> **interface** and the <name> is the name of a CHOICE data type, there is an implied interface. This implied interface has the same name as the CHOICE and defines signals equivalent to the CHOICE alternatives.

7.3 Referenced definition

The concrete syntax is extended to include operations that are defined textually.

Concrete grammar

<definition> ::=
<macro definition>
| <procedure definition>
| <operation definition>

The <definition> of [ITU-T Z.103] is extended to include <operation definition>. Each <operation definition> shall have corresponding <operation reference> in the associated <package diagram> or <system specification>.

8 Structural concepts

The structural concepts are as defined in [ITU-T Z.101] with the addition of the optional identification of a set of encoding rules for gates and channels. Otherwise, the structural concepts are as defined in [ITU-T Z.101].

8.1 Types, instances and gates

This Recommendation adds the optional identification of a set of encoding rules to the definition of gates.

8.1.1 Structural type definitions

The structural type definitions are as defined in [ITU-T Z.101].

8.1.2 Type expressions

Type expressions are as defined in [ITU-T Z.101].

8.1.3 Abstract types

Abstract types are as defined in [ITU-T Z.102].

8.1.4 Gates with encoding rules and anonymous choice data type for a gate

A gate is allowed to have a set of encoding rules. An output of signal from an agent via the gate is encoded as specified by the set of encoding rules. Information received via the gate is decoded according to the set of encoding rules. If no specific set of encoding rules is given, the encoding is not defined by the SDL-2010 specification.

A gate definition defines an anonymous choice data type, which is not described in Basic SDL-2010.

Abstract grammar

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

Basic SDL-2010 is extended to allow *Gate-definition* to have *Encoding-rules*.

If an external channel with a set of *Encoding-rules* is connected to the gate of an agent or composite state, the *Encoding-rules* of the *Gate-definition* for the gate shall be the same as the *Encoding-rules* for the channel. There shall be, at most, one such channel connected to the gate and the signals conveyed in a direction for the gate shall be the same as the signals conveyed in the corresponding direction in the channel.

If an internal channel of an agent or agent type is connected to the gate of the agent or agent type that has a *Gate-definition* with a set of *Encoding-rules*, the *Encoding-rules* of the channel shall be the same. The signals conveyed in a direction for the gate shall be the same as the signals conveyed in the corresponding direction in the channel. More than one such internal channel is allowed.

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 grammar

<gate definition> ::=
{ { <gate symbol 1> | <inherited gate symbol 1> }
is associated with
{ <gate> [<encoding rules>] [<signal list area>] } ***set***
| { <gate symbol 2> | <inherited gate symbol 2> }
is associated with
{ <gate> [<encoding rules>] [<signal list area> <signal list area>] } ***set***
} [***is connected to*** <endpoint constraint>]

Shorthand SDL-2010 <gate definition> is extended to include <encoding rules>.

If there is no set of <encoding rules> that is associated with an <inherited gate symbol 1> or <inherited gate symbol 2> 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>.

NOTE – To keep the language grammar simple only one <encoding rules> is allowed for a bi-directional channel (<gate symbol 2> or <inherited gate symbol 2>); therefore, if a different encoding is required in each direction, the communication has to be specified by two gates that each carry signals in one direction and each connected to a different channel.

<interface gate definition> ::=
{ <gate symbol 1> | <inherited gate symbol 1> }
is associated with { <interface identifier> [<encoding rules>] }
[***is connected to*** <endpoint constraint>]

Shorthand SDL-2010 <interface gate definition> is extended to include <encoding rules>.

<as gate> ::=
as gate <gate identifier>

A <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.

Semantics

The set of *Encoding-rules* of a *Gate-definition* of an agent type or composite state type corresponds to the set of *Encoding-rules* of the channel in the enclosing scope in (the set of) instance specifications.

Model

The set of *Encoding-rules* of a *Gate-definition* of the implied agent type of an <agent diagram> (or the implied composite state type of a <composite state diagram>) is the same as the set of *Encoding-rules* of the external channel (explicit or implied) from which the *Gate-definition* is derived. There is no *Encoding-rules* item in this *Gate-definition* if there is none in the external channel.

The set of *Encoding-rules* of an implied *Gate-definition* of a system *Agent-type-definition* (whether defined by a <system type diagram> or implied from a <system diagram>) is the same as the set of *Encoding-rules* of the internal channel (explicit or implied) from which the *Gate-definition* is derived. There is no *Encoding-rules* item in this *Gate-definition* if there is none in the internal channel.

If an explicit <gate on diagram> is given for the <system type diagram>, the set of *Encoding-rules* of the corresponding *Gate-definition* of the *Agent-type-definition* is determined by the <encoding rules> of the <gate on diagram> if the <encoding rules> is present. If the <encoding rules> is absent, the set of *Encoding-rules* is determined from the internal channel in the same way as for an implied *Gate-definition*.

8.2 Type references and operation references

Type references are as defined in [ITU-T Z.101].

NOTE – A <gate property area> in a type reference is a <gate definition> or <interface gate definition> and therefore optionally contains an <encoding rules> specification, which has to be consistent with the definition given with the type.

Concrete grammar

<operation kind> ::=
{ **operator** | **method** }

Basic SDL-2010 <operation kind> is extended to allow an operation to be a **method**.

8.3 Context parameters

Context parameters (including gate context parameters) are as defined in [ITU-T Z.102].

NOTE – When a gate in parameterized type is defined by a formal context parameter, the set of encoding rules of the gate in a specialized type used to define instances will be determined by the actual gate definition identified by the actual context parameter.

8.4 Specialization

Specialization is as defined in [ITU-T Z.102] with the additional rules for shorthand notation in [ITU-T Z.103].

9 Agents

Agents are as defined in [ITU-T Z.101] or as in [ITU-T Z.102] or [ITU-T Z.103], if they are being used.

10 Communication

Encoding of data extends the definition of channels and connections.

10.1 Channel

A channel connecting two agents determines the encoding (if any) to be used for communication between the agents. A channel that is connected to the environment of an agent has the encoding defined (if any) for that gate connecting it with the environment.

A channel definition defines an anonymous choice data type, which is not described in Basic SDL-2010.

Abstract grammar

Channel-definition :: *Channel-name*
[*Encoding-rules*]
[**NODELAY**]
*Channel-path-**set***

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*.

NOTE 1 – In the *Concrete grammar* it is allowed to omit the <encoding rules> if the channel is connected to gates with encoding, and the *Encoding-rules* derived from the gates are as described in the *Model* below.

Concrete grammar

<channel definition area> ::=
<channel symbol 1>
is associated with
{ [<channel name> [<encoding rules>]] [<signal list area>] } ***set***
is connected to {
{ <agent area> | <state machine area> | <gate on diagram>
| <external channel identifiers> }
{ <agent area> | <state machine area> | <gate on diagram>
| <external channel identifiers> } } ***set***
| <channel symbol 2>
is associated with
{ [<channel name> [<encoding rules>]] [<signal list area>] [<signal list area>] } ***set***
is connected to {
{ <agent area> | <state machine area> | <gate on diagram>
| <external channel identifiers> }
{ <agent area> | <state machine area> | <gate on diagram>
| <external channel identifiers> } } ***set***

Shorthand SDL-2010 <channel definition area> is extended to allow <encoding rules> to be specified for a channel.

NOTE 2 – To keep the language grammar simple only one <encoding rules> is allowed for a bi-directional channel (<channel symbol 2>; therefore if a different encoding is required in each direction, the communication has to be specified by two channels that each carry signals in one direction.

<as channel> ::=
as channel <channel identifier>

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>.

Semantics

The *Encoding-rules* of a *Channel-definition* connected to a set of instance specifications is used in the behaviour of the instances.

Model

If the <encoding rules> item is omitted and the <channel symbol 1> or <channel symbol 2> is connected to a <gate on diagram> with <encoding rules>, the *Channel-definition* has the same *Encoding-rules* as the *Gate-definition* of the <gate on diagram>. If the *Gate-definition* has no *Encoding-rules*, the *Channel-definition* has no *Encoding-rules*.

Implicit channels have no *Encoding-rules* if the gates they are connected to have no *Encoding-rules*; otherwise, the implicit channel has the same *Encoding-rules* as each of the gates.

NOTE 3 – If the channel is connected to the frame of the enclosing diagram and there is no <gate on diagram>, this represents a connection.

10.2 Connection

Model

The *Encoding-rules* of an implicit gate of a connection is the same as the connected external channel. A channel without an <encoding rules> (or an implicit channel) that is connected to a gate that has an *Encoding-rules* has the same *Encoding-rules*.

10.3 Signal

Signal is extended so that a signal definition introduces a structure data type (optionally with field names) and use of a structure data type to define the sorts of the signal.

Concrete grammar

<signal definition> ::=
<type preamble>
<signal name>
[<formal context parameters>]
[<virtuality constraint>]
[<specialization>]
[<sort list> | <named fields sort list> | **struct** <sort identifier>]

Comprehensive SDL-2010 <signal definition list> is extended to allow a <named fields sort list>.

The <sort identifier> of **struct** <sort identifier> of a <signal definition> shall identify the sort of a structure data type.

In a <signal definition>, a <sort identifier> (after **struct)** adds an *Aggregation-kind* and *Sort-reference-identifier* for each field of the structure as the *Aggregation-kind* and *Sort-reference-identifier* of a *Signal-parameter* to the end of the *Signal-parameter* list of the *Signal-definition*. The *Aggregation-kind* for a *Signal-parameter* corresponding to a field is the same as the *Result-aggregation* for the operator to obtain the value of that field. The *Sort-reference-identifier* for a *Signal-parameter* corresponding to a field is the same as the *Operation-result* for the operator to obtain the value of that field.

<named fields sort list> ::=
[<visibility>]
([<visibility>] <aggregation kind> <field name> <sort>
{ **,** [<visibility>] <aggregation kind> <field name> <sort>}* **)**

Each <aggregation kind> and <sort> in a <named fields sort list> of the <signal definition> adds the corresponding *Aggregation-kind* and *Sort-reference-identifier* of a *Signal-parameter* to the end of the *Signal-parameter* list of the *Signal-definition*.

<as signal> ::=
as signal <signal identifier>

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. If there is <visibility> before the bracketed list of <named fields sort list> items, this is equivalent to <visibility> before the keyword **struct** of a <structure definition>. If there is <visibility> before the <aggregation kind> and <field name> of a <named fields sort list> item, this is equivalent to <visibility> in the <fields of sort> of a <structure definition>. The meaning of <visibility> in <fields of sort> and <structure definition> is explained in clause 12.1.8.4.

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>.

10.4 Signal list area

Signal list area is as defined in [ITU-T Z.101] and as in [ITU-T Z.102] and [ITU-T Z.103] if those are being used.

10.5 Remote procedures

Remote procedures are as defined in [ITU-T Z.102].

10.6 Remote variables

Remote variables are as defined in [ITU-T Z.102].

10.7 Communication path encoding rules, encode and decode

The set of encoding rules specifies which set of encoding rules is used to encode and decode data conveyed by a particular channel or gate.

Abstract grammar

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*

The *Rules-identifier* shall be one of the literal identifiers of the data type `Encoding` (see below in *Semantics*). If the actual rule identified corresponds to an encoding defined in [ITU-T X.690], [ITU-T X.691], [ITU-T X.693] and [ITU-T X.696], the set of signals carried by an *Encoding-path* shall correspond to elements of an ASN.1 CHOICE type that is carried by the channel, or the channel shall define a choice type that is equivalent to an ASN.1 CHOICE type. The *Encoding-path* and ASN.1 CHOICE type correspond in one direction if each signal name corresponds to a CHOICE name and for each signal the (single) parameter of the signal is the same as the data type of the corresponding CHOICE.

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 (for example, `encodemyprotocol` where the additional `Encoding` literal is `myprotocol`). This procedure shall have one parameter of the implicit choice type for the relevant path for the invocation (see below in *Semantics*). 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 (for example, `decodemyprotocol` where the additional `Encoding` literal is `myprotocol`). 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 (see below in *Semantics*).

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-identifier*s 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, 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 grammar

<encoding rules> ::=
encode <rules identifier>

<rules identifier> ::=
<literal>

<encoding expression> ::=
encode { <signal identifier> [<actual parameters>] | <expression> }
[<encoding path>]

<encoding path> ::=
as { [**channel**] <channel identifier>
| [**gate**] <gate identifier>
| [**interface**] <interface identifier> **with** <rules identifier> }

<decoding expression> ::=
decode <expression> [<encoding path>]

The set of encoding rules of the path identified by the <channel identifier> or <gate identifier> (or in the case of an <interface identifier>, by the <rules identifier>) is used. A <channel identifier> in an <encoding path> is a shorthand that represents a (possibly anonymous) gate.

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 if a <signal identifier> is given) in the context of the <encoding expression>.

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.

The <interface identifier> of an <encoding path> represents the *Data-type-identifier* of an *Encoding-path*. The <rules identifier> of an <encoding path> represents the *Rules-identifier* of the implicit *Encoding-rules* for the *Encoding-path*.

The encode and decode procedures associated with a <rules identifier> shall be visible where the <rules identifier> is used.

Semantics

The *Encoding-rules* item determines the set of rules used to change the implementation-dependent encoding for internal data:

– either to a standardized implementation-independent encoding (for one of the data type `Encoding` literals `text` to `OER`, or `PER` as a synonym for `APER`),

– or to a defined implementation or application encoding (for a literal added to a specialization of the data type `Encoding`).

Encode is invoked whenever a signal is output via a path with a set of encoding rules specified and the corresponding encode procedure is called. When a signal is input from such a path, decode of the data into the internal encoding is invoked by calling the corresponding decode procedure. This encode and decode therefore has no impact on the semantics of SDL, as defined in [ITU-T Z.101], but requires specific encoding of signals for the specified paths and, therefore, enables different parts of the system to be implemented separately.

When an *Encoding-expression* or *Decoding-expression* is interpreted, the appropriate procedure is called.

Encode relative to the set of encoding rules for a path is invoked in an *Encoding-expression* to produce a `Charstring`, `Bitstring` or `Octetstring` depending on the context and the set of encoding rules used. The `Charstring`, `Bitstring` or `Octetstring` produced by encode is decoded by a *Decoding-expression* using the same set of encoding rules for the same path. The set of encoding rules of the encoding path identified by the *Gate-identifier* or *Rules-identifier* (after an interface *Data-type-identifier*) is used.

For an *Encoding-expression*, the data is encoded as if it were going to be output on that path. The result is a data type corresponding to the set of encoding rules for the path.

For a *Decoding-expression*, the data is decoded as if it had been received on the specified path. The result is an expression corresponding to the implicit data type for input from that channel in the decode context as defined below. If decoding fails, the InvalidReference exception is raised.

For a channel, gate or interface that conveys the signals `signal1`, `signal2` and `signal3`, a choice data type with a unique, anonymous name is defined that corresponds to the SDL-2010:

```
value type UniqueAnon /* representing the unique anononymous name */
{ choice
         signal1 as signal signal1;
         signal2 as signal signal2;
         signal3 NULL;
}
```

where

```
as signal signal1
```

represents the structure data type with a unique, anonymous name

```
         value type signal1UniqueAnon
         { struct
            Ua11 Sort11 optional;
            Ua12 Sort12 optional;
            Ua13 Sort13 optional;
            /* ... and so on for each parameter of signal1 */
          } ;
```

as a consequence of the signal definition

```
         signal signal1 (Sort11, Sort12, Sort13 /* ... and so on */);
```

and `Ua11`, `Ua12` and `Ua13` are unique, anonymous field names for the structure data type, and similarly

```
as signal signal2
```

represents the structure data type with a unique, anonymous name

```
         value type signal2UniqueAnon
         { struct
            Ua21 Sort21 optional;
            Ua22 Sort22 optional;
          } ;
```

as a consequence of the signal definition

```
         signal signal2 (Sort21, Sort22);
```

and `signal3` is defined by

```
         signal signal3;
```

The anonymous identifier of the choice data type associated with a path is denoted by the **as channel** <channel identifier> or **as gate** <gate identifier> for the path, so that a legal variable declaration is:

```
     dcl message as gate user_input;
```

where `user_input` is the name of a channel or gate with encoding and a valid assignment is:

```
     message := decode encoded_value as user_input;
```

Similarly, the anonymous identifier of the data type associated with an interface is denoted by the **as interface** <interface identifier>, so that a legal variable declaration is:

```
dcl if_message as interface user_interface;
```

where `user_interface` is the name of an interface and a valid assignment (assuming the information was encoded using PER) is:

```
if_message := decode encoded_value as user_interface with PER;
```

The enumerated data type for the standardized set of encoding rules is defined in the package `Predefined` (see clause 14).

Model

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>.

If <encoding path> is omitted from <encoding expression>, the unique path required by the condition in the *Concrete grammar* is inserted.

If <encoding path> is omitted from <decoding expression>, the unique path required by the condition in the *Concrete grammar* is inserted.

10.7.1 The set of text encoding rules

The set of text encoding rules is provided so that it is possible to convey information on communication paths by means of text strings between elements in the system, and between the system and the environment. The encoding of characters is not defined. Though the text strings are, in general, readable by humans, that is not the purpose of the text encoding rules.

Semantics

If the set of encoding rules is specified as `text` the result of an encoding is a `Charstring`.

The actual string is determined as follows:

– LEFT CURLY BRACKET and RIGHT CURLY BRACKET { } delimit the values of data types and show where the values start and stop, except where they occur within the encoding of a `Charstring` in which case they represent the actual <left curly bracket> or <right curly bracket> characters of [ITU-T Z.101];

– COMMA characters are used to delimit elements within a list (for example, in a **struct** encoding);

– Otherwise the actual string of characters is determined for each data type as defined in Annex B and illustrated as the 'Generated `Charstring`' in the examples in Annex B.

A complete signal is encoded as a list of values and is treated as a **choice** encoding of the data type for the path with encoding.

The full details of text encoding are given in Annex B.

10.7.2 The sets of encoding rules standardized in the ITU-T X.69x series

The use of one of the names `BER`, `CER`, `DER`, `PER`, `APER`, `UPER`, `CAPER`, `CUPER`, `BXER`, `CXER`, `EXER` or `OER` (corresponding to [ITU-T X.690], [ITU-T X.691], [ITU-T X.693] and [ITU-T X.696] encoding rules – see *Semantics* in clause 10.7 above and clause 14.17 below) shall only be specified if the signals carried by the path or included in the interface are defined by an ASN.1 CHOICE data type or are able to be expressed as an ASN.1 CHOICE data type. The values are encoded according to this data type treated as an ASN.1 ABSTRACT-SYNTAX.

NOTE – Whether it is possible to express a set of signals not defined by an ASN.1 CHOICE as an ASN.1 CHOICE is not currently further defined.

11 Behaviour

11.1 Start

Start is as defined in [ITU-T Z.101].

11.2 State

Semantics

If during the consideration of the signals on the input port, the signal being considered does not correspond to any of the signals valid for any of the states of the agent because it is not possible to decode the message received (as in clause 10.7) to a valid signal, the `InvalidReference` exception is raised.

11.3 Input

Input is extended to allow a choice value to be stored for the signal that was input, and is also extended so that it is possible to store signals received on paths with encoding in the encoded form.

If an <encoded input> is given for a path (a channel or gate), the messages received from that path are received and stored in the variable given in their encoded form. It is allowed to specify these signals in other inputs or saves of the same state only if a <via path> is specified. Although the model given below describes the signals as first being decoded and then encoded again, it is expected that real implementations will optimize this to copying the encoded value to the variable given in the <encoded input>.

Abstract grammar

Input-node :: [*Priority-name*]
Signal-identifier [*Gate-identifier*]
[*Provided-expression*]
[*In-choice*]
[*Variable-identifier*]*
Transition

The *Input-node* of Comprehensive SDL-2010 is extended to include an optional *In-choice* to allow the storing of the signal value.

In-choice :: *Variable-identifier*

The *Variable-identifier* of an *In-choice* shall have a choice data type that includes a field for the signal type of the *Signal-identifier*.

Concrete grammar

<input list> ::=
<stimulus> [<in choice>] [<priority clause>]
{ **,** <stimulus> [<in choice>] [<priority clause>] }*
| <asterisk input list> [<in choice>] [<priority clause>]
| <encoded input>

The <input list> syntax of Comprehensive SDL-2010 is extended to allow the storing the signal value according to an <in choice> and to allow an <encoded input>.

<in choice> ::=
in { <choice variable> | <signal expression> }

The <choice variable> of an <in choice> shall have a choice data type that includes fields where all the signals that are mentioned in the <stimulus> are covered: that is, for each signal (for example, with the identity `signal_id`) there shall be a field with the unique name denoted by an <as signal> for the signal (**`as signal`** `signal_id`) and the data type defined by <as signal> for the signal (**`as signal`** `signal_id`).

A <signal expression> of a <in choice> represents the implicit anonymous choice variable of the agent with the choice data type that is capable of holding any of the values of signals in the valid input signal set of the agent.

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).

<encoded input> ::=
encode <variable> [<encoding path>]

If the <encoding path> is omitted from <encoded input>, there shall be exactly one path with encoding leading to the context of the input that has a set of encoding rules that produces the sort (`Charstring`, `Octetstring` or `Bitstring`) of the <variable>. Otherwise the <encoding path> shall have a set of encoding rules that produces the sort (`Charstring`, `Octetstring` or `Bitstring`) of the <variable>.

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).

Semantics

If the signal specified in the input is received via a channel that has a set of encoding rules specified, the signal is decoded according to that set of encoding rules.

If the *Input-node* has an *In-choice*, the values conveyed by signal are assigned to the choice variable. The field of the choice variable corresponding to the signal (the choice field) receives the signal: that is, each value conveyed by the signal is assigned to the corresponding field of the choice field. If the signal conveys no values, there is an assignment of NULL to the field of the choice variable corresponding to the signal.

Model

When applying the model for <asterisk input list>, the <in choice> is added after each <stimulus>. When applying the model for an <interface identifier> as a <signal list item> of a <stimulus> the <in choice> is added after each <stimulus>.

If the <encoding path> is omitted from <encoded input>, the encoding path is the unique path for the <encoded input>.

An <input part> for an <encoded input> is transformed to list of <input part> items, one for each signal this is receivable with the <encoded input>. For each of the signals that are receivable from the gate or channel specified in an <encoded input>, there is an <input part> with an <input list> item for the signal, an <in choice> item for the implicit **signal** variable and a <via path> for the gate or channel. For each of the signals of an interface identified by an <interface identifier> of an <encoding path> of an <encoded input>, there is an <input part> with an <input list> item for the signal, an <in choice> item for a <signal expression> and an undefined <via path>. Each revised <input part> contains the same <enabling condition> as the original <input part> for the <encoded input>. The <transition> of each revised <input part> contains a task, where the <variable> given in the <encoded input> is assigned the value of an <encoding expression> for the value in the implicit **signal** variable and the <encoding path> of the <encoding expression>. This task is followed by the original transition for the <encoded input>.

NOTE – If the <encoding path> is specified by an <interface identifier> **with** <rules identifier>, because the <encoded input> is transformed into inputs for the signals of the identified interface, no other input or save of the state is allowed to mention the same signals.

11.4 Priority input

Priority input is extended to allow a choice value to be stored for the signal that was input.

Concrete grammar

<priority input list> ::=
<priority stimulus> {**,** <priority stimulus>}*
| <asterisk input list> [<in choice>] [<priority clause>]

The <priority input list> syntax of [ITU-T Z.102] is extended to allow the storing the signal value for an <asterisk input list> according to an <in choice>.

<priority stimulus> ::=
<stimulus> [<in choice>] [<priority clause>]

The <priority stimulus> syntax of [ITU-T Z.102] is extended to allow the storing the signal value according to an <in choice>.

11.5 Continuous signal

Continuous signal is as defined in [ITU-T Z.102].

11.6 Enabling condition

Semantics

If the signal specified is received via a channel that has a set of encoding rules specified, the signal is decoded according to that set of encoding rules.

11.7 Save

Save is as defined in [ITU-T Z.101], [ITU-T Z.102] and [ITU-T Z.103].

11.8 Implicit transition

Implicit transition is as defined in [ITU-T Z.103].

11.9 Spontaneous transition

Spontaneous transition is as defined in [ITU-T Z.102].

11.10 Label

Label is as defined in [ITU-T Z.101], [ITU-T Z.102] and [ITU-T Z.103].

11.11 State machine and composite state

State machine and composite state are as defined in [ITU-T Z.101], [ITU-T Z.102] and [ITU-T Z.103].

11.12 Transition

Transition is as defined in [ITU-T Z.101], [ITU-T Z.102] and [ITU-T Z.103].

11.13 Action

11.13.1 Task

Task is as defined in [ITU-T Z.101], [ITU-T Z.102] and [ITU-T Z.103].

11.13.2 Create

Create is as defined in [ITU-T Z.101] and [ITU-T Z.103].

11.13.3 Procedure call

Procedure call is as defined in [ITU-T Z.101] and [ITU-T Z.102].

11.13.4 Output

If an <expression output> is given for a path (a channel or gate), the expression is a choice value used to derive the signal to output. The choice sort corresponds to the set of signals for the path.

If an <encoded output> is given for a path (a channel or gate), the expression given is used as the signal to output. In the model given below the expression is decoded to check the signal being sent.

Abstract grammar

Output-node	::	{ *Signal-identifier Actual-parameters* | *Expression* | *Encoded-expression*} *Activation-delay* *Signal-priority* [*Signal-destination*] *Direct-via*
Encoded-expression	::	*Expression*

The Basic SDL-2010 *Output-node* is extended to allow *Expression* or *Encoded-expression* (instead of *Signal-identifier* followed by *Actual-parameters*).

If an *Output-node* has an *Expression* (instead of *Signal-identifier* followed by *Actual-parameters*) the sort of the *Expression* shall be the sort of a choice data type, where choice fields correspond to outgoing signals carried from the local agent by one of the gates of the agent. A choice field of the choice data type corresponds if its field name is the same as:

a) a signal name that unambiguously identifies an outgoing signal; or

b) the unique anonymous *Sort* name for a *Data-type-definition* for a structure data type, denoted by an <as signal> that identifies an outgoing signal;

and the sort of the choice field is a structure data type where the sort of each field of the structure is the same as the corresponding (by position) parameter of the identified outgoing signal.

If an *Output-node* has an *Encoded-expression* (instead of *Signal-identifier* followed by *Actual-parameters*) the sort of the *Expression* of the *Encoded-expression* shall be `Charstring`, `Octetstring` or `Bitstring`. The *Direct-via* shall have exactly one *Gate-identifier*, and this shall identify a gate that has *Encoding-rules* with the same sort (`Charstring`, `Octetstring` or `Bitstring`) as the *Expression* of the *Encoded-expression*.

Concrete grammar

```
<output body item> ::=
                    {       <signal identifier> [<actual parameters>]
                      |     <expression output>
                      |     <encoded output> }
                    [ <activation delay> ] [ <signal priority> ]
```

The Basic SDL-2010 <output body item> is extended to allow <expression output> or <encoded output> (instead of <signal identifier> followed by optional <actual parameters>).

```
<expression output> ::=
                        <expression>

<encoded output> ::=
                        encode <expression>
```

Semantics

A signal output via a path that has a set of encoding rules specified is encoded according to that set of encoding rules (regardless of whether the *Output-node* has a *Signal-identifier*, *Expression* or *Encoded-expression*). The pid of the originating agent, the availability Time (if greater than now), and any non-zero signal priority value are also conveyed by the signal instance, but are not part of the encoded information.

If the *Output-node* has an *Expression*, and the value of the choice is undefined (that is, `undefined(e)` is `true`, where `e` is the expression), no signal is output. Otherwise if the *Output-node* has an *Expression* with a defined choice value, the name of the field present in the choice determines the signal to be considered for output. The signal name with the *Signal-destination* and *Direct-via* determine the destination agent instance as defined for an *Output-node* with a *Signal-identifier* in Basic SDL-2010. If there is a process instance that contains both the sender and the receiver, the data items conveyed by the signal instance are the values of corresponding present fields of the structure data type of the choice field. Otherwise, the data items conveyed by the signal instance are newly created replicates of these values (which for PART aggregation is the same as the value). Each conveyed data item is equal to the corresponding actual parameter of the output. If the field of the structure data type of the choice field is absent, the corresponding signal data item is "undefined". The signal instance is delivered to the destination agent in the same way as for an *Output-node* with a *Signal-identifier* in Basic SDL-2010, or is discarded if there is no agent instance or the agent is unreachable. The *Activation-delay* and *Signal-priority* are handled in the same way as for an *Output-node* with a *Signal-identifier* in Basic SDL-2010.

If an *Output-node* has an *Encoded-expression*, the signal is determined by decoding the contents of the expression according to the *Encoding-rules* for the gate identified by the *Direct-via*. The signal is the signal with the same name as the choice present in the decoding. If the signal is not valid for the path, or decoding fails for any reason (for example, if the string of the expression does not validly correspond to one of the signals for the path), the `OutOfRange` exception is raised and no signal is sent. If a *Signal-destination* is given and a destination instance does not exist or is unreachable, the signal is discarded. Otherwise the signal is conveyed in the encoded form over the path. The *Activation-delay* and *Signal-priority* are handled in the same way as for an *Output-node* with a *Signal-identifier* in Basic SDL-2010.

NOTE – If the *Output-node* has an *Encoded-expression*, because the expression is already encoded as a string for the path, it is possible to send the string value with no further conversion. It is not therefore expected that an implementation would normally decode and re-encode an <encoded output>.

11.13.5 Decision

Decision is as defined in [ITU-T Z.101], [ITU-T Z.102] and [ITU-T Z.103].

11.14 Statement list

Statement list is as defined in [ITU-T Z.101], [ITU-T Z.102] and [ITU-T Z.103].

11.15 Timer

Timer is as defined in [ITU-T Z.101], [ITU-T Z.102] and [ITU-T Z.103].

12 Data

The concept of data in SDL-2010 is defined in this clause. This includes the data terminology, the concepts to define new data types and the predefined data.

Data in SDL-2010 is principally concerned with data types. A data type defines a set of elements or data items, referred to as sort, and a set of operations that are allowed to be applied to these data items. The sorts and operations define the properties of the data type. These properties are defined by data type definitions.

A data type consists of a set, which is the *sort* of the data type, and one or more *operations*. As an example, consider the predefined data type `Boolean`. The sort `Boolean` of the data type Boolean consists of the elements `true` and `false`. Among the operations of the data type `Boolean` are `"="` (equal), `"/="` (not equal), `"not"`, `"and"`, `"or"`, `"xor"`, and `"=>"` (implies). As a further example, consider the predefined data type Natural. It has the sort Natural consisting of the elements `0`, `1`, `2`,

etc., and the operations "=", "/=", "+", "-", "*", "/", "mod", "rem", "<", ">", "<=", ">=", and power.

The Specification and Description Language provides several predefined data types, which are familiar in both their behaviour and syntax. The predefined data types are described in clause 13.

Variables are objects that are associated with an element of a sort by assignment. When the variable is accessed, the associated data item is returned.

The elements of the sort of a data type are either *values*, or *pids*, which are references to agents. The sort of a data type is defined in the following ways:

a) Explicitly enumerating the elements of the sort.

b) Forming the Cartesian product of sorts S_1, S_2, … S_n; the sort is equal to the set that consists of all tuples formed by taking the first element from sort S_1, taking the second element from sort S_2, … , and finally, taking the last element from sort S_n.

c) For the sorts of pids, by defining an interface (see clause 12.1.2).

d) As one of several sorts that are predefined and form the basis of the predefined data types described in Annex D. The predefined sort Pid is described in clause 12.1.5.

The elements of a value sort are values. The elements of a pid sort are pids.

Operations are defined from and to elements of sorts. For instance, the application of the operation for summation ("+") from and to elements of the Integer sort is valid, whereas summation of elements of the Boolean sort is not.

Each data item belongs to exactly one sort. That is, sorts never have data items in common.

For most sorts there are literal forms to denote elements of the sort: for example, for Integers, "2" is used rather than "1 + 1". It is allowed that more than one literal denote the same data item; for example, 12 and 012 are used to denote the same Integer data item. It is also allowed that the same literal denotation is used for more than one sort; for example, 'A' is both a Character and a Character String of length one. Some sorts have no literal forms to denote the elements of the sort; for example, the sorts that are formed as the Cartesian product of other sorts. In that case, the elements of these sorts are denoted by operations that construct the data item from elements of the component sort(s).

An expression denotes a data item. If an expression does not contain a variable or an imperative expression, e.g., if it is a literal of a given sort, each occurrence of the expression will always denote the same data item. These "passive" expressions correspond to a functional use of the language.

An expression that contains variables or imperative expressions is interpreted as having (usually) different results during the interpretation of a Specification and Description Language system, depending on the data item associated with the variables. The active use of data includes assignment to variables, use of variables, and initialization of variables. The difference between active and passive expressions is that the result of a passive expression is independent of when it is interpreted, whereas an active expression has (usually) different results depending on the current values or pids associated with variables or the current system state.

12.1 Data definitions

Data definitions are used to define data types. The basic mechanisms to define data are data type definitions (see clause 12.1.1) and interfaces (see clause 12.1.2). Definition of additional operations is described in clause 12.1.4. The definition of the sort of the data type as well as operations implied for the sort are given by data type constructors (see 12.1.6). Clause 12.1.7 shows how to define the behaviour of the operations of a data type. Clause 12.1.8 details a variety of features related to data definitions including syntypes, synonyms and visibility restriction. Specialization (see

clause 12.1.9) allows the definition of a data type to be based on another data type, referred to as its supertype.

Abstract grammar

Value-data-type-definition :: *Sort*
[*Data-type-identifier*]
*Literal-signature-**set***
*Static-operation-signature-**set***
*Procedure-definition-**set***
*Data-type-definition-**set***
*Syntype-definition-**set***
*Variable-definition-**set***
[*Default-initialization*]
[*Abstract*]

Value-data-type-definition is extended. It includes a *Variable-definition* set for synonym definition <entity in data type> items of the <data type definition> represented by the *Value-data-type-definition*. See clause 12.1.8.3 Synonym definition.

Concrete grammar

<sort> ::=
<basic sort> [**(** <range condition> **)**]
| <anchored sort>
| <pid sort>
| <inline data type definition>
| <inline syntype definition>

An <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>.

<inline data type definition> ::=
value [<data type specialization>]
[[<comment body>] <left curly bracket> <data type definition body>
<right curly bracket>]

<inline syntype definition> ::=
syntype <basic sort>
[[<comment body>] <left curly bracket>
{ <default initialization> [[<end>] <constraint>] | <constraint> } <end>
<right curly bracket>]

<basic sort> ::=
<datatype type expression>
| <as signal>
| <as interface>
| <as channel>
| <as gate>
| <as signallist>
| <syntype>

The <basic sort> of Basic SDL-2010 is extended to include <as signal>, <as interface>, <as channel>, <as gate> and <as signallist>.

An <as signal> represents the sort of the structure data type introduced by the identified signal definition.

An <as interface> represents the sort of the choice data type introduced by the identified interface definition.

An <as channel> represents the sort of the choice data type introduced by the identified channel definition.

An <as gate> represents the sort of the choice data type introduced by the identified gate definition.

An <as signallist> represents the sort of the implicit read-only string variable that it is assumed is used for storing unconsumed signal instances that have arrived at the agent.

<anchored sort> ::=
    this [<basic sort>]
  | **parent** [<basic sort>]

An <anchored sort> with <basic sort> is only allowed within the definition of <basic sort>.

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 the <data type definition>.

The meaning of an <anchored sort> is given in *Model* of clause 12.1.9.

Model

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.

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>**)**.

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>.

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>.

12.1.1 Data type definition

Although SDL-2010 does not include generators for data types, parameterized data types in **package** Predefined of SDL-2010 replace the generators such as Array that were included in the **package** Predefined in SDL-92. The legacy data type definition below includes legacy generators syntax so that these parameterized data types are allowed to be used with SDL-92 syntax.

Concrete grammar

<entity in data type> ::=
    <data type definition>
  | <syntype definition>
  | <synonym definition>
  | <legacy data type definition>
  | <legacy syntype definition>

The alternatives <synonym definition>, <legacy syntype definition> and <legacy data type definition> extend <entity in data type> compared with Basic SDL-2010. The legacy forms are alternative syntax to represent a *Data-type-definition* or *Syntype-definition*. The alternative <synonym definition> is explained in clause 12.1.8.3.

<data type definition> ::=
{<package use clause>}*
<type preamble> <data type heading> [<data type specialization>]
{ <end>
| [<comment body>] <left curly bracket> <data type definition body>
<right curly bracket> }

The <data type specialization> is added to <data type definition> so that it is possible to define a data type inheriting and specializing another data type (see clause 12.1.9). If the <type preamble> is **virtual** or **redefined**, a virtual data type is defined.

<data type heading> ::=
value type <data type name>
[<formal context parameters>] [<virtuality constraint>]

The <data type heading> is extended compared with Basic SDL-2010 to include <formal context parameters>, to allow use of a data type with context parameters.

<legacy data type definition> ::=
newtype <sort name>
[<formal context parameters>]
[<data type specialization>
|<legacy generators>
|<structure definition>]
[<literal list>]
[<legacy operator signatures>]
{ <legacy operator definition>
| <legacy operator reference>
| <legacy external operator definition> }*
[<default initialization> [<end>]]
[**constants** <range condition>]
endnewtype [<sort name>]

<legacy generators> ::=
<sort identifier> (<legacy generator actual> { , <legacy generator actual> }*)

The <sort identifier> of <legacy generators> should identify one of the parameterized data types in **package** `Predefined`. The <legacy generator actual> should be an appropriate actual parameter for the parameterized data types.

<legacy generator actual> ::=
<sort>
| <literal signature>
| <operator name>
| <constant expression>

To be consistent with SDL-92, the <data type specialization> in a <legacy data type definition> should contain a <legacy data inheritance>.

To be consistent with SDL-92, the <structure definition> in a <legacy data type definition> should not contain <visibility>, **optional** or <field default initialization>.

To be consistent with SDL-92, the <literal list> in a <legacy data type definition> should not contain <visibility> or <named number>.

If a <legacy data type definition> definition contains a <range condition>, it represents the definition of syntype and an anonymous parent data type.

The definition of a legacy operator shall be defined either by the <legacy operator definition> or the operator referenced by a <legacy operator reference> or <legacy external operator definition>.

A <formal context parameter> of <formal context parameters> of a <data type heading> or <legacy data type definition> shall be either a <sort context parameter> or a <synonym context parameter list>.

Basic SDL-2010 is extended so that 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>. An <external operation definition> is an additional alternative to <operation reference> compared with Basic SDL-2010.

Semantics

A *Value-data-type-definition* describes the elements of the sort and operations induced by the way the sort is constructed, and operations each characterized by an operation signature and the corresponding *Procedure-definition*. The operations are the set of operations that are allowed for the elements of a sort (see clause 12.1.4). The data type identified by the *Data-type-identifier* of the data type definition is the inherited data type. That is, the defined data type is a specialization (see clause 12.1.3) of the identified data type.

12.1.2 Interface definition

An interface is used to define a set of signals, remote procedures and remote variables provided by or used on a gate of an agent. The defining context of entities defined in the interface is the scope unit of the interface, and the entities defined are visible where the interface is visible. An interface is also allowed to refer to signals, remote procedures, or remote variables defined outside the interface through the <interface use list>.

An interface is used in a signal list to denote that the signals, remote procedure calls and remote variables of the interface definition are included in the signal list.

An interface definition defines an anonymous choice data type, which is not described in Basic SDL-2010.

Concrete grammar

```
<interface definition> ::=
                        {<package use clause>}*
                        [<virtuality>] <interface heading>
                        [<interface specialization>]
                        {           <end>
                            |       [ <comment body> ]
                                    <left curly bracket>
                                        <entity in interface>*
                                    <right curly bracket>
                        }
                    |   <signal list definition>
```

Basic SDL-2010 is extended to allow <virtuality> to be specified for an <interface definition>, and <interface specialization> to allow the interface definition to be a specialization of another interface definition. Basic SDL-2010 is also extended with <end> for an empty set of entities defined in the interface, in which case there shall be an <interface specialization>; otherwise the interface has no identified signals, remote procedures or remote variables.

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.

```
<interface heading> ::=
                        interface <interface name>
                            [<formal context parameters>] [<virtuality constraint>]
```

The <formal context parameters> shall only contain parameters specified by a <signal context parameter list>, or a <remote procedure context parameter>, or a <remotevariable context parameter list> or a <sort context parameter>.

<entity in interface> ::=
<signal definition list>
| <interface use list>
| <interface variable definition>
| <interface procedure definition>

The Basic SDL-2010 <entity in interface> is extended to allow <interface variable definition> entities for remote variables and <interface procedure definition> entities for remote procedures.

<interface variable definition> ::=
dcl <remote variable name> { **,** <remote variable name>}* <sort> <end>

<interface procedure definition> ::=
procedure <remote procedure name> <procedure signature> <end>

NOTE – The <remote variable name> or <interface procedure definition> each has to be a <name>, whereas in SDL-2000 each is allowed to be a name or a number (an <integer name> or a <real name>).

The semantics of <virtuality> is defined in clause 8.4.2 of [ITU-T Z.102].

The content of an interface is the set of all signals, remote procedures and remote variables that are defined in an <entity in interface> of the interface or referenced in the <interface use list> or included in the interface by specialization (that is, inheritance or context parameterization).

<as interface> ::=
as interface <interface identifier>

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>.

Model

The inclusion of an <interface identifier> in a <signal list> means the content of the interface (that is, all signal identifiers, all remote procedure identifiers and remote variable identifiers forming part of the <interface definition>) are included in the <signal list>.

Internally connected gates of an agent (or agent type) are explicit or implicit gates of the agent (or agent type respectively) that are connected via implicit or explicit channels to the gates of either the state machine of the agent (or agent type respectively) or a contained agent. The interface defined by an agent or agent type contains in its <interface specialization> all interfaces given in the incoming signal list associated with internally connected gates. The interface contains in its <interface use list> all signals, remote variables and remote procedures given in the incoming signal list associated with internally connected gates. In addition, the interface for an agent type that inherits another agent type also contains in its <interface specialization> the implicit interface defined by the inherited agent type.

If the containing entity is an agent type that inherits another agent type, then the interface of the state machine of the agent type also contains in its interface specialization the implicit interface of the state machine of the inherited agent type.

12.1.3 Operation signature

Basic SDL-2010 is extended to include methods, legacy syntax and operation visibility. A method is an operation that is applied to a variable and is able (but does not have to) modify the value associated with a variable.

Concrete grammar

<operation signatures> ::=
[<operator list>] [<method list>]

The Basic SDL-2010 <operation signatures> is extended to allow a <method list>.

<legacy operator signatures> ::=
operators
<legacy operator signature> { <end> <legacy operator signature> }* [<end>]

A <legacy data type definition> has <legacy operator signatures> instead of <operation signatures> for the concrete syntax of operation signatures.

<legacy operator signature> ::=
<operator name> : <arguments> -> <sort>

A <legacy operator signature> represents an *Operation-signature.*

The <sort> of a <legacy operator signature> represents the *Result* of the *Operation-signature*.

<method list> ::=
methods <operation signature> { <end> <operation signature> }* <end>

<operation signature> ::=
<operation preamble>
<operation name>
[<arguments>] [<result>]

The Basic SDL-2010 <operation signature> is extended to include an <operation preamble> that defines the visibility scope of the operation. Omitting the <result> is allowed for a method, as explained below.

<operation preamble> ::=
[<visibility>]

The <operation preamble> of an <operation signature> in a <sort signature> of a <sort constraint> shall be empty.

<result> in <operation signature> shall be omitted only if the <operation signature> occurred in a <method list>.

Model

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 original argument list was empty), <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>.

NOTE – An empty <parameter kind> is allowed and has the same meaning as the <parameter kind> **in** (see Procedure in [ITU-T Z.103]).

12.1.4 Generic data type operations

These are as defined in Basic SDL-2010.

12.1.5 `Pid` and pid sorts

These are as defined in Basic SDL-2010, except specialization is allowed so the output compatibility check (b) in clause 12.1.5 of [ITU-T Z.101] applies.

12.1.6 Data type constructors

12.1.6.1 Literals

Concrete grammar

<literal list> ::=
[<visibility>] **literals** <literal signature> { **,** <literal signature> }* <end>

The Basic SDL-2010 <literal list> is extended to allow restricting visibility of the literals.

Semantics

The meaning of <visibility> in <literal list> is explained in clause 12.1.8.4.

12.1.6.2 Structure data types

The concrete syntax of structure data types is extended to include visibility constraints and a shorthand for several consecutive fields of the same sort.

Concrete grammar

<structure definition> ::=
[<visibility>] **struct** [<field list>] <end>

The Basic SDL-2010 <structure definition> is extended to allow restricting <visibility> of the structure data type.

<fields of sort> ::=
[<visibility>]
<field of kind>{ **,** <field of kind> }* <field sort>

The Basic SDL-2010 <fields of sort> is extended to allow restricting <visibility> of fields of the structure. The Basic SDL-2010 <fields of sort> is also extended to allow two or more adjacent fields of the same sort to be defined more concisely.

The meaning of <visibility> in <fields of sort> and <structure definition> is explained in clause 12.1.8.4.

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 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` as in [ITU-T Z.101] clause 12.1.6.2 *Concrete grammar*.

NOTE 1 – In the absence of <data type specialization> there is only one constructor `Make` operator to inherit, except in the case that more than one constructor `Make` operator is explicitly given.

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) field extract operators as in [ITU-T Z.101] clause 12.1.6.2 *Concrete grammar*.

d) field modify operators as in [ITU-T Z.101] clause 12.1.6.2 *Concrete grammar*.

e) field present operators as in [ITU-T Z.101] clause 12.1.6.2 *Concrete grammar*.

f) a generic operator named `Undefined` as in [ITU-T Z.101] clause 12.1.6.2 *Concrete grammar*.

The generic constructor 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.

Semantics

When no constructor is given for a `Make` of a structure data type `T` that inherits from a structure data type `S`, in a `Make` of `T` the result for each field inherited from `S` is the same as the result for that field from a `Make` of `S` with the same actual values for the inherited parameters. The remaining fields of the result a `Make` of `T`, each field is associated with the result of the corresponding parameter, or if no value is given for the field, the default initialization for that field, or "undefined" if there is no default initialization for the field.

NOTE 2 – When no constructor is given for a `Make` of a structure data type `T` that inherits from a structure data type `S`, and the `Make` of `S` is the generic operator named `Make` as in [ITU-T Z.101] clause 12.1.6.2 *Concrete grammar*, the `Make` of `T` is same as the generic operator named `Make` in [ITU-T Z.101] clause 12.1.6.2 *Concrete grammar*.

Model

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

12.1.6.3 Choice data types

The concrete syntax of choice data types is extended to include visibility constraints.

Concrete grammar

<choice definition> ::=
[<visibility>] **choice** [<choice list>] <end>

The Basic SDL-2010 <choice definition> is extended to allow restricting visibility of the choice data type.

<choice of sort> ::=
[<visibility>]
<aggregation kind> <field name>{ **,** <aggregation kind> <field name> }* <field sort>

The Basic SDL-2010 <choice of sort> is extended to allow restricting visibility of the choice field. The Basic SDL-2010 <choice of sort> is also extended to allow two or more adjacent fields of the same sort to be defined more concisely.

The meaning of <visibility> in <choice of sort> and <choice definition> is explained in clause 12.1.8.4.

The generic operations to create choice values are applied as operators.

The generic operations for choice values that have an initial **in/out** parameter (to modify fields, access fields, test for the presence of a particular field, test for which field is present in the choice value, and test for a choice value being undefined) are treated as methods.

Model

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>.

After transforming any <aggregation kind> <field name> pair list as above, a <choice definition> introducing a sort named `C` implies for each field `fn` the following in the <operator list> of `C`:

```
fn ( fs ) -> C;
```

After transforming any <aggregation kind> <field name> pair list as above, a <choice definition> introducing a sort named `C` implies for each field `fn` the following in the <method list> of `C`:

```
fnExtract -> fs;
fnModify ( fs ) -> C;
fnPresent -> <<package Predefined>>Boolean;
```

These are transformed according to the model for methods in clause 12.1.4 to:

```
fnExtract ( in/out C ) -> fs;
fnModify  ( in/out C, fs ) -> C;
fnPresent ( in/out C ) -> <<package Predefined>>Boolean;
```

12.1.7 Behaviour of operations

The syntax for the definition of operations is extended compared with Basic SDL-2010.

Concrete grammar

<operation definition item> ::=
<operation definition>
| <operation reference>
| <external operation definition>

The Basic SDL-2010 <operation definition item> is extended to allow operations to be defined textually and to allow operations to be defined external to the <sdl specification> possibly in another notation.

<operation definition> ::=
{<package use clause>}*
<operation heading>
[<end> <entity in operation>+]
[<comment body>] <left curly bracket>
<statements>
<right curly bracket>

<operation heading> ::=
<operation kind> <operation preamble> [<qualifier>] <operation name>
[<formal operation parameters>]
[<operation result>]

The Basic SDL-2010 <operation heading> is extended with <operation preamble> and the extension of <operation kind> (in clause 8.2) allows operations to be methods. 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.

NOTE 1 – <operation preamble> is placed after <operation kind> to avoid ambiguity with the optional <operation signatures> which also sometimes starts with an <operation preamble>. The initial keyword of <operation definitions> is never the same as the initial keyword of an <operation signature>.

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.

<formal operation parameters> ::=
(<formal variable parameters> {**,** <formal variable parameters> }* **)**
| [<end>] **fpar** <formal variable parameters> {**,** <formal variable parameters> }*

The Basic SDL-2010 <formal operation parameters> is extended to allow a bracketed list as an alternative.

<entity in operation> ::=
<data definition>
| <variable definition>
| <select definition>
| <macro definition>

<operation result> ::=
{ <result sign> | **returns** } <result aggregation> [<variable name>] <sort>

NOTE 2 – The <variable name> has to be a <name>, whereas in SDL-2000 it is allowed to be a name or a number (an <integer name> or a <real name>).

The Basic SDL-2010 <operation result> is extended to allow the use of <result sign> instead of **returns** and a <variable name> for the result. If the result is named, the derived procedure has a result named with this name.

<external operation definition> ::=
<operation kind> <operation signature> external <end>

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>.

An <external operation definition> is an operator or method whose behaviour body is not included in the <sdl specification> and is not necessarily defined in the SDL-2010 or any previous version of the Specification and Description Language. An <external operation definition> is treated as if written in SDL-2010, and the <operation signature> represents *Operation-signature* and *Procedure-definition* for the operation. How the abstract grammar for external operations is derived is not further defined by SDL-2010.

For each <operation signature>, at most one corresponding <operation definition> shall be given. No operation shall have both an <operation diagram> and an <operation definition>.

The <statement>s in <operation definition> shall contain neither an <imperative expression> nor an <identifier> defined outside the enclosing <operation definition> or <operation diagram>, respectively, except for <synonym identifier>s, <operation identifier>s, <literal identifier>s and <sort>s.

<operation diagram> ::=
{ <operation page> }+

The Basic SDL-2010 <operation diagram> is extended to allow more than one page. The same <qualifier> shall be present on all pages or be omitted in the <operation heading> on every page.

If the operation is unambiguously identified from the <operation heading> by <operation name> or <qualifier> and <operation name>, the <formal operation parameters> and <operation result> need to be included in the <operation heading> of at least one page. The <formal operation parameters> and <operation result> of the <operation heading> shall be the same on every page on which they are included. Similarly, the <package use area> needs to be included for at least one page, and shall be the same for every page for which it is included.

<operation text area> ::=
<text symbol> ***contains***
{ <data definition>
| <variable definition>
| <macro definition>
| <select definition> }*

The Basic SDL-2010 <operation text area> is extended to allow <macro definition> and <select definition>.

The <start area> in <operation diagram> shall not contain <virtuality>.

The restriction on an <operation body area> containing an <identifier> (in Basic SDL-2010), does not apply for any <synonym identifier>.

NOTE 3 – <synonym identifier> is not mentioned in [ITU-T Z.101] because synonym is not included in Basic SDL-2010.

Any <statement> in <operation definition> shall 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.

The following syntax provides compatibility with SDL-92 models: <legacy operator definition>, <legacy operator reference> and <legacy external operator definition>. These are used instead of <operation definitions> within a <legacy data type definition> (see clause 12.1.8).

<legacy operator definition> ::=
{<package use clause>}*
<operation heading> <end>
{ <entity in operation> }*
<start>
endoperator
[{<operation identifier> | <operation name> }] <end>

<start> is defined in [ITU-T Z.106].

The body of the transition for <start> in a <legacy operator definition> shall contain only items that are allowed in an operation definition.

<legacy operator reference> ::=
<operation heading> **referenced** <end>

<legacy external operator definition> ::=
operator <operation name> [<legacy procedure signature>] **external** <end>

An <operation heading> in a <legacy operator definition> or <legacy operator reference> shall use the keyword **operator**.

A <legacy operator definition> corresponds to an <operation definition>.

A <legacy operator reference> corresponds to an <operation reference>.

A <legacy external operator definition> corresponds to an <external operation definition>.

Semantics

An operator is a constructor for elements of the sort identified by the result. It returns a value, or an agent identity. A method acts on a variable (through the implicit in/out parameter of the procedure

for the method) and therefore is able to modify the variable it acts on. In addition, a method optionally produces a result.

An operator shall not modify items that are reachable by the actual parameters. An item is considered modified in an operator if there is a potential control flow inside the operator resulting in modification. Therefore an operator with an *Inout-parameter* (in the procedure for the operator) shall not assign a value to the parameter or call a method or procedure that modifies the parameter.

NOTE 4 – For an operator there is no practical difference between an *In-parameter* and an *Inout-parameter* (though they will be probably have different implementations); an *Out-parameter* has no practical use, and an operator returns only a single value (whereas for a procedure each *Inout-parameter* or *Out-parameter* returns a value so a procedure is able to return multiple values).

Model

If an <operation diagram> has more than one <operation page>, this is transformed to a single page that contains every <operation text area> and <operation body area> on the pages. This transformation takes place before the contents of the page are transformed.

For every <operation definition> or <operation diagram> which does not have a corresponding <operation signature>, an <operation signature> is constructed.

A <operation definition> is derived syntax for an <operation diagram> that is a single <operation page> having the same <operation heading> and the <package use area> of the <operation diagram> containing the <package use clause> of the <operation definition>. The <transition area> of the <procedure start area> of the <operation page> consists of a <task area> containing the <statements> of the <operation definition> followed by an unlabelled <return area>. The <entity in operation> items of the <operation definition> are inserted into an <operation text area> of the <operation diagram> and allow local items (such as variables) of the operation to be defined. This transformation takes place after handling any <select definition> or <macro definition> in the <operation definition>.

12.1.8 Additional data definition constructs

This clause covers further constructs for data.

12.1.8.1 Syntypes

Concrete grammar

<syntype definition> ::=
{<package use clause>}* { <syntype definition syntype> | <syntype definition data type> }

<syntype definition data type> ::=
<type preamble> <data type heading> [<data type specialization>]
{ [<comment body>] <left curly bracket> <data type definition body> <constraint> <end>
<right curly bracket>
}

The Basic SDL-2010 <syntype definition> is extended to allow definition in the form of a data type definition with a constraint. Such a <syntype definition> with a <syntype definition data type> and the keywords **value type** is derived syntax defined below. In addition there is a legacy form used in legacy data type definitions.

<legacy syntype definition> ::=
syntype
<syntype name> <equals sign> <parent sort identifier>
[<default initialization> [<end>]]
[**constants** <range condition>]
endsyntype [<syntype name>]

NOTE – See also <legacy data type definition> for syntype combined with a **newtype**.

Semantics

Specifying a syntype identifier is the same as specifying the parent sort identifier of the syntype, except for the cases listed for Basic SDL-2010 in [ITU-T Z.101] and following cases:

a) remote variable or remote procedure definition if one of the sorts for derivation of implicit signals is a syntype;

b) <any expression>, where the result will be within the range (see clause 12.3.4.6).

Model

A <syntype definition> with the keywords **value type** or **newtype** 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>.

12.1.8.2 Constraint

Constraint on data types is defined in Basic SDL-2010.

12.1.8.3 Synonym definition

A synonym allows the use of a name for a constant that represents one of the data items of a sort. This is achieved by storing the value in a read-only variable.

Concrete grammar

<synonym definition> ::=
synonym <synonym definition item> { **,** <synonym definition item> }*<end>

<synonym definition item> ::=
<internal synonym definition item>
| <external synonym definition item>

<internal synonym definition item> ::=
<synonym name> [<sort>] <equals sign> <constant expression>

<external synonym definition item> ::=
<synonym name> <predefined sort> <equals sign> **external**

NOTE 1 – The <synonym name> has to be a <name>, whereas in SDL-2000 it is allowed to be a name or a number (an <integer name> or a <real name>).

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*.

NOTE 2 – Writing to the variable is not possible, because <synonym> is not allowed where assignments could take place.

If 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>. If the <sort> in the <synonym definition> is omitted the *Sort-reference-identifier* of the *Variable-definition* is derived from the constant expression sort.

If a <sort> is specified, this determines the *Sort-reference-identifier* of the *Variable-definition.* If a <sort> is specified, the sort of the <constant expression> has to be compatible with this sort.

The *Variable-definition* has a **PART** *Aggregation-kind*.

The <constant expression> in the concrete syntax denotes a *Constant-expression* in the abstract syntax as defined in [ITU-T Z.101].

The <constant expression> shall not refer to the synonym defined by the <synonym definition>, either directly or indirectly (via another synonym).

An <external synonym definition item> defines a <synonym> whose result is not defined in a specification (see clause 13).

Semantics

The result that the synonym represents is determined by the context in which the synonym definition appears.

A synonym has a value, which is the value of the constant expression associated with the variable for the synonym definition.

A synonym has a sort, which is the sort of the variable for the synonym definition.

Model

A <synonym definition> that defines multiple synonyms is a shorthand for a sequence of <synonym definition> items, each defining one synonym.

12.1.8.4 Restricted visibility

Concrete grammar

<visibility> ::=
public | **protected** | **private**

A <visibility> keyword shall not precede a <literal list>, <structure definition>, or <choice definition> in a <data type definition> containing <data type specialization>. A <visibility> keyword shall not be used in an <operation signature> that redefines an inherited operation signature.

Semantics

<visibility> controls visibility of a literal name or operation name.

When a <literal list> is preceded by <visibility>, this <visibility> applies to all <literal signature>s. When a <structure definition> or <choice definition> is preceded by <visibility>, then this <visibility> applies to all implied <operation signatures>.

When a <fields of sort> or <choice of sort> is preceded by a <visibility>, this <visibility> applies to all implied <operation signatures>.

If a <literal signature> or <operation signature> contains the keyword **private** in <visibility>, then the *Operation-name* derived from this <operation signature> is only visible within the scope of the <data type definition> that contains the <operation signature>. When a <data type definition> containing such <operation signature> is specialized, the <operation name> in <operation signature> is implicitly renamed to an anonymous name. Every occurrence of this <operation name> within the <operation definitions> or <operation diagram>s corresponding to this <operation signature> is renamed to the same anonymous name, when the <operation signature> and the corresponding operation definition are inherited by specialization.

NOTE 1 – As a consequence, the operator or method defined by this <operation signature> is usable only in operation applications within the data type definition that originally defined this <operation signature>, but not in any subtype thereof.

If a <literal signature> or <operation signature> contains the keyword **protected** in <visibility>, then the *Operation-name* derived from this <operation signature> is only visible within the scope of the <data type definition> that contains the <operation signature>.

NOTE 2 – Because inherited operators and methods are copied into the body of the subtype, the operator or method defined by this <operation signature> is accessible within the scope of any <data type definition> that is a subtype of the <data type definition> that originally defined this <operation signature>.

NOTE 3 – If a <literal signature> or <operation signature> does not contain <visibility>, the *Operation-name* derived from this <operation signature> is visible everywhere where the <sort name> that is defined in the enclosing <data type definition> is visible.

Model

If a <literal signature> or <operation signature> contains the keyword **public** in <visibility>, this is derived syntax for a signature having no visibility protection.

12.1.9 Specialization of data types

Specialization allows the definition of a data type based on another (super) type.

Concrete grammar

<data type specialization> ::=
inherits <data type type expression> [<renaming> | <legacy data inheritance>] [**adding**]

<interface specialization> ::=
inherits <interface type expression> { **,** <interface type expression> }* [**adding**]

<renaming> ::=
(<rename list> **)**

<rename list> ::=
<rename pair> { **,** <rename pair> }*

<rename pair> ::=
<rename pair operation name>
| <rename pair literal name>

<rename pair operation name> ::=
<operation name> <equals sign> <base type operation name>

<rename pair literal name> ::=
<literal name> <equals sign> <base type literal name>

The *Data-type-identifier* in the *Data-type-definition* represented by the <data type definition> in which <data type specialization> (or <interface specialization>) is contained identifies the data type represented by the <data type type expression> in its <data type specialization> (see also clause 8.1.2).

An *Interface-definition* is allowed to contain a list of *Data-type-identifier* items. The interface denoted by an *Interface-definition* is a specialization of all the interfaces denoted by the *Data-type-identifier* list items.

The resulting content of a specialized interface definition (<interface specialization>) consists of the content of the supertypes with the content of the specialized definition. This implies that the set of signals, remote procedures and remote variables of the specialized interface definition is the union of those given in the specialized definition itself and those of the supertypes. The resulting set of definitions shall obey the rules for visibility and naming.

The <data type constructor> with the <data type specialization> shall not be a <choice definition>.

The <data type constructor> with the <data type specialization> shall 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>: if the <data type constructor> used in a (direct or indirect) supertype was a <literal list>, the <data type constructor> shall be a <literal list>; if the supertype <data type constructor> was a <structure definition>, the subtype <data type constructor> shall be a <structure definition>.

Renaming changes the name of inherited literals, operators and methods in the derived data type.

All <literal name>s and all <base type literal name>s in a <rename list> shall be distinct.

All <operation name>s and all <base type operation name>s in a <rename list> shall be distinct.

A <base type operation name> specified in a <rename list> shall be an operation with <operation name> defined in the data type definition defining the <base type> of <data type type expression>.

A subtype with <renaming> of a sort is not allowed as an actual sort parameter constrained to be **atleast** that sort.

<legacy data inheritance> ::=
[<legacy literal renaming>]
[[**operators**] { **all** | (<legacy inheritance list>) } [<end>]]

<legacy literal renaming> ::=
literals <rename pair> { , <rename pair> }* <end>

<legacy inheritance list> ::=
<legacy inherited operator> { , <legacy inherited operator> }*

<legacy inherited operator> ::=
<operation name> | <rename pair>

To be consistent with SDL-92, the <operation name> as <legacy inherited operator> should refer to an operation defined in the base type. Specifying **operators all** or a named operation without renaming has no influence on the inherited operations, which are determined according to the SDL-2010 rules.

Semantics

The sort defined by the specialization is a subsort of the sort defined by the base type. The sort defined by the base type is a supersort of the sort defined by the specialization.

Sort compatibility determines when a sort is allowed to be used in place of another sort, and when it is not allowed. This relation is used for assignments (see clause 12.3.3), for parameter passing (see clause 12.2.7 and clause 9.4 in [ITU-T Z.101]), for re-declaration of parameter types during inheritance (see clause 12.1.2), and for actual context parameters.

Let P and R be two sorts. P is sort compatible with R if and only if one of the following apply:

a) P and R are the same sort;

b) P is directly sort compatible with R (see below);

c) R is denoted by a <sort identifier>, and the <sort identifier> is defined by an interface (a pid sort) and, for some sort Q, P is sort compatible with Q, and Q is sort compatible with R.

NOTE 1 – Sort compatibility is transitive only for sorts defined by interfaces, not for sorts defined by value types.

Let Y and Z be different sorts. Y is directly sort compatible with Z if and only if any of the following applies:

a) Y is denoted by an <anchored sort> of the form **this** Z;

b) Y is denoted by a <pid sort> (see clause 12.1.2 and clause 12.1.2 of [ITU-T Z.101]) and Zis a supersort of Y.

Model

The model for specialization in clause 8.4 of [ITU-T Z.102] is used, augmented as follows.

A specialized data type is based on another (base) data type by using a <data type definition> in combination with a <data type specialization>. The sort defined by the specialization is disjoint from the sort defined by the base type.

If the sort defined by the base type has literals defined, the literal names are inherited as names for literals of the sort defined by the specialized type unless literal renaming has taken place for that literal. Literal renaming has taken place for a literal if the base type literal name appears as the second name in a <rename pair>, in which case the literal is renamed to the first name in that pair.

If the base type has operators or methods defined, the operation names are inherited as names for operators or methods of the sort being defined, unless the operator or method has been declared as private (see clause 12.1.8.4) 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>, 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 signature of the operator is considered only after applying these models.

NOTE 2 – An <anchored sort> of the form **this** T has the same meaning as a <basic sort> of the form T in a specialization of T.

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 3 – From the specialization semantics in clause 8.4 of [ITU-T Z.102], an operator of T is inherited if the signature of the specialized operator is not the same as the signature of the base type operator, which is only the case if the signature is not already present in the subtype (a generic operator or matching user defined operator) or renaming had taken place, or contains one <sort> that is: a <basic sort> of the form T, or an <anchored sort> of the form **this** T.

12.2 Use of data

The following clauses define how sorts, literals, operators, methods and synonyms are interpreted in expressions.

12.2.1 Expression and expressions as actual parameters

Abstract grammar

Constant-expression	::	*Literal*
	|	*Conditional-expression*
	|	*Equality-expression*
	|	*Operation-application*
	|	*Range-check-expression*
	|	*Agent-instance-pid-value*

The Basic SDL-2010 *Constant-expression* is extended to include *Agent-instance-pid-value* to allow initialization of variables to the pid values of agent instances.

Agent-instance-pid-value :: *Agent-instance**

Agent-instance :: *Agent-name Instance-number*

Instance-number = *Expression*

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*.

Active-expression = *Variable-access*
| *Conditional-expression*
| *Operation-application*
| *Equality-expression*
| *Imperative-expression*
| *Range-check-expression*
| *Value-returning-call-node*
| *Encoding-expression*
| *Decoding-expression*
| *Agent-instance-pid-value*

The Basic SDL-2010 *Active-expression* is extended to include *Encoding-expression* and *Decoding-expression* to allow the use of encoding rules in expressions, and is extended to include *Agent-instance-pid-value* to allow the pid value for an agent instance to be dynamically determined.

The *Instance-number* of an *Agent-instance* in the *Agent-instance-pid-value* of a *Active-expression* shall be a positive `Natural` *Constant-expression* less than or equal to the *Active-agents-expression* of the *Agent-definition* for the *Agent-name* in the same *Agent-instance.*

Concrete grammar

For simplicity of description, no distinction is made between the concrete syntax of *Constant-expression* and *Active-expression.*

<expression0> ::=
<operand>
| <create expression>
| <value returning procedure call>
| <decoding expression>
| <encoding expression>
| <agent instance pid value>

The Basic SDL-2010 <expression0> is extended to allow a <create expression>, <decoding expression>, <encoding expression> and <agent instance pid value>. An <expression0> that contains a <create expression>, <decoding expression> or <encoding expression> is not a <constant expression0> and represents an *Active-expression*. An <expression0> that contains a <agent instance pid value> is an *Agent-instance-pid-value* of an *Active-expression* if (and only if) any of the <Natural expression> items of an <agent instance> of the <agent instance pid value> represents an *Active-expression*. Otherwise an <expression0> that contains a <agent instance pid value> is an *Agent-instance-pid-value* of a *Constant-expression* and is evaluated only once when the system is initialized.

<agent instance pid value> ::=
system [<name>]
{ **value** < agent instance> **endvalue**
| <left curly bracket> <agent instance> <right curly bracket> }*

<agent instance> ::=
[**block** | **process**] <agent name>
[<left square bracket> <Natural expression> <right square bracket>]

If a <name> is given in an <agent instance pid value>, it shall be the name of the system. The keyword **system** in <agent instance pid value> represents the first item of the *Agent-instance* list of the *Agent-instance-pid-value* with the *Name* for the system and an *Instance-number* for the `Natural` number "1".

If an optional keyword **block** or **process** is given in an < agent instance> it shall match the kind of the agent (block or process).

If the square brackets and <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`.

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".

<primary> ::=
<operation application>
| <literal>
| **(** <expression> **)**
| <conditional expression>
| <extended primary>
| <active primary>
| <synonym>

The Basic SDL-2010 <primary> is extended to allow a <synonym>.

Semantics

An *Agent-instance-pid-value* is a pid value of the implicit pid sort for the type of the agent instance set and is compatible with the predefined `Pid` sort. If the *Agent-instance* list is empty, the *Agent-instance-pid-value* is the `Pid` value of the state machine of the system: that is signals sent to this `Pid` value are sent to the system state machine. Otherwise an *Agent-instance-pid-value* is the unique pid value for the agent instance given by the rightmost item of the *Agent-instance* list, so that signals output to this pid value are received by the agent instance. If the rightmost item of the *Agent-instance* list does not identify a valid agent instance, the *Agent-instance-pid-value* is the `Pid` value `Null`.

NOTE – An *Agent-instance-pid-value* that is a *Constant-expression* is interpreted once during initialization of the system, and the result of the interpretation is preserved. Whenever the value of the *Agent-instance-pid-value* is needed during interpretation, a replicate of the computed value is used. If the initial agent instance no longer exists signals sent to this instance are discarded.

12.2.2 Literal

Literal is as defined in Basic SDL-2010.

12.2.3 Extended primary

Extended variable for an indexed primary is as defined in Basic SDL-2010, but for a field primary it is extended.

Concrete grammar

<field primary> ::=
<primary> <exclamation mark> { <field name> | <field number> | <as signal> }
| <primary> <full stop> { <field name> | <field number> | <as signal> }
| <field name>

The <field primary> of Basic SDL-2010 is extended to allow the field to be identified by a <field number> or <as signal>, and to allow the short form <field name> (without a <primary>) in the context of the data type definition of a structure or choice data type.

<field number> ::=
<integer name>

The <field number> shall represent a non-zero, positive integer less than or equal to the number of fields in the sort of the variable.

Model

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.

NOTE 1 – A <field number> has to be used to denote a field of a structure data type with a unique anonymous name introduced by a <signal definition> with a <sort list>.

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>.

NOTE 2 – An <as signal> has to be used to denote a field of a choice data type introduced by a <gate definition>, <channel definition area> or <interface definition>.

When the <field primary> has the form <field name>, this is derived syntax for
this ! <field name>

12.2.4 Equality expression

Equality expression is as defined in Basic SDL-2010.

12.2.5 Conditional expression

Conditional expression is as defined in Basic SDL-2010.

12.2.6 Operation application

Operation application is extended to include the syntax for method application

Concrete grammar

<operation application> ::=
<operator application>
| <method application>

<method application> ::=
<primary> <full stop> <operation identifier> [<actual parameters>]

A <method application> is legal concrete syntax only if <operation identifier> represents a method.

When the operation application in an operation has the syntactical form:
<operation identifier> [<actual parameters>]

then, during derivation of the *Operation-identifier* from context, the form:

this <full stop> <operation identifier> [<actual parameters>]

is also considered if the <operation identifier> could identify a method. The model in clause 12.3.2 is applied before resolution by context is attempted.

An <expression> in <actual parameters> corresponding to an *Inout-parameter* or *Out-parameter* in the *Procedure-definition* associated with the *Operation-signature* shall not be omitted and shall be a <variable access> or <extended primary>.

NOTE – Omission of <actual parameters> is allowed in an <operation application> if all actual parameters have been omitted.

Model

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> followed by a comma before the first optional expression in <actual parameters>.

If the <primary> of a <method application> is not a variable or **this**, one of the transformations below is applied before replacing the <full stop> as above, and there is an implicit assignment of the <primary> to an implicit variable with the sort of the first parameter of the operation (that is, the method sort) and this variable is used instead of the <primary>.

If the <method application> is in a statement, the following transformation is applied. The <statement> in which the <method application> occurs is replaced by a <left curly bracket> followed by a <variable definition> for the implicit variable, followed by an <assignment statement> assigning the <primary> to the implicit variable, followed by the original <statement> where the implicit variable replaces the <primary> in the <method application>. This is then either a <compound statement> or a <loop compound statement> depending on context.

If the <method application> is in an action that is not in a statement (for example, in the <question> of a <decision area>), the implicit variable is at the enclosing context and <task area> is placed in the flow immediately before the action. The task area contains the <assignment statement> and the implicit variable replaces the <primary> in the <method application>.

12.2.7 Range check expression

Range check expression is as defined in Basic SDL-2010.

12.2.8 Synonym

Concrete grammar

<synonym> ::=

<synonym identifier>

A <synonym> represents the *Variable-identifier* for the identified synonym.

Semantics

The synonym gives the *Constant-expression* associated with the variable.

12.3 Active use of data

This clause extends the active use of data defined in Basic SDL-2010.

12.3.1 Variable definition

Variable definition is as defined in Basic SDL-2010 and (for exported) Comprehensive SDL-2010 with the extension to include remote variables defined in an interface definition.

Concrete grammar

The <remote variable identifier> of an <exported variable> is defined by a <remote variable definition> (as in Comprehensive SDL-2010), or in an <interface variable definition> of an explicit <interface definition>.

12.3.2 Variable access

Variable access is extended from Basic SDL-2010 and (for import expression) Comprehensive SDL-2010 to allow the use of **this**.

Concrete grammar

<variable access> ::=
<variable identifier>
| <import expression>
| **this**

this shall only occur in method definitions.

Model

The transformation for <import expression> is given in Comprehensive SDL-2010.

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.

12.3.3 Assignment

Assignment is as defined in Basic SDL-2010.

12.3.3.1 Extended variable

Extended variable for an indexed variable is as defined in Basic SDL-2010, but for a field variable it is extended to allow the field to be identified by an integer instead of a name.

Concrete grammar

<field variable> ::=
<variable> <exclamation mark> { <field name> | <field number> | <as signal> }
| <variable> <full stop> { <field name> | <field number> | <as signal> }

The <field variable> of Basic SDL-2010 is extended to allow the field to be identified by a <field number> or <as signal>.

Model

An <as signal> in a <field variable> is a shorthand for the <field name> of a field of the sort of the <field variable>, 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>.

NOTE – An <as signal> has to be used to denote a field of a choice data type introduced by a <gate definition>, <channel definition area> or <interface definition>.

12.3.3.2 Default initialization

Default initialization of Basic SDL-2010 is extended to include virtuality.

Default initialization of instances of a data type is specified to be virtual by means of the keyword **virtual** in <virtuality>. Redefinition of a virtual default initialization is allowed in specializations. This is indicated by a default initialization with the keyword **redefined** or **finalized** in <virtuality>.

If the derived type contains no <constant expression> in its default initialization, then the derived type does not have a default initialization.

Concrete grammar

<default initialization> ::=
default [<virtuality>] [<constant expression>]

The <constant expression> shall only be omitted if <virtuality> is **redefined** or **finalized**. In that case the *Data-type-definition* or *Syntype-definition* for the <data type definition> or <syntype definition> including the <default initialization> has no *Default-initialization*.

Semantics

Redefinition of a virtual default initialization corresponds closely to redefinition of virtual types (see clause 8.4.2 of [ITU-T Z.102]).

12.3.4 Imperative expression

The transformations described in the *Model* of this clause are made at the same time as the expansion for import is made. A label attached to an action in which an imperative expression appears is moved to the first task inserted during the described transformation. If several imperative expressions appear in an expression, the tasks are inserted in the same order as the imperative expressions appear in the expression.

Abstract grammar

Imperative-expression	=	*Now-expression*
	|	*Pid-expression*
	|	*Timer-active-expression*
	|	*Timer-remaining-duration*
	|	*Active-agents-expression*
	|	*Any-expression*
	|	*State-expression*
	|	*Signal-expression*
	|	*Signallist-expression*

The grammar is extended to include *Any-expression, State-expression, Signal-expression and Signallist-expression*.

Concrete grammar

<imperative expression> ::=
<now expression>
| <pid expression>
| <timer active expression>
| <timer remaining duration>
| <active agents expression>
| <any expression>
| <state expression>
| <signal expression>
| <signallist expression>

12.3.4.1 Now expression

Now expression is as defined in Basic SDL-2010.

12.3.4.2 Pid expression

The `Pid` expression is extended to describe the <create expression> that returns the offspring pid value.

Concrete grammar

<create expression> ::=
create <create body>

A <create body> of a <create expression> represents the *Create-request-node* of the *Create-request-node* as a *Graph-node* as described in [ITU-T Z.101] in the model for <create expression>.

Model

The use of <create expression> in an expression is shorthand for inserting a *Create-request-node* just before the action where the <create expression> occurs, followed by an assignment of **offspring** to an implicitly declared anonymous variable with a `Pid` sort. An access to the implicit variable then replaces the <create expression> 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> items.

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>.

12.3.4.3 Timer active expression and timer remaining duration

The timer active expression and timer remaining duration are as defined in Basic SDL-2010.

12.3.4.4 Active agents expression

The active agents expression is as defined in Basic SDL-2010.

12.3.4.5 Import expression

Concrete grammar

The concrete syntax for an import expression is defined in [ITU-T Z.102].

Semantics

In addition to the semantics defined in [ITU-T Z.102], an import expression is interpreted as a variable access (see clause 12.3.2) to the implicit variable for the import expression.

Model

The import expression has implied syntax for the importing of the result as defined in [ITU-T Z.102] and also has an implied *Variable-access* of the implied variable for the import in the context where the <import expression> appears.

The use of <import expression> in an expression is shorthand for inserting a task just before the action, where the expression occurs which assigns to an implicit variable the result of the <import expression> and then uses that implicit variable in the expression. It is not valid for an <import expression> to occur several times in an expression or one action.

12.3.4.6 Any expression

Any-expression is useful for modelling behaviour, where stating a specific data item would imply over-specification. From a result returned by an *Any-expression*, no assumption should be made on other results returned by subsequent or previous interpretation of the *Any-expression*.

Abstract grammar

Any-expression :: *Sort-reference-identifier*

Concrete grammar

<any expression> ::=
any (<sort> **)**

Semantics

An *Any-expression* returns an unspecified element of the sort or syntype designated by *Sort-reference-identifier*, if that sort or syntype is a value sort. If *Sort-reference-identifier* denotes *a Syntype-identifier*, the result will be within the range of that syntype. If the sort or syntype designated by *Sort-reference-identifier* is a pid sort, the *Any-expression* returns `Null`.

12.3.4.7 State expression

Abstract grammar

State-expression :: { }

Concrete grammar

<state expression> ::=
state

Semantics

A state expression is interpreted as a `Charstring` that contains the spelling of the name of the most recently entered state of the nearest enclosing scope unit. If there is no such state, <state expression> denotes the empty string (").

12.3.4.8 Signal expression

A *Signal-expression* accesses the last signal value stored by an *In-choice* of an *Input-node* for the implicit anonymous choice variable that is capable of holding any of the values of signals in the valid input signal set of the agent.

Abstract grammar

Signal-expression :: { }

Concrete grammar

<signal expression> ::=
signal

Semantics

Every agent instance has an implicit anonymous choice variable that is capable of holding any of the values of signals in the valid input signal set of the agent. When the agent is initialized this variable is undefined. The variable is updated by an *Input-node* with an *In-choice* with a *Variable-identifier* that identifies the implicit anonymous choice variable. The implicit anonymous choice variable is not updated if the *Input-node* does not have an *In-choice* or the *Variable-identifier* of the *In-choice* identifies another variable.

A *Signal-expression* accesses the implicit anonymous choice variable of the agent with the choice data type that is capable of holding any of the values of signals in the valid input signal set of the agent: that is, the data type defined by "**`as interface`** `agent_id`" where `agent_id` is the identifier for the agent.

12.3.4.9 Signallist expression

A *Signallist-expression* represents a variable that holds the unconsumed, available signals in the agent input port.

Abstract grammar

Signallist-expression :: { }

Concrete grammar

<signallist expression> ::=
signallist

<as signallist> ::=
as signallist

The <as signallist> represents the data type of the implicit read-only string variable that it is assumed is used for storing unconsumed signal instances that have arrived at the agent.

Semantics

Every agent instance has an input port that holds the unconsumed, available signal instances that have arrived at the agent.

If the name of the agent is `agent_name`, it has an implicitly defined interface with the name `agent_name` with the valid input signal set of the agent as the *Signal-identifier-**set*** of the interface. There is a choice data type defined for the interface with a choice of sort for each distinct *Signal-definition* identified by the *Signal-identifier-**set*** of the *Interface-definition* (see clause 12.1.2, Interface definition). Given that the name of the agent is `agent_name`, the anonymous implicit name of this choice data type is denoted by "**as interface** `agent_name`".

The signal instances in the input port are assumed to be stored in order of arrival in an implicit read-only string variable with a data type that is

```
value type AnonSignalString
   inherits String < as interface agent_name > ( empty = emptystring )
endvalue type AnonSignalString;
```

where `AnonSignalString` is an anonymous unique name.

The *Signallist-expression* has a sort, which is the string sort defined as above. The *Signallist-expression* represents the implicit variable.

The *Signallist-expression* provides access to each of the unconsumed available signals in the input port as an element of the implicit read-only string variable by indexing the *Signallist-expression* with a positive `Natural` *Expression*.

`length(`**`signallist`**`)` is the number of available signals (signal instances with availability time > `now` are ignored) and has the value zero if there are no available signals in the input port.

`signallist` `= empty` if and only if there are no available signals in the input port.

`first(`**`signallist`**`)` is the first signal instance to be considered in a state, and has the same meaning as **`signallist`**`[ 1 ]`. Whether this signal instance is consumed depends on whether for the current state this signal saved, or whether there are inputs with higher priority.

`signallist``[ n ]` where `n` is a Natural expression is the n^{th} available signal in the input port – a choice value.

`signallist``[ n ]!Present` gives a literal that is the name of the n^{th} available signal in the input port and one of the literals for the anonymous data type with the field names of the choice sort **`as interface`**, and can be used to make a decision based on the signal name.

Once the name of the n^{th} signal in the input port has been determined, the parameters of the signal can be extracted using a field name (if one was given in the signal definition) or field number. For example:

```
signal signal1 ( Pid, Integer);
signal signal2 (b2 Boolean, i2 Integer);
value type sig3struct { struct c3 Charstring; } endvalue type sig3struct;
signal signal3 struct sig3struct;
dcl agent1 Pid; dcl n Integer ( >= 1) ;
dcl routeNumber2 Integer;
dcl sig3copy sig3struct;
```

Given the definitions above

```
if signallist[n]!Present= signal1 /* is it a signal1 signal */
then agent1 := signallist[n]!as signal signal1!1;
     /* yes, determined by PresentExtract() of the choice value,
        from the choice for signal1 – selected by <as signal>
        (must use <as signal> if signal1 not a unique signal name) -
        extract the first struct field (the Pid field) using <field number>,
        and assign the Pid value */
/* */
if signal2Present (signallist[n]) /* is it a signal2 signal */
then routeNumber2:= signallist[n].signal2!i2;
     /* yes, determined by testing the choice value for signal2
        from the choice for signal2 – selected by unique signal2
        (assumes signal2 is a unique field name) -
        extract the named i2 field of the struct value for the signal,
        and assign the Integer value */
/* */
if signal3Present (signallist[n]) /* is it a signal3 signal */
then sig3copy:= signallist[n]!signal3;
     /* yes, determined by testing the choice value for signal3
        assign the choice for signal3*/
```

NOTE – As well as having a signal type and the actual parameters of the instance, a signal instance in the input port has additional information associated with it: the sender `Pid`, the availability time (always <= `now`), the signal priority and the arrival gate. None of these are accessible using *Signal-expression*. In a particular application it is suggested to place this information in signal parameters to make it available – probably the best way if ASN.1 and encoding is being used.

12.3.5 Value returning procedure call

The value returning procedure call is extended to cover remote procedures, additional constraints and transformation models.

Concrete grammar

<value returning procedure call> ::=
[**call**] <procedure call body>
| [**call**] <remote procedure call body>

A <value returning procedure call> shall not occur in the <Boolean expression> of a <continuous signal area> or <enabling condition area>.

The <remote procedure call body> represents a *Value-returning-call-node*, where *Procedure-identifier* contains only the *Procedure-identifier* of the procedure implicitly defined by the *Model* in clause 10.5 Remote procedure of [ITU-T Z.102].

Model

NOTE – When 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 as described in clause 11.13.3 *Model* of [ITU-T Z.101].

13 Generic system definition

See [ITU-T Z.102] and [ITU-T Z.103].

14 Package Predefined

In the following definitions, all references to names defined in the package Predefined are considered to be treated as prefixed by the qualification <<**package** Predefined>>. To increase readability, this qualification is omitted.

The extensions defined in Annex A are used in this package. It is not allowed to use these extensions in a user defined package.

```
/* */
package Predefined
/*
```

14.1 Boolean sort

14.1.1 Definition

```
*/
value type Boolean;
  literals unordered true, false;
  operators
      "not" ( this Boolean )                -> this Boolean;
      "and" ( this Boolean, this Boolean )  -> this Boolean;
      "or"  ( this Boolean, this Boolean )  -> this Boolean;
      "xor" ( this Boolean, this Boolean )  -> this Boolean;
      "=>"  ( this Boolean, this Boolean )  -> this Boolean;
axioms
     not( true  )     == false;
     not( false )     == true ;
/* */
     true  and true   == true ;
     true  and false  == false;
     false and true   == false;
     false and false  == false;
/* */
     true  or true    == true ;
     true  or false   == true ;
     false or true    == true ;
     false or false   == false;
/* */
     true  xor true   == false;
     true  xor false  == true ;
     false xor true   == true ;
     false xor false  == false;
/* */
     true  => true    == true ;
     true  => false   == false;
     false => true    == true ;
     false => false   == true ;
endvalue type Boolean;
/*
```

14.1.2 Usage

The Boolean sort is used to represent true and false values. Often it is used as the result of a comparison.

The Boolean sort is used widely throughout SDL.

14.2 Character sort

14.2.1 Definition

```
*/
value type Character;
  literals
     NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
     BS,  HT,  LF,  VT,  FF,  CR,  SO,  SI,
     DLE, DC1, DC2, DC3, DC4, NAK, SYN, ETB,
     CAN, EM,  SUB, ESC, IS4, IS3, IS2, IS1,
     ' ', '!', '"', '#', '$', '%', '&', '''',
```

```
     '(', ')', '*', '+', ',', '-', '.', '/',
     '0', '1', '2', '3', '4', '5', '6', '7',
     '8', '9', ':', ';', '<', '=', '>', '?',
     '@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
     'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
     'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
     'X', 'Y', 'Z', '[', '\', ']', '^', '_',
     '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
     'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
     'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
     'x', 'y', 'z', '{', '|', '}', '~', DEL;
/* ''''' is an apostrophe, ' ' is a space, '~' is a tilde */
/* */
  operators
     chr  ( Integer ) -> this Character;
/*   "<", "<=", ">", ">=", and "num" are implicitly defined (see clause 12.1.6.1). */
axioms
     for all a,b in Character (
     for all i in Integer (
/* definition of Chr */
     chr(num(a)) == a;
     chr(i+128)  == chr(i);
     ));
endvalue type Character;
/*
```

14.2.2 Usage

The Character sort is used to represent characters of the International Reference Alphabet [ITU-T T.50].

14.3 String sort

14.3.1 Definition

```
*/
value type String < type Itemsort >;
/* Strings are "indexed" from one */
  operators
     emptystring                                  -> this String;
     mkstring   ( Itemsort                    )  -> this String;
     Make       ( Itemsort                    )  -> this String;
     length     ( this String                    ) -> Integer;
     first      ( this String                    ) -> Itemsort;
     last       ( this String                    ) -> Itemsort;
     "//"       ( this String, this String       ) -> this String;
     Extract    ( this String, Integer           ) -> Itemsort raise InvalidIndex;
     Modify     ( this String, Integer, Itemsort ) -> this String;
     substring  ( this String, Integer, Integer ) -> this String raise InvalidIndex;
     /* substring (s,i,j) gives a string of length j starting from the ith element */
     remove     ( this String, Integer, Integer ) -> this String;
     /* remove (s,i,j) gives a string with a substring of length j starting from
        the ith element removed */
axioms
     for all e in Itemsort ( /*e - element of Itemsort*/
     for all s,s1,s2,s3 in String (
     for all i,j in Integer (
/* constructors are emptystring, mkstring, and "//" */
/* equalities between constructor terms */
     s // emptystring== s;
     emptystring // s== s;
     (s1 // s2) // s3== s1 // (s2 // s3);
/* */
/* definition of length by applying it to all constructors */
     <<type String>>length(emptystring)== 0;
     <<type String>>length(mkstring(e))== 1;
     <<type String>>length(s1 // s2)    == length(s1) + length(s2);
     Make(s)                              == mkstring(s);
/* */
/* definition of Extract by applying it to all constructors,
       error cases handled separately */
```

```
     Extract(mkstring(e),1)                    == e;
     i <= length(s1) ==> Extract(s1 // s2,i)== Extract(s1,i);
     i > length(s1) ==> Extract(s1 // s2,i)  == Extract(s2,i-length(s1));
     i<=0 or i>length(s) ==> Extract(s,i)   == raise InvalidIndex;
/* */
/* definition of first and last by other operations */
     first(s)   == Extract(s,1);
     last(s)    == Extract(s,length(s));
/* */
/* definition of substring(s,i,j) by induction on j,
        error cases handled separately */
     i>0 and i-1<=length(s)            ==>
                    substring(s,i,0)   == emptystring;
/* */
     i>0 and j>0 and i+j-1<=length(s)  ==>
                    substring(s,i,j)   == substring(s,i,j-1) // mkstring(Extract(s,i+j-1));
/* */
     i<=0 or j<0 or i+j-1>length(s)    ==>
                    substring(s,i,j)   == raise InvalidIndex;
/* */
/* definition of Modify by other operations */
     Modify(s,i,e) == substring(s,1,i-1) // mkstring(e) // substring(s,i+1,length(s)-i);
/* definition of remove */
     remove(s,i,j)   == substring(s,1,i-1) // substring(s,i+j,length(s)-i-j+1);
     )));
endvalue type String;
/*
```

14.3.2 Usage

The `Make`, `Extract`, and `Modify` operators will typically be used with the shorthand forms defined in clause 12.2.3 of [ITU-T Z.101] and clause 12.3.3.1 of [ITU-T Z.101] for accessing the values of strings and assigning values to strings.

14.4 Charstring sort

14.4.1 Definition

```
*/
value type Charstring
     inherits String < Character > ( '' = emptystring )
     adding ;
        operators ocs in nameclass
             '''' ( (' ':'&') or '''''' or ('(': '~') )+ '''' -> this Charstring;
/* character strings of any length of any characters from a space ' ' to a tilde '~' */
axioms
     for all c in Character nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)                    ==> cs == mkstring(c);
/* string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1                     ==> cs  == cs1 // cs2;
     ));
endvalue type Charstring;
/*
```

14.4.2 Usage

The Charstring sort defines strings of characters.

A Charstring literal is allowed to contain printing characters and spaces.

A non-printing character is usable as a string by using `mkstring`, for example, `mkstring(DEL)`.

Example:

```
synonym newline_prompt Charstring = mkstring(CR) // mkstring(LF) // '$>';
```

14.5 Integer sort

14.5.1 Definition

```
*/
value type Integer;
  literals unordered nameclass (('0':'9')*) ('0':'9'));
  operators
     "-"    ( this Integer                ) -> this Integer;
     "+"    ( this Integer, this Integer  ) -> this Integer;
     "-"    ( this Integer, this Integer  ) -> this Integer;
     "*"    ( this Integer, this Integer  ) -> this Integer;
     "/"    ( this Integer, this Integer  ) -> this Integer raise DivisionByZero;
     "mod"  ( this Integer, this Integer  ) -> this Integer raise DivisionByZero;
     "rem"  ( this Integer, this Integer  ) -> this Integer;
     "<"    ( this Integer, this Integer  ) -> Boolean;
     ">"    ( this Integer, this Integer  ) -> Boolean;
     "<="   ( this Integer, this Integer  ) -> Boolean;
     ">="   ( this Integer, this Integer  ) -> Boolean;
     power  ( this Integer, this Integer  ) -> this Integer;
     integer ( Integer                    ) -> this Integer;
     num     ( this Integer               ) -> Integer;
     bs in nameclass '''' ( (('0' or '1')*'''B') or ((('0':'9') or ('A':'F'))*'''H') )
              -> this Integer;
axioms  noequality
     for all a,b,c in this Integer (
/* constructors are 0, 1, +, and unary - */
/* equalities between constructor terms */
     (a + b) + c          == a + (b + c);
     a + b                == b + a;
     0 + a                == a;
     a + (- a)            == 0;
     (- a) + (- b)      == - (a + b);
     <<type Integer>> - 0 == 0;
     - (- a)              == a;
/* */
/* definition of binary "-" by other operations */
     a - b                == a + (- b);
/* */
/* definition of "*" by applying it to all constructors */
     0 * a                == 0;
     1 * a                == a;
     (- a) * b            == - (a * b);
     (a + b) * c          == a * c + b * c;
/* */
/* definition of "<" by applying it to all constructors */
     a < b                                    == 0 < (b - a);
     <<type Integer>> 0 < 0                   == false;
     <<type Integer>> 0 < 1                   == true ;
     0 < a           == true ==> 0 < (- a)    == false;
     0 < a and 0 < b == true ==> 0 < (a + b)  == true ;
/* */
/* definition of ">", "equal", "<=", and ">=" by other operations */
     a > b                == b < a;
     equal(a, b)          == not(a < b or a > b);
     a <= b               == a < b or a = b;
     a >= b               == a > b or a = b;
/* */
/* definition of "/" by other operations */
     a / 0                                          == raise DivisionByZero;
     a >= 0 and b > a           == true ==> a / b == 0;
     a >= 0 and b <= a and b > 0  == true ==> a / b == 1 + (a-b) / b;
     a >= 0 and b < 0           == true ==> a / b == - (a / (- b));
     a < 0 and b < 0              == true ==> a / b == (- a) / (- b);
     a < 0 and b > 0              == true ==> a / b == - ((- a) / b);
/* */
/* definition of "rem" by other operations */
     a rem b == a - b * (a/b);
/* */
/* definition of "mod" by other operations */
     a >= 0 and b > 0               ==> a mod b  == a rem b;
```

```
     b < 0                              ==> a mod b  == a mod (- b);
     a < 0 and b > 0 and a rem b = 0 ==> a mod b  == 0;
     a < 0 and b > 0 and a rem b < 0 ==> a mod b  == b + a rem b;
     a mod 0 == raise DivisionByZero;
/* */
/* definition of power by other operations */
     power(a, 0)            == 1;
     b > 0 ==> power(a, b)  == a * power(a, b-1);
     b < 0 ==> power(a, b)  == power(a, b+1) / a; );
/* */
/* definition of literals */
     <<type Integer>> 2 == 1 + 1;
     <<type Integer>> 3 == 2 + 1;
     <<type Integer>> 4 == 3 + 1;
     <<type Integer>> 5 == 4 + 1;
     <<type Integer>> 6 == 5 + 1;
     <<type Integer>> 7 == 6 + 1;
     <<type Integer>> 8 == 7 + 1;
     <<type Integer>> 9 == 8 + 1;
/* */
/* literals other than 0 to 9 */
     for all a,b,c in this Integer nameclass (
     spelling(a) == spelling(b) // spelling(c),
     length(spelling(c)) == 1               ==> a == b * (9 + 1) + c;
     );
/* */
/* hex and binary representation of Integer */
     for all b in Bitstring nameclass (
     for all i in bs nameclass (
     spelling(i) == spelling(b)             ==> i == <<type Bitstring>>num(b);
     ));
/* integer makes parent sort into subsort */
/* num makes subsort into parent sort */
     for all i in this Integer (
     for all p in Integer (
     spelling(i) == spelling(p)             ==> num(i) == p ;
     ));
     integer(num(i)) == i;
endvalue type Integer;
/*
```

14.5.2 Usage

The Integer sort is used for mathematical integers with decimal, hex, or binary notation.

14.6 Natural syntype

14.6.1 Definition

```
*/
syntype Natural = Integer constants >= 0; endsyntype Natural;
/*
```

14.6.2 Usage

The natural syntype is used when positive integers only are required. All operators will be the integer operators but when a value is used as a parameter or assigned, the value is checked. A negative value will be an error.

14.7 Real sort

14.7.1 Definition

```
*/
value type Real;
  literals unordered nameclass
           ('0':'9')* ('0':'9').('0':'9')('0':'9')*
             ('' or (('e' or 'E') ('' or '+' or '-' ) ('0':'9') ('0':'9')* ) );
/*that is, decimal notation with an optional exponent */
  operators
```

```
     "-"   ( this Real             ) -> this Real;
     "+"   ( this Real, this Real  ) -> this Real;
     "-"   ( this Real, this Real  ) -> this Real;
     "*"   ( this Real, this Real  ) -> this Real;
     "/"   ( this Real, this Real  ) -> this Real raise DivisionByZero;
     "<"   ( this Real, this Real  ) -> Boolean;
     ">"   ( this Real, this Real  ) -> Boolean;
     "<="  ( this Real, this Real  ) -> Boolean;
     ">="  ( this Real, this Real  ) -> Boolean;
     float ( Integer               ) -> this Real;
     fix   ( this Real             ) -> Integer;
axioms noequality
     for all r,s in Real (
     for all a,b,c,d in Integer (
/* constructors are float and "/" */
/* equalities between constructor terms allow to reach always a form
        float(a) / float(b) where b > 0 */
     r / float(0)                                      == raise DivisionByZero;
     r / float(1)                                      == r;
     c /= 0 ==> float(a) / float(b)                    == float(a*c) / float(b*c);
     b /= 0 and d /= 0 ==>
     (float(a) / float(b)) / (float(c) / float(d))  == float(a*d) / float(b*c);
/* */
/* definition of unary "-" by applying it to all constructors */
     - (float(a) / float(b))                        == float(- a) / float(b);
/* */
/* definition of "+" by applying it to all constructors */
     (float(a) / float(b)) + (float(c) / float(d))  ==float(a*d + c*b) / float(b*d);
/* */
/* definition of binary "-" by other operations */
     r - s                                             == r + (- s);
/* */
/* definition of "*" by applying it to all constructors */
     (float(a) / float(b)) * (float(c) / float(d))  == float(a*c) / float(b*d);
/* */
/* definition of "<" by applying it to all constructors */
     b > 0 and d > 0 ==>
     (float(a) / float(b)) < (float(c) / float(d))  == a * d < c * b;
/* */
/* definition of ">", "equal", "<=", and ">=" by other operations */
     r > s      == s < r;
     equal(r, s) == not(r < s or r > s);
     r <= s     == r < s or r = s;
     r >= s     == r > s or r = s;
/* */
/* definition of fix by applying it to all constructors */
     a >= b and b > 0==> fix(float(a) / float(b)) == fix(float(a-b) / float(b)) + 1;
     b > a  and a >= 0  ==> fix(float(a) / float(b)) == 0;
     a < 0  and b > 0==> fix(float(a) / float(b)) == - fix(float(-a)/float(b)) - 1;));
/* */
     for all r,s in Real nameclass (
     for all i,j,k in Integer nameclass (
        spelling(r) == spelling(i)          ==> r == float(i);
/* */
        spelling(r) == spelling(i)          ==> i == fix(r);
/* */
        spelling(r) == spelling(i) // spelling(s),
        spelling(s) == '.' // spelling(j) ==> r == float(i) + s;
/* */
        spelling(r) == '.' // spelling(i),
        length(spelling(i)) == 1            ==> r == float(i) / 10;
/* */
        spelling(r) == '.' // spelling(i) // spelling(j),
        length(spelling(i)) == 1,
        spelling(s) == '.' // spelling(j) ==> r == (float(i) + s) / 10;
/* */
        spelling(r) == spelling(i) // '.' // spelling(j) // 'e' // spelling (k),
        spelling(s) == spelling(i) // '.' // spelling(j) ==> r == s * float(power(10,k));
/* */
        spelling(r) == spelling(i) // '.' // spelling(j) // 'E' // spelling (k),
        spelling(s) == spelling(i) // '.' // spelling(j) ==> r == s * float(power(10,k));
```

```
/* */
        spelling(r) == spelling(i) // '.' // spelling(j) // 'e+' // spelling (k),
        spelling(s) == spelling(i) // '.' // spelling(j) ==> r == s * float(power(10,k));
/* */
        spelling(r) == spelling(i) // '.' // spelling(j) // 'E+' // spelling (k),
        spelling(s) == spelling(i) // '.' // spelling(j) ==> r == s * float(power(10,k));
/* */
        spelling(r) == spelling(i) // '.' // spelling(j) // 'e-' // spelling (k),
        spelling(s) == spelling(i) // '.' // spelling(j) ==> r == s / float(power(10,k));
/* */
        spelling(r) == spelling(i) // '.' // spelling(j) // 'E-' // spelling (k),
        spelling(s) == spelling(i) // '.' // spelling(j) ==> r == s / float(power(10,k));
     ));
endvalue type Real;
/*
```

14.7.2 Usage

The real sort is used to represent real numbers.

The real sort represents all numbers which are able to be represented as one integer divided by another.

Numbers which are not able to be represented in this way (irrational numbers – for example, the square root of 2) are not part of the real sort. However, for practical engineering a sufficiently accurate approximation is usually usable.

14.8 Array sort

14.8.1 Definition

```
*/
value type Array < type Index; type Itemsort >;
  operators
     Make                                  -> this Array ;
     Make   ( Itemsort                   ) -> this Array ;
     Modify ( this Array,Index,Itemsort ) -> this Array ;
     Extract( this Array,Index          ) -> Itemsort raise InvalidIndex;
axioms
     for all item, itemi, itemj in Itemsort (
     for all i, j in Index (
     for all a, s in Array (
        <<type Array>>Extract(make,i)                          == raise InvalidIndex;
        <<type Array>>Extract(make(item),i)                    == item ;
        i = j ==> Modify(Modify(s,i,itemi),j,item)             == Modify(s,i,item);
        i = j ==> Extract(Modify(a,i,item),j)                  == item ;
        i = j == false ==> Extract(Modify(a,i,item),j)         == Extract(a,j);
        i = j == false ==> Modify(Modify(s,i,itemi),j,itemj)==
                                                    Modify(Modify(s,j,itemj),i,itemi);
/*equality*/
        <<type Array>>Make(itemi) = Make(itemj)                == itemi = itemj;
        a=s == true, i=j == true, itemi = itemj ==>
                 Modify(a,i,itemi) = Modify(s,j,itemj)         == true;
/* */
        Extract(a,i) = Extract(s,i) == false ==>  a = s        == false;)));
endvalue type Array;
/*
```

14.8.2 Usage

An array is used to define one sort which is indexed by another. For example:

```
value type indexbychar inherits Array< Character, Integer >
endvalue type indexbychar;
```

defines an array containing integers and indexed by characters.

Arrays are usually used in combination with the shorthand forms of `Make`, `Modify`, and `Extract` defined in clause 12.2.3 of [ITU-T Z.101] and clause 12.3.3.1 of [ITU-T Z.101]. For example:

```
dcl charvalue indexbychar;
```

```
        task charvalue := (. 12 .);
        task charvalue('A') := charvalue('B')-1;
```

14.9 Vector

14.9.1 Definition

```
*/
value type Vector < type Itemsort; synonym MaxIndex >
     inherits Array< Indexsort, Itemsort >;
syntype Indexsort = Integer constants 1:MaxIndex endsyntype;
endvalue type Vector;
/*
```

14.10 Powerset sort

14.10.1 Definition

```
*/
value type Powerset < type Itemsort >;
  operators
     empty                                    -> this Powerset;
     "in"   ( Itemsort, this Powerset       ) -> Boolean;       /* is member of          */
     incl   ( Itemsort, this Powerset       ) -> this Powerset; /* include item in set   */
     del    ( Itemsort, this Powerset       ) -> this Powerset; /* delete item from set  */
     "<"    ( this Powerset, this Powerset  ) -> Boolean;       /* is proper subset of   */
     ">"    ( this Powerset, this Powerset  ) -> Boolean;       /* is proper superset of */
     "<="   ( this Powerset, this Powerset  ) -> Boolean;       /* is subset of          */
     ">="   ( this Powerset, this Powerset  ) -> Boolean;       /* is superset of        */
     "and"  ( this Powerset, this Powerset  ) -> this Powerset; /* intersection of sets  */
     "or"   ( this Powerset, this Powerset  ) -> this Powerset; /* union of sets         */
     length ( this Powerset                 ) -> Integer;
     take   ( this Powerset                 ) -> Itemsort raise Empty;
axioms
     for all i,j in Itemsort (
     for all p,ps,a,b,c in Powerset (
/* constructors are empty and incl */
/* equalities between constructor terms */
     incl(i,incl(j,p))                        == incl(j,incl(i,p));
     i = j ==> incl(i,incl(j,p))              == incl(i,p);
/* definition of "in" by applying it to all constructors */
     i in <<type Powerset>>empty              == false;
     i in incl(j,ps)                          == i=j or i in ps;
/* definition of del by applying it to all constructors */
     <<type Powerset>>del(i,empty)            == empty;
     i =  j ==> del(i,incl(j,ps))           == del(i,ps);
     i /= j ==> del(i,incl(j,ps))           == incl(j,del(i,ps));
/* definition of "<" by applying it to all constructors */
     a <  <<type Powerset>>empty              == false;
     <<type Powerset>>empty < incl(i,b)== true;
     incl(i,a) < b                          == i in b and del(i,a) < del(i,b);
/* definition of ">" by other operations */
     a > b == b < a;
/* definition of "=" by applying it to all constructors */
     empty = incl(i,ps)                     == false;
     incl(i,a) = b     == i in b and del(i,a) = del(i,b);
/* definition of "<=" and ">=" by other operations */
     a <= b                                   == a < b or a = b;
     a >= b                                   == a > b or a = b;
/* definition of "and" by applying it to all constructors */
     empty and b                              == empty;
     i in b     ==> incl(i,a) and b         == incl(i,a and b);
     not(i in b) ==> incl(i,a) and b          == a and b;
/* definition of "or" by applying it to all constructors */
     empty or b                               == b;
     incl(i,a) or b                           == incl(i,a or b);
/* definition of length */
     length(<<type Powerset>>empty)           == 0;
     i in ps ==> length(ps)                   == 1 + length(del(i, ps));
/* definition of take */
```

```
     take(empty)                          == raise Empty;
     i in ps ==> take(ps)                 == i;
     ));
endvalue type Powerset;
/*
```

14.10.2 Usage

Powersets are used to represent mathematical sets. For example:

```
value type Boolset inherits Powerset< Boolean > endvalue type Boolset;
```

is used for a variable which is allowed to be empty or contain (`true`), (`false`) or (`true, false`).

14.11 Duration sort

14.11.1 Definition

```
*/
value type Duration;
  literals unordered nameclass
           ('0':'9')* ('0':'9') '.'  ('0':'9') ('0':'9')*
            ('' or (('e' or 'E') ('' or '+' or '-' ) ('0':'9') ('0':'9')* ) );
  operators
     protected duration ( Real           ) -> this Duration;
     "+"   ( this Duration, this Duration ) -> this Duration;
     "-"   ( this Duration                ) -> this Duration;
     "-"   ( this Duration, this Duration ) -> this Duration;
     ">"   ( this Duration, this Duration ) -> Boolean;
     "<"   ( this Duration, this Duration ) -> Boolean;
     ">="  ( this Duration, this Duration ) -> Boolean;
     "<="  ( this Duration, this Duration ) -> Boolean;
     "*"   ( this Duration, Real          ) -> this Duration;
     "*"   ( Real, this Duration          ) -> this Duration;
     "/"   ( this Duration, Real          ) -> Duration;
axioms noequality
/* constructor is duration(Real)*/
     for all a, b in Real nameclass (
     for all d, e in Duration nameclass (
/* definition of "+" by applying it to all constructors */
     duration(a) + duration(b)    == duration(a + b);
/* */
/* definition of unary "-" by applying it to all constructors */
     - duration(a)              == duration(-a);
/* */
/* definition of binary "-" by other operations */
     d - e                        == d + (-e);
/* */
/* definition of "equal", ">", "<", ">=", and "<=" by applying it to all constructors */
     equal(duration(a), duration(b)) == a = b;
     duration(a) > duration(b)    == a > b;
     duration(a) < duration(b)    == a < b;
     duration(a) >= duration(b) == a >= b;
     duration(a) <= duration(b) == a <= b;
/* */
/* definition of "*" by applying it to all constructors */
     duration(a) * b              == duration(a * b);
     a * d                        == d * a;
/* */
/* definition of "/" by applying it to all constructors */
     duration(a) / b            == duration(a / b);
/* */
     spelling(d) == spelling(a) ==>
                           d  == duration(a);
     ));
endvalue type Duration;
/*
```

14.11.2 Usage

The duration sort is used for the value to be added to the current time to set timers. The literals of the sort duration are the same as the literals for the real sort. The meaning of one unit of duration will depend on the system being defined.

Duration values are allowed be multiplied and divided by real values. Unless otherwise specified, the time unit is 1 second.

14.12 Time sort

14.12.1 Definition

```
*/
value type Time;
  literals unordered nameclass
           ('0':'9')* ('0':'9') '.' ('0':'9') ('0':'9')*
             ('' or (('e' or 'E') ('' or '+' or '-' ) ('0':'9') ('0':'9')* ) );
  operators
     protected time  ( Duration   ) -> this Time;
     "<"   ( this Time, this Time  ) -> Boolean;
     "<="  ( this Time, this Time  ) -> Boolean;
     ">"   ( this Time, this Time  ) -> Boolean;
     ">="  ( this Time, this Time  ) -> Boolean;
     "+"   ( this Time, Duration) -> this Time;
     "+"   ( Duration, this Time) -> this Time;
     "-"   ( this Time, Duration) -> this Time;
     "-"   ( this Time, this Time  ) -> Duration;
axioms noequality
/* constructor is time */
     for all t, u in Time nameclass (
     for all a, b in Duration nameclass (
/* definition of ">", "equal" by applying it to all constructors */
     time(a) > time(b)          == a > b;
     equal(time(a), time(b))    == a = b;
/* */
/* definition of "<", "<=", ">=" by other operations */
     t < u                      == u > t;
     t <= u                     == (t < u) or (t = u);
     t >= u                     == (t > u) or (t = u);
/* */
/* definition of "+" by applying it to all constructors */
     time(a) + b                == time(a + b);
     a + t                      == t + a;
/* */
/* definition of "-" : Time, Duration by other operations */
     t - b                      == t + (-b);
/* */
/* definition of "-" : Time, Time by applying it to all constructors */
     time(a) - time(b)          == a - b;
/* */
     spelling(a) == spelling(t) ==>
                            a   == time(t);
     ));
endvalue type Time;
/*
```

14.12.2 Usage

The `now` expression returns a value of the time sort. It is allowed that time value has a duration added or subtracted from it to give another time value. A time value subtracted from another time value gives a duration. Time values are used to set the expiry time of timers.

The origin of time is system dependent. A unit of time is the amount of time represented by adding one duration unit to a time. Unless otherwise specified, the time unit is 1 second.

14.13 Bag sort

14.13.1 Definition

```
*/
value type Bag < type Itemsort >;
  operators
     empty                          -> this Bag;
     "in"   ( Itemsort, this Bag  ) -> Boolean;/* is member of  */
     incl   ( Itemsort, this Bag  ) -> this Bag;  /* include item in set     */
     del    ( Itemsort, this Bag  ) -> this Bag;  /* delete item from set    */
     "<"    ( this Bag, this Bag  ) -> Boolean;/* is proper subbag of       */
     ">"    ( this Bag, this Bag  ) -> Boolean;/* is proper superbag of     */
     "<="   ( this Bag, this Bag  ) -> Boolean;/* is subbag of  */
     ">="   ( this Bag, this Bag  ) -> Boolean;/* is superbag of            */
     "and"  ( this Bag, this Bag  ) -> this Bag;  /* intersection of bags    */
     "or"   ( this Bag, this Bag  ) -> this Bag;  /* union of bags           */
     length ( this Bag            ) -> Integer;
     take   ( this Bag            ) -> Itemsort raise Empty;
axioms
     for all i,j in Itemsort (
     for all p,ps,a,b,c in Bag (
/* constructors are empty and incl */
/* equalities between constructor terms */
     incl(i,incl(j,p))              == incl(j,incl(i,p));
/* definition of "in" by applying it to all constructors */
     i in <<type Bag>>empty         == false;
     i in incl(j,ps)                == i=j or i in ps;
/* definition of del by applying it to all constructors */
     <<type Bag>>del(i,empty)       == empty;
     i =  j ==> del(i,incl(j,ps))  == ps;
     i /= j ==> del(i,incl(j,ps))  == incl(j,del(i,ps));
/* definition of "<" by applying it to all constructors */
     a <  <<type Bag>>empty         == false;
     <<type Bag>>empty < incl(i,b)  == true;
     incl(i,a) < b                 == i in b and del(i,a) < del(i,b);
/* definition of ">" by other operations */
     a > b == b < a;
/* definition of "=" by applying it to all constructors */
     empty = incl(i,ps)            == false;
     incl(i,a) = b                 == i in b and del(i,a) = del(i,b);
/* definition of "<=" and ">=" by other operations */
     a <= b                         == a < b or a = b;
     a >= b                         == a > b or a = b;
/* definition of "and" by applying it to all constructors */
     empty and b                    == empty;
     i in b     ==> incl(i,a) and b == incl(i,a and b);
     not(i in b) ==> incl(i,a) and b == a and b;
/* definition of "or" by applying it to all constructors */
     empty or b                     == b;
     incl(i,a) or b                 == incl(i,a or b);
/* definition of length */
     length(<<type Bag>>empty)      == 0;
     i in ps    ==> length(ps)      == 1 + length(del(i, ps));
/* definition of take */
     take(empty)                    == raise Empty;
     i in ps    ==> take(ps)        == i; ));
endvalue type Bag;
/*
```

14.13.2 Usage

Bags are used to represent multi-sets. For example:

```
value type Boolset inherits Bag< Boolean > endvalue type Boolset;
```

is used for a variable which allowed to be empty or contain (`true`), (`false`), (`true, false`) (`true, true`), (`false, false`),... .

Bags are used to represent the `SET OF` construction of ASN.1.

14.14 ASN.1 Bit and Bitstring sorts

14.14.1 Definition

```
*/
value type Bit
     inherits Boolean ( 0 = false, 1 = true );
     adding;
  operators
     num ( this Bit  ) -> Integer;
     bit ( Integer) -> this Bit raise OutOfRange;
axioms
     <<type Bit>>num (0)           == 0;
     <<type Bit>>num (1)           == 1;
     <<type Bit>>bit (0)           == 0;
     <<type Bit>>bit (1)           == 1;
     for all i in Integer (
        i > 1 or i < 0 ==> bit (i)== raise OutOfRange;
        )
endvalue type Bit;
/* */
value type Bitstring
  operators
     bs in nameclass
     '''' ( (('0' or '1')*'''B') or ((('0':'9') or ('A':'F'))*'''H') )-> this Bitstring;
/*The following operators that are the same as String except Bitstring
     is indexed from zero*/
     mkstring   (Bit                              ) -> this Bitstring;
     Make       (Bit                              ) -> this Bitstring;
     length     ( this Bitstring                  ) -> Integer;
     first      ( this Bitstring                  ) -> Bit;
     last       ( this Bitstring                  ) -> Bit;
     "//"       ( this Bitstring, this Bitstring  ) -> this Bitstring;
     Extract    ( this Bitstring, Integer         ) -> Bit raise InvalidIndex;
     Modify     ( this Bitstring, Integer, Bit    ) -> this Bitstring;
     substring  ( this Bitstring, Integer, Integer) -> this Bitstring raise InvalidIndex;
     /* substring (s,i,j) gives a string of length j starting from the ith element */
     remove     ( this Bitstring, Integer, Integer) -> this Bitstring;
     /* remove (s,i,j) gives a string with a substring of length j starting from
        the ith element removed */
/*The following operators are specific to Bitstrings*/
     "not" ( this Bitstring                 ) -> this Bitstring;
     "and" ( this Bitstring, this Bitstring ) -> this Bitstring;
     "or"  ( this Bitstring, this Bitstring ) -> this Bitstring;
     "xor" ( this Bitstring, this Bitstring ) -> this Bitstring;
     "=>"  ( this Bitstring, this Bitstring ) -> this Bitstring
     num   ( this Bitstring                 ) -> Integer;
     bitstring ( Integer                    ) -> this Bitstring raise OutOfRange;
     octet ( Integer                        ) -> this Bitstring raise OutOfRange;
axioms
/* Bitstring starts at index 0 */
/* Definition of operators with the same names as String operators*/
     for all b in Bit ( /*b is bit in string*/
     for all s,s1,s2,s3 in Bitstring (
     for all i,j in Integer (
/* constructors are ''B, mkstring, and "//" */
/* equalities between constructor terms */
     s // ''B          == s;
     ''B// s           == s;
     (s1 // s2) // s3== s1 // (s2 // s3);
/* definition of length by applying it to all constructors */
     <<type Bitstring>>length(''B)               == 0;
     <<type Bitstring >>length(mkstring(b))      == 1;
     <<type Bitstring >>length(s1 // s2)         == length(s1) + length(s2);
     Make(s)                                     == mkstring(s);
/* definition of Extract by applying it to all constructors,
        with error cases handled separately */
     Extract(mkstring(b),0)                      == b;
     i < length(s1)       ==> Extract(s1 // s2,i) == Extract(s1,i);
     i >= length(s1)      ==> Extract(s1 // s2,i) == Extract(s2,i-length(s1));
     i<0 or i=>length(s)  ==> Extract(s,i)        == raise InvalidIndex;
```

```
/* definition of first and last by other operations */
     first(s)   == Extract(s,0);
     last(s)    == Extract(s,length(s)-1);
/* definition of substring(s,i,j) by induction on j,
        error cases handled separately */
     i>=0 and i < length(s)             ==>
          substring(s,i,0)  == ''B;
/* */
     i>=0 and j>0 and i+j<=length(s) ==>
          substring(s,i,j)  == substring(s,i,j-1) // mkstring(Extract(s,i+j));
/* */
     i<0 or j<0 or i+j>length(s)       ==>
          substring(s,i,j)  == raise InvalidIndex;
/* */
/* definition of Modify by other operations */
     Modify(s,i,b) == substring(s,0,i) // mkstring(b) // substring(s,i+1,length(s)-i);
/* definition of remove */
     remove(s,i,j)   == substring(s,0,i) // substring(s,i+j,length(s)-i-j);
     )));
/*end of definition of string operators indexed from zero*/
/* */
/* Definition of ''H and 'x'H in terms of ''B, 'xxxx'B for Bitstring*/
     <<type Bitstring>>''H  == ''B;
     <<type Bitstring>>'0'H == '0000'B;
     <<type Bitstring>>'1'H == '0001'B;
     <<type Bitstring>>'2'H == '0010'B;
     <<type Bitstring>>'3'H == '0011'B;
     <<type Bitstring>>'4'H == '0100'B;
     <<type Bitstring>>'5'H == '0101'B;
     <<type Bitstring>>'6'H == '0110'B;
     <<type Bitstring>>'7'H == '0111'B;
     <<type Bitstring>>'8'H == '1000'B;
     <<type Bitstring>>'9'H == '1001'B;
     <<type Bitstring>>'A'H == '1010'B;
     <<type Bitstring>>'B'H == '1011'B;
     <<type Bitstring>>'C'H == '1100'B;
     <<type Bitstring>>'D'H == '1101'B;
     <<type Bitstring>>'E'H == '1110'B;
     <<type Bitstring>>'F'H == '1111'B;
/* */
/* Definition of Bitstring specific operators*/
     <<type Bitstring>>mkstring(0)          == '0'B;
     <<type Bitstring>>mkstring(1)          == '1'B;
/* */
     for all s, s1, s2, s3 in Bitstring (
     s  =  s                                == true;
     s1 =  s2                               == s2 = s1;
     s1 /= s2                               == not ( s1 = s2 );
     s1 =  s2 == true            ==> s1     == s2;
     ((s1 = s2) and (s2 = s3))  ==> s1 = s3  == true;
     ((s1 = s2) and (s2 /= s3)) ==> s1 = s3  == false;
/* */
     for all b, b1, b2 in Bit (
     not(''B)                       == ''B;
     not(mkstring(b) // s)          == mkstring( not(b) ) // not(s);
/* definition of or */
/* The length of or-ing two strings is the maximal length of both strings */
     ''B or ''B                     == ''B;
     length(s) > 0 ==> ''B or s == mkstring(0) or s;
     s1 or s2                       == s2 or s1;
     mkstring(b1 or b2) // (s1 or s2)   ==(mkstring(b1) // s1) or (mkstring(b2) // s2);
/* */
/* definition of remaining operators based on "or" and "not" */
     s1 and s2                      == not (not s1 or not s2);
     s1 xor s2                      == (s1 or s2) and not(s1 and s2);
     s1 => s2                       == not (s1 and s2);
     ));
/* */
/*Definition of 'xxxxx'B literals, num, bitstring and octet */
     for all s in Bitstring (
     for all b in Bit (
```

```
     for all i in Integer (
     /* definition of num */
     <<type Bitstring>>num (''B)            == 0;
     num (s // mkstring (b))                == num (b) + 2 * num (s);
     /* definition of bitstring */
     <<type Bitstring>>bitstring (0)        == '0'B;
     <<type Bitstring>>bitstring (1)        == '1'B;
     i > 1               ==> bitstring (i)  == bitstring (i / 2) // bitstring (i mod 2);
     i < 0               ==> bitstring (i)  == raise OutOfRange;
     /* definition of octet */
     i >= 128 and i <= 255  ==> octet (i)   == bitstring (i) or '00000000'B;
     i >=   0 and i <= 127  ==> octet (i)   ==
                      substring('00000000',0,8-length(bitstring(i))// bitstring(i);
     /* the bitstring is extended to the left enough zero bits to make it an octet */
     i < 0 or i > 255  ==> octet (i)    == raise OutOfRange;
     )))
/*Definition of 'xxxxx'H literals */
     for all b1,b2,b3,h1,h2,h3 in bs nameclass (
     for all bs1, bs2, bs3, hs1, hs2, hs3 in Charstring (
     spelling(b1) = '''' // bs1 // '''B',
     spelling(b2) = '''' // bs2 // '''B',
     bs1 /= bs2                             ==> b1 = b2    == false;
/* */
     spelling(h1) = '''' // hs1 // '''H',
     spelling(h2) = '''' // hs2 // '''H',
     hs1 /= hs2 ==> h1 = h2 == false;
     spelling(b1) = '''' // bs1 // '''B',
     spelling(b2) = '''' // bs2 // '''B',
     spelling(b3) = '''' // bs1 // bs2 // '''B',
     spelling(h1) = '''' // hs1 // '''H',
     spelling(h2) = '''' // hs2 // '''H',
     spelling(h3) = '''' // hs1 // hs2 // '''H',
     length(bs1) = 4,
     length(hs1) = 1,
     length(hs2) > 0,
     length(bs2) = 4 * length(hs2),
     h1 = b1                                ==> h3 = b3    == h2 = b2;
/* */
/* connection to the String generator */
     for all b in Bit literals (
     spelling(b1)  = '''' // bs1 // bs2 // '''B',
     spelling(b2)  = '''' // bs2 // '''B',
     spelling(b)   = bs1                    ==> b1         == mkstring(b) // b2;
     )));
endvalue type Bitstring;
/*
```

14.15 ASN.1 Octet and Octetstring sorts

14.15.1 Definition

```
*/
syntype Octet = Bitstring size (8);
endsyntype Octet;
/* */
value type Octetstring
     inherits String < Octet > ( ''B = emptystring )
     adding
     operators
     os in nameclass
          '''' ( ((('0' or '1')8)*'''B') or ((((0':'9') or ('A':'F'))2)*'''H') )
           -> this Octetstring;
     bitstring   ( this Octetstring ) -> Bitstring;
     octetstring ( Bitstring ) -> this Octetstring;
axioms
     for all b,b1,b2 in Bitstring (
     for all s in Octetstring (
     for all o in Octet(
     <<type Octetstring>> bitstring(''B)           == ''B;
     <<type Octetstring>> octetstring(''B)         == ''B;
     bitstring( mkstring(o) // s )                 == o // bitstring(s);
```

```
/* */
     length(b1) > 0,
     length(b1) < 8,
     b2 == b1 or '00000000'B ==> octetstring(b1)   == mkstring(b2);
/* */
     b == b1 // b2,
     length(b1) == 8            ==> octetstring(b)== mkstring(b1) // octetstring(b2);
     )));
/* */
     for all b1, b2 in Bitstring (
     for all o1, o2 in os nameclass (
     spelling( o1 ) = spelling( b1 ),
     spelling( o2 ) = spelling( b2 ) ==> o1 = o2 == b1 = b2
     ));
endvalue type Octetstring;
/*
```

14.16 Pid sort

14.16.1 Definition

The data type `Pid` is the basis of all interface types.

```
*/
interface Pid { literals Null }
/*
```

14.17 Encoding sort

The following enumerated data type introduces the literals to support the encoding of data according to the standardized set of encoding rules. The set of literals is allowed to be extended by specializing `Encoding`.

```
*/
value type Encoding
     { literals text, BER, CER, DER, APER, UPER, CAPER, CUPER, BXER, CXER, EXER, OER }
/*
```

which is used to denote the required set of encoding rules as follows:

`text` for the set of text encoding rules defined in this Recommendation and produces a `Charstring`;

`BER` for the set of Basic Encoding Rules of ASN.1 (see [ITU-T X.690]) and produces an `Octetstring`;

`CER` for the set of Canonical Encoding Rules of ASN.1 (see [ITU-T X.690]) and produces an `Octetstring`;

`DER` for the set of Distinguished Encoding Rules of ASN.1 (see [ITU-T X.690]) and produces an `Octetstring`;

`APER` for the basic ALIGNED variant of the Packed Encoding Rules of ASN.1 (see [ITU-T X.691]) and produces an `Octetstring`;

`UPER` for the basic UNALIGNED variant of the Packed Encoding Rules of ASN.1 (see [ITU-T X.691]) and produces a `Bitstring`;

`CAPER` for the ALIGNED variant of Canonical Packed Encoding Rules (CANONICAL-PER ALIGNED) of ASN.1 (see [ITU-T X.691]) and produces an `Octetstring`;

`CUPER` for the UNALIGNED variant of Canonical Packed Encoding Rules (CANONICAL-PER ALIGNED) of ASN.1 (see [ITU-T X.691]) and produces a `Bitstring`;

`BXER` for the Basic XML Encoding Rules (BASIC-XER) of ASN.1 (see [ITU-T X.693]) and produces a `Charstring`;

`CXER` for the Canonical XML Encoding Rules (CANONICAL-XER) of ASN.1 (see [ITU-T X.693]) and produces a `Charstring`;

EXER for the Extended XML Encoding Rules (EXTENDED-XER) of ASN.1 (see [ITU-T X.693]) and produces a Charstring;

OER for the Octet Encoding Rules of ASN.1 (see [ITU T X.696]) and produces an Octetstring.

The synonym PER is added to the **package** Predefined as an alternative for APER as follows:

```
*/
synonym PER Encoding = APER;
/*
```

The operator last of the data type Encoding is redefined to an unknown name, and therefore is inaccessible, so that an application or implementation is able to extend the data type Encoding without any change where only the above rules are used. It is possible to specialize the data type Encoding by adding additional literals.

14.18 Support for ASN.1 character, symbol string and NULL types

The following definitions are specifically to support character, string types and the NULL type of ASN.1. They have the same names in SDL-2010 and ASN.1.

```
*/
syntype NumericChar = Character constants
' ', '0', '1', '2', '3', '4',  '5', '6',
'7', '8', '9' endsyntype;
/* */

/*      NumericString sort      */
/*      Definition      */
value type NumericString
     inherits String < NumericChar > ( '' = emptystring )
     adding
        operators ocs in nameclass
         '''' ( ('0':'9') or '''''' or (' ') )+ '''' -> this NumericString;
/*   character strings of any length of any characters from a space ' ' to a '9'      */
axioms
     for all c in NumericChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)              ==> cs == mkstring(c);
/*   string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1              ==> cs  == cs1 // cs2;
     ));
endvalue type NumericString;
/* */
syntype PrintableChar = Character constants
' ', '0', '1', '2', '3', '4',  '5', '6',
'7', '8', '9', 'A', 'B', 'C',  'D', 'E',
'F', 'G', 'H', 'I', 'J', 'K',  'L', 'M',
'N', 'O', 'P', 'Q', 'R', 'S',  'T', 'U',
'V', 'W', 'X', 'Y', 'Z', 'a',  'b', 'c',
'd', 'e', 'f', 'g', 'h', 'i',  'j', 'k',
'l', 'm', 'n', 'o', 'p', 'q',  'r', 's',
't', 'u', 'v', 'w', 'x', 'y',  'z', '''',
'(', ')', '+', ',', '-', '.',  '/', ':',
'=', '?'
endsyntype;
/* */

/*   PrintableString sort   */
/*   Definition      */
value type PrintableString
     inherits String < PrintableChar > ( '' = emptystring )
     adding
        operators ocs in nameclass
           '''' (
                ' ' or ''''''
            or  ('0': '9') or ('A': 'Z') or ('a': 'z')
            or  '(' or ')' or '+' or ',' or '-' or '.' or '/' or ':' or '=' or '(?'
```

```
          )+ '''' -> this PrintableString;
/*   printable character strings of any length */
axioms
     for all c in PrintableChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)             ==> cs == mkstring(c);
/*   string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1              ==> cs  == cs1 // cs2;
     ));
endvalue type PrintableString;
/* */
syntype TeletexChar = Character constants
/* TeletexString characters as in [ITU-T X.680] clause 41 Table 8 */
endsyntype;
/* */

/*   TeletexString sort */
/*   Definition   */
value type TeletexString
     inherits String < TeletexChar > ( '' = emptystring )
     adding
        operators ocs in nameclass
/*   characters specified in [ITU-T X.680] clause 41 Table 8 for TeletexString  */
         -> this TeletexString;
axioms
     for all c in TeletexChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)             ==> cs == mkstring(c);
/*   string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1              ==> cs  == cs1 // cs2;
     ));
endvalue type TeletexString;

syntype VideotexChar = Character constants
/* VideotexString characters as in [ITU-T X.680] clause 41 Table 8 */
endsyntype;
/* */

/*   VideotexString sort    */
/*   Definition     */
value type VideotexString
     inherits String < VideotexChar > ( '' = emptystring )
     adding
        operators ocs in nameclass
/*   VideotexString characters as in [ITU-T X.680] clause 41 Table 8 */
         -> this VideotexString;
axioms
     for all c in VideotexChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)             ==> cs == mkstring(c);
/*   string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1              ==> cs  == cs1 // cs2;
     ));
endvalue type VideotexString;

syntype IA5Char = Character endsyntype;

syntype IA5String = Charstring endsyntype;

value type UniversalChar
     literals /* see [ITU-T X.680] clause 41.6 */
operators
     uchr   ( Integer ) -> this UniversalChar;
endvalue type;
/* */

/*   UniversalCharString sort   */
/*   Definition      */
```

```
value type UniversalCharString
     inherits String < UniversalChar > ( '' = emptystring )
     adding
        operators ocs in nameclass
/*   see [ITU-T X.680] clause 41.6 */
        -> this UniversalCharString;
axioms
     for all c in UniversalChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)             ==> cs == mkstring(c);
/*   string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1              ==> cs  == cs1 // cs2;
     ));
endvalue type UniversalCharString;
/* */

/*   UTF8String sort */
syntype UTF8String = UniversalCharString endsyntype;
/* */

value type GeneralChar
     literals /* All G and all C sets + SPACE + DELETE
        See [ITU-T X.680] clause 41 Table 8 */
operators
     gchr   ( Integer ) -> this GeneralChar;
endvalue type;

/*   GeneralCharString sort  */
/*   Definition      */
value type GeneralCharString
     inherits String < GeneralChar > ( '' = emptystring )
     adding
        operators ocs in nameclass
/*   All G and all C sets + SPACE + DELETE
        See [ITU-T X.680] clause 41 Table 8 */
        -> this GeneralCharString;
axioms
     for all c in GeneralChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)              ==> cs == mkstring(c);
/*   string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1              ==> cs  == cs1 // cs2;
     ));
endvalue type GeneralCharString;
/* */
syntype GraphicChar = GeneralChar constants
/*   All G sets +SPACE as specified in [ITU-T X.680] clause 41 Table 8 */
endsyntype;
/* */

/*   GraphicCharString sort  */
/*   Definition      */
value type GraphicCharString
     inherits String < GraphicChar > ( '' = emptystring )
     adding
        operators ocs in nameclass
/*   All G sets + SPACE as specified in [ITU-T X.680] clause 41 Table 8 */
        -> this GraphicCharString;
axioms
     for all c in GraphicChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)             ==> cs == mkstring(c);
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1              ==> cs  == cs1 // cs2;
     ));
endvalue type GraphicCharString;

syntype VisibleChar = Character constants
/* VisibleString characters specified in [ITU-T X.680] clause 41 Table 8 */
```

```
endsyntype;
/* */

/*   VisibleString sort */
/*   Definition   */
value type VisibleString
     inherits String < VisibleChar > ( '' = emptystring )
     adding
        operators ocs in nameclass
/*   VisibleString characters specified in [ITU-T X.680] clause 41 Table 8*/
        -> this VisibleString;
axioms
     for all c in VisibleChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)             ==> cs == mkstring(c);
/*   string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1              ==> cs  == cs1 // cs2;
     ));
endvalue type VisibleString;

syntype BMPChar = UniversalChar constants
/* see [ITU-T X.680] clause 41.15 */
endsyntype;
/* */
/*   BMPCharString sort   */
/*    Definition     */
value type BMPCharString
     inherits String < BMPChar > ( '' = emptystring )
     adding
        operators ocs in nameclass
/* see [ITU-T X.680] clause 41.15 */
        -> this BMPCharString;
axioms
     for all c in BMPChar nameclass (
     for all cs, cs1, cs2 in ocs nameclass (
     spelling(cs) == spelling(c)             ==> cs == mkstring(c);
/* string 'A' is formed from character 'A' etc. */
     spelling(cs) == spelling(cs1) // spelling(cs2),
     length(spelling(cs2)) == 1              ==> cs  == cs1 // cs2;
     ));
endvalue type BMPCharString;
/* */

value type NULL
literals NULL
endvalue type;
/*
```

14.19 Predefined exceptions

```
*/
exception
     OutOfRange,          /* A range check has failed. */
     InvalidReference,    /* Null was used incorrectly. Wrong Pid for this signal. */
     NoMatchingAnswer,    /* No answer matched in a decision without else part. */
     UndefinedVariable,   /* A variable was used that is "undefined". */
     UndefinedField,      /* An undefined field of a choice or struct was accessed. */
     InvalidIndex,        /* A String or Array was accessed with an incorrect index. */
     InvalidSort,         /* Dynamic sort failure */
     InvalidCall,         /* Dynamic call failure */
     DivisionByZero;      /* An Integer or Real division by zero was attempted. */
     Empty;               /* No element could be returned. */
/* */
endpackage Predefined;
```

Annex A

Abstract data types and axioms

(This annex forms an integral part of this Recommendation.)

A.1 Introduction

Predefined data items in the Specification and Description Language are based on abstract data types, which are defined in terms of their abstract properties rather than in terms of some concrete implementation. Even though the definition of an abstract data type gives one possible way of implementing that data type, an implementation is not mandated to choose that way of implementing the abstract data type, as long as the same abstract behaviour is preserved.

The predefined data types, including the Boolean sort that defines properties for two literals true and false, are defined in this annex. The two Boolean *terms* true and false shall not be (directly or indirectly) defined to be equivalent. Every Boolean constant expression that is used outside data type definitions shall be interpreted as either true or false. If it is not possible to reduce such an expression to true or false, then the specification is incomplete and allows more than one interpretation of the data type.

A.2 Notation

For this purpose, this annex extends the concrete syntax of the Specification and Description Language by means of describing the abstract properties of the operations added by a data type definition. However, this additional syntax is used for explanation only and does not extend the syntax defined in the main text. A specification using the syntax defined in this annex is therefore not valid SDL-2010.

The abstract properties described here do not specify a specific representation of the predefined data. Instead, an interpretation shall conform to these properties. When an <expression> is interpreted, the evaluation of the expression produces a value (e.g., as the result of an <operation application>). Two expressions, E1 and E2, are equivalent if:

a) there is an <equation> E1 == E2; or

b) one of the equations derived from the given set of <quantified equations>s is E1 == E2; or

c) i) E1 is equivalent to EA; and

   ii) E2 is equivalent to EB; and

   iii) there is an equation EA == EB or an equation derived from the given set of quantified equations such that EA == EB; or

d) by substituting a sub-term of E1 by a term of the same class as the sub-term producing a term E1A, it is possible to show that E1A is in the same class as E2.

Otherwise, the two expressions are not equivalent.

Two expressions that are equivalent represent the same value.

Interpretation of expressions conforms to these properties if two equivalent expressions represent the same value, and two non-equivalent expressions represent different values.

A.2.1 Axioms

Axioms determine which terms represent the same value. From the axioms in a data type definition, the relationship between argument values and result values of operators is determined and hence meaning is given to the operators. Axioms are either given as Boolean axioms or in the form of algebraic equivalence equations.

An operation defined by <axiomatic operation definitions> is treated as a complete definition with respect to specialization. That is, when a data type defined by the package Predefined is specialized and an operation is redefined in the specialized type, all axioms mentioning the name of the operation are replaced by the corresponding definition in the specialized type.

Concrete grammar

<axiomatic operation definitions> ::=
axioms <axioms>

<axioms> ::=
<equation> { <end> <equation>}* [<end>]

<equation> ::=
<unquantified equation>
| <quantified equations>
| <conditional equation>
| <literal equation>
| <noequality>

<unquantified equation> ::=
<term> == <term>
| <Boolean axiom>

<term> ::=
<constant expression>
| <error term>

<quantified equations> ::=
<quantification> **(**<axioms> **)**

<quantification> ::=
for all <value name> { **,** <value name> }* **in** <sort>

NOTE – **for** is considered a keyword for the purpose of this Annex.

This annex changes <operations> (see clause 12.1.1) as described below.
<operations> ::=
<operation signatures>
{ <operation definitions> | <axiomatic operation definitions> }

<axiomatic operation definitions> are only permitted for the description of the behaviour of operators.

An <identifier> which is an unqualified name appearing in a <term> is one of the following:

a) an <operation identifier>;

b) <literal identifier>;

c) a <value identifier> if there is a definition of that name in a <quantification> of <quantified equations> enclosing the <term>, which then shall have a suitable sort for the context.

Semantics

A ground term is a term that does not contain any value identifiers. A ground term represents a particular, known value. For each value in a sort there exists at least one ground term which represents that value.

Each equation is a statement about the algebraic equivalence of terms. The left hand side term and right hand side term are stated to be equivalent so that where one term appears, it is allowed to substitute the other term. When a value identifier appears in an equation, simultaneous substitution is allowed in that equation of the same term for every occurrence of the value identifier. For this substitution, the term is any ground term of the same sort as the value identifier.

Value identifiers are introduced by the value names in quantified equations. A value identifier is used to represent any data values belonging to the sort of the quantification. An equation will hold if the same value is simultaneously substituted for every occurrence of the value identifier in the equation regardless of the value chosen for the substitution.

In general, there is no need or reason to distinguish between a ground term and the result of the ground term. For example, the ground term for the unity Integer element is (normally) written "1". Usually there are several ground terms which denote the same data item, e.g., the Integer ground terms "0+1", "3–2" and "(7+5)/12", and it is usual to consider a simple form of the ground term (in this case "1") as denoting the data item.

A value name is always introduced by quantified equations, and the corresponding value has a value identifier, which is the value name qualified by the sort identifier of the enclosing quantified equations. For example:

```
for all z in X (for all z in X ... )
```

introduces only one value identifier named z of sort X.

In the concrete syntax of axioms, it is not allowed to specify a qualifier for value identifiers.

Each value identifier introduced by quantified equations has a sort, which is the sort identified in the quantified equations by the <sort>.

A term has a sort, which is the sort of the value identifier or the result sort of the (literal) operator.

Unless it is possible to deduce from the equations that two terms denote the same value, each term denotes a different value.

A.2.2 Conditional equations

A conditional equation allows the specification of equations that only hold when certain restrictions hold. The restrictions are written in the form of simple equations.

Concrete grammar

<conditional equation> ::=
<restriction> { , <restriction> }* ==> <restricted equation>

<restricted equation> ::=
<unquantified equation>

<restriction> ::=
<unquantified equation>

Semantics

A conditional equation defines that terms denote the same data item only when any value identifier in the restricted equation denotes a data item, which it is possible to show from other equations that satisfy the restrictions.

The semantics of a set of equations for a data type that includes conditional equations is derived as follows:

a) Quantification is removed by generating every possible ground term equation that is derivable from the quantified equations. As this is applied to both explicit and implicit quantification, a set of unquantified equations in ground terms only is generated.

b) Let a conditional equation be called a provable conditional equation if all the restrictions (in ground terms only) are able to be proved to hold from unquantified equations that are not restricted equations. If there exists a provable conditional equation, it is replaced by the restricted equation of the provable conditional equation.

c) If there are conditional equations remaining in the set of equations and none of these conditional equations are a provable conditional equation, then these conditional equations are deleted; otherwise, return to step b).

d) The remaining set of unquantified equations defines the semantics of the data type.

A.2.3 Equality

Concrete grammar

<noequality> ::=
noequality

Model

Any <data type definition> introducing some sort named S has the following implied <operation signature> in its <operator list>, unless <noequality> is present in the <axioms>:

equal (S, S) -> Boolean;

where Boolean is the predefined Boolean sort.

Any <data type definition> introducing a sort named S such that it contains only <axiomatic operation definitions> in <operator list> has an implied equation set:

```
for all a,b,c in S (
   equal(a, a) == true;
   equal(a, b) == equal(b, a);
   equal(a, b) and equal(b, c)  ==> equal(a, c) == true;
   equal(a, b) == true          ==> a == b;)
```

and an implied <literal equation>:

```
for all L1,L2 in S literals (
   spelling(L1) /= spelling(L2) ==> L1 = L2 == false;)
```

A.2.4 Boolean axioms

Concrete grammar

<Boolean axiom> ::=
<Boolean term>

Semantics

A Boolean axiom is a statement of truth that holds under all conditions for the data type being defined.

Model

An axiom of the form:

<Boolean term>;

is derived syntax for the concrete syntax equation:

<Boolean term> == << **package** Predefined/**type** Boolean >> true;

A.2.5 Conditional term

Semantics

An equation containing a conditional term is semantically equivalent to a set of equations where all the quantified value identifiers in the Boolean term have been eliminated. It is possible to form this set of equations by simultaneously substituting, throughout the conditional term equation, each <value identifier> in the <conditional expression> by each ground term of the appropriate sort. In this set of equations, the <conditional expression> will always have been replaced by a Boolean ground term. In the following, this set of equations is referred to as the expanded ground set.

A conditional term equation is equivalent to the equation set that contains:

a) for every *equation* in the expanded ground set for which the <conditional expression> is equivalent to true, that *equation* from the expanded ground set with the <conditional expression> replaced by the (ground) <consequence expression>; and

b) for every *equation* in the expanded ground set for which the <conditional expression> is equivalent to false, that *equation* from the expanded ground set with the <conditional expression> replaced by the (ground) <alternative expression>.

Note that in the special case of an equation of the form:

ex1 == **if** a **then** b **else** c **fi**;

this is equivalent to the pair of conditional equations:

a == true ==> ex1 == b;
a == false ==> ex1 == c;

A.2.6 Error term

Errors are used to allow the properties of a data type to be fully defined even for cases when it is not possible to give a specific meaning to the result of an operator.

Concrete grammar

<error term> ::=
raise <exception name>

An <error term> shall not be used as part of a <restriction>.

It shall not be possible to derive from *equations* that a <literal identifier> is equal to <error term>.

Semantics

A term is allowed to be an <error term>, so that it is possible to specify the circumstances under which an operator produces an error. If these circumstances arise during interpretation, then the exception with <exception name> is raised.

A.2.7 Unordered literals

Concrete grammar

<unordered> ::=
unordered

This annex changes the concrete syntax for the literal list type constructor (see clause 12.1.6.1 of [ITU-T Z.101]) as follows:

<literal list> ::=
[<visibility>] **literals** [<unordered>]
<literal signature> { **,** <literal signature> }* <end>

Semantics

If <unordered> is used, the ordering operations "<", ">","<=",">=", first, last, pred, succ, and num are not implicitly defined for this data type (see clause 12.1.6.1 of [ITU-T Z.101]).

A.2.8 Literal equations

Concrete grammar

<literal equation> ::=
<literal quantification>
(<equation> { <end> <equation> }* [<end>] **)**

<literal quantification> ::=
for all <value name> { **,** <value name> }* **in** <sort> **literals**
| **for all** <value name> { **,** <value name> }* **in** { <sort> | <value identifier> } **nameclass**

Semantics

Literal mapping is a shorthand for defining a large (possibly infinite) number of axioms ranging over all the literals of a sort or all the names in a name class. The literal mapping allows the literals for a sort to be mapped onto the values of the sort.

<spelling term> is used in literal quantifications to refer to the character string that contains the spelling of the literal. This mechanism allows the `Charstring` operators to be used to define literal equations.

Model

A <literal equation> is shorthand for a set of <axioms>. In each of the <equation>s contained in a <literal equation>, the <value identifier>s defined by the <value name> in the <literal quantification> are replaced. In each derived <equation>, each occurrence of the same <value identifier> is replaced by the same <literal identifier> of the <sort> of the <literal quantification> (if literals was used) or by the same <literal identifier> of the nameclass referred to (if nameclass was used). The derived set of <axioms> contains all possible <equation>s derivable in this way.

The derived <axioms> for <literal equation>s are added to <axioms> (if any) defined after the keyword **axioms**.

If a <literal quantification> contains one or more <spelling term>s, then there is replacement of the <spelling term>s with `Charstring` literals (see clause 14.4). The `Charstring` is used to replace the <value identifier> after the <literal equation> containing the <spelling term> and is expanded as defined below, using <value identifier> in place of <operation name>.

NOTE – Literal equations do not affect nullary operators defined in <operation signature>s.

A.2.9 Name class

A name class is shorthand for a (possibly infinite) set of literal names or operator names defined by a regular expression.

A <name class literal> is an alternative way of specifying a <literal name>. A <name class operation> is an alternative way of specifying an <operation name> of a nullary operation.

Concrete grammar

The syntax for <literal name> is extended with an alternative:

```
                    |   <name class literal>
<name class literal> ::=
                        nameclass <regular expression>
```

The syntax for <operation name> is extended with an alternative:

```
                    |   <name class operation>
<name class operation> ::=
                        <operation name> in nameclass <regular expression>
<regular expression> ::=
                        <partial regular expression> { [or ] <partial regular expression> }*
<partial regular expression> ::=
                        <regular element> [ <integer name> | <plus sign> | <asterisk> ]
<regular element> ::=
                        ( <regular expression> )
                    |   <character string>
```

| <regular interval>

<regular interval> ::=
<character string> { <colon> | <range sign> } <character string>

The names formed by the <regular expression> shall satisfy the lexical rules for names or <character string>, <hex string>, or <bit string>.

The <character string>s in a <regular interval> shall both have a length of one, excluding the leading and trailing <apostrophe>s.

A <name class operation> is allowed only in an <operation signature>. An <operation signature> containing <name class operation> shall only occur in an <operator list> and shall not contain <arguments>.

When a name contained in the equivalent set of names of a <name class operation> occurs as the <operation name> in an <operation application>, it shall not have <actual parameters>.

The equivalent set of names of a name class is defined as the set of names that satisfy the syntax specified by the <regular expression>. The equivalent sets of names for the <regular expression>s contained in a <data type definition> shall not overlap.

Model

A <name class literal> is equivalent to this set of names in the abstract syntax. When a <name class operation> is used in an <operation signature>, a set of <operation signature>s is created by substituting each name in the equivalent set of names for the <name class operation> in the <operation signature>.

A <regular expression> which is a list of <partial regular expression>s without an **or** specifies that the names are formed from the characters defined by the first <partial regular expression> followed by the characters defined by the second <partial regular expression>.

When an **or** is specified between two <partial regular expression>s, then the names are formed from either the first or the second of these <partial regular expression>s. **or** is more tightly binding than simple sequencing.

If a <regular element> is followed by <Natural literal name>, the <partial regular expression> is equivalent to the <regular element> being repeated the number of times specified by the <Natural literal name>.

If a <regular element> is followed by '*' the <partial regular expression> is equivalent to the <regular element> being repeated zero or more times.

If a <regular element> is followed by <plus sign> the <partial regular expression> is equivalent to the <regular element> being repeated one or more times.

A <regular element> which is a bracketed <regular expression> defines the character sequences defined by the <regular expression>.

A <regular element> which is a <character string> defines the character sequence given in the character string (omitting the quotes).

A <regular element> which is a <regular interval> defines all the characters specified by the <regular interval> as alternative character sequences. The characters defined by the <regular interval> are all the characters greater than or equal to the first character and less than or equal to the second character according to the definition of the Character sort (see clause A.2).

The names generated by a <name class literal> are defined first by length (a shorter literal comes before any longer ones) then in lexicographical order according to the ordering of the character sort. The characters are considered case sensitive.

NOTE – Examples exist in clause 14.

A.2.10 Name class mapping

A name class mapping is shorthand for defining a (possibly infinite) number of operation definitions ranging over all the names in a <name class operation>. The name class mapping allows behaviour to be defined for the operators and methods defined by a <name class operation>. A name class mapping takes place when an <operation name> that occurred in a <name class operation> within an <operation signature> of the enclosing <data type definition> is used in <operation definitions> or <operation diagram>s.

A spelling term in a name class mapping refers to the character string that contains the spelling of the name. This mechanism allows the `Charstring` operations to be used to define name class mappings.

Concrete grammar

The syntax for <primary> is extended with an alternative

| <spelling term>

<spelling term> ::=
spelling ({ <operation name> | <value identifier> })

The <value identifier> in a <spelling term> shall be a <value identifier> defined by a <literal quantification>.

A <spelling term> is legal concrete syntax only within an <operation definition> or <operation diagram>, if a name class mapping has taken place.

Model

A name class mapping is shorthand for a set of <operation definition>s or a set of <operation diagram>s. The set of <operation definition>s is derived from an <operation definition> by substituting each name in the equivalent set of names of the corresponding <name class operation> for each occurrence of <operation name> in the <operation definition>. The derived set of <operation definition>s contains all possible <operation definition>s that are able to be generated in this way. The same procedure is followed for deriving a set of <operation diagram>s.

The derived <operation definition>s and <operation diagram>s are considered legal even though a <string name> is not allowed as an <operation name> in the concrete syntax.

The derived <operation definition>s are added to <operation definitions> (if any) in the same <data type definition>. The derived <operation diagram>s are added to the list of diagrams where the original <operation definition> had occurred.

If an <operation definition> or <operation diagram> contains one or more <spelling term>s, each <spelling term> is replaced with a `Charstring` literal (see clause 14.4).

If, during the above transformation, the <operation name> in the <spelling term> had been replaced by an <operation name>, the <spelling term> is shorthand for a `Charstring` derived from the <operation name>. The `Charstring` contains the spelling of the <operation name>.

If, during the above transformation, the <operation name> in the <spelling term> had been replaced by a <string name>, the <spelling term> is shorthand for a `Charstring` derived from the <string name>. The `Charstring` contains the spelling of the <string name>.

Annex B

Specification of the set of text encoding rules

(This annex forms an integral part of this Recommendation.)

The data types are presented below with data types of the **package** `Predefined` in the order they occur, followed by other data types.

In this annex characters that appear in encoded text are identified by the names (in uppercase letters) they are given in [ISO/IEC 10646]. For example: LATIN CAPITAL LETTER T identifies the character in [ISO/IEC 10646] used to encode the `Boolean` value `true`.

B.1 Boolean

`Boolean` values, False and True shall be encoded as LATIN CAPITAL LETTER F and LATIN CAPITAL LETTER T respectively.

Example

```
dcl Var_Boolean Boolean;
task Var_Boolean := true;
```

Generated `CharString`: T

B.2 Character

`Character` values shall be encoded as the actual Character with the exception of the `ESC` character, which is encoded as two ESCAPE characters. An undefined or missing Character value shall be encoded as an ESCAPE character followed by the NULL character.

NOTE – The Generated `CharString` is allowed to contain non-printing characters.

Example

```
dcl Var_Character Character;
task Var_Character := 'M';
```

Generated `CharString`: M

B.3 String

`String` has a parameter for the item sort and shall be encoded as a list of values of the item sort enclosed in a LEFT CURLY BRACKET and a RIGHT CURLY BRACKET separated by COMMA characters.

Example

```
value type IntString inherits String <Integer>;
dcl str IntString;

task str:= ''//mkstring(6)//mkstring(9)//mkstring(1948);
```

Generated `CharString`: {6,9,1948}

B.4 Charstring, IA5String, NumericString, PrintableString, VisibleString

Although `Charstring` is based on `String`, because it is a predefined data type and is commonly used, it is given a special encoding. `IA5String` and `Charstring` cover the same range of characters and have the same encoding. `NumericString`, `PrintableString` and `VisibleString` are subsets of `IA5String`.

These string types shall be encoded as a string of characters enclosed in APOSTROPHE (') characters without change except the <apostrophe> (') which shall be encoded as two APOSTROPHE (') characters.

NOTE 1 – Within a character string <comma>s, <left curly bracket>s and <right curly bracket>s are treated as normal characters.

NOTE 2 – The Generated `Charstring` therefore possibly contains non-printing characters including end of file or end of record characters.

Example

```
dcl Var_Charstring Charstring;

task Var_Charstring := 'Fred''s world;
```

Generated `Charstring`: 'Fred's world'

B.5 Integer

`Integer` shall be encoded as a decimal integer notation without leading DIGIT ZERO characters where negative numbers are immediately preceded by a HYPHEN-MINUS without leading DIGIT ZERO characters. Zero shall never be treated as a negative number.

Example

```
dcl i Integer;

task i := 2-7;
```

Generated `Charstring`: –5

B.6 Natural

`Natural` is a **syntype** of `Integer` and shall therefore have the same encoding as `Integer`.

B.7 Real

The value 0.0 shall be encoded as DIGIT ZERO followed by FULL STOP followed by DIGIT ZERO.

Any negative value shall be encoded as a HYPHEN-MINUS immediately followed by the encoding of the value negated (a positive Real value).

A positive Real value shall be encoded as a single digit in the range 1 to 9, followed by a FULL STOP, followed by at least one and up to 11 decimal fraction digits, followed by LATIN SMALL LETTER E 'e', followed by an exponent. The exponent is the Integer encoding of the exponent value: the power to the base 10 to apply to the first part of the number. A negative exponent is allowed.

It is allowed to omit trailing fractional zero digits. It is not possible to precisely encode some Real values, and it is possible that unequal Real values that have very small difference have the same encoding. In any case, whether a Real value is precisely correct within an application will depend on how Real has been implemented. For example, if the Real value 2.0/7.0 is actually stored as the ratio of two integers (2 and 7), this is absolutely precise, whereas a more conventional encoding could be 0.2857142857, which is only correct to 10 decimal places.

Examples

```
dcl p, q, r1, r2 Real := 2000.0, 7.0, 0, 0;

task r1:= p/q;
task r2:= q/p;
```

Generated `Charstring` for r1: 2.85714285714e2

Generated `Charstring` for r2: 3.5e-3

B.8 Array

`Array` has two parameters: an index sort and a component sort. Either sort in principle is any sort, but usually the index sort has a finite number of values.

Where the index sort has a finite number of ordered values an `Array` shall be encoded as a list of element encoding values, one for each element, separated by COMMA characters within a single pair of LEFT CURLY BRACKET and RIGHT CURLY BRACKET characters.

Example

```
value type ABC {literals A, B, C};
value type A1 inherits Array <ABC, Integer>;
dcl avalue, bvalue A1;

task avalue:= (. 3 .);
task bvalue[A]:= 3;
task bvalue[B]:= 5;
task bvalue[C]:= 7;
```

Generated `Charstring` for avalue:{3,3,3}

Generated `Charstring` for bvalue:{3,5,7}

It is possible the index is not ordered (that is, the "<" operator is not defined, or for example the index sort is a **structure** or **choice**). It is also possible the index sort does not have a finite number of values (that is, no possible number of values is infinite for example `Charstring` or `Real`). For unordered or infinite index sort, the `Array` shall be encoded as the most frequent value followed by pairs of values for each element. If two or more values occur with the highest frequency one of these is chosen on an arbitrary basis as the most frequent value. Each pair shall be between a LEFT CURLY BRACKET and a RIGHT CURLY BRACKET and separated by a COMMA: the first value is an index value and the second value is element value. No index values in the pairs shall be repeated.

Example

```
value type Dehashing inherits Array <Charstring, Charstring> := (.''.);
/* note that the default value is an empty string */
dcl hashtable Dehashing;

task htable ('ac'):= 'action';
task htable ('ab'):= 'ability';
task htable ('zzzz'):= 'end of document';
```

Generated `Charstring` for `htable`: {",{'ab','ability'},{'ac','action'},{'zzzz','end of document'}}

B.9 Vector

Vector is a special case of Array that always has index sort that is a subset of Natural with a range from 1 to a specified maximum value and any item sort. Vector shall therefore be encoded in the same way as an Array with a finite number of ordered index values: that is a list of element encoding values, one for each element, separated by COMMA characters within a single LEFT CURLY BRACKET and RIGHT CURLY BRACKET pair.

B.10 Powerset

A `Powerset` of a sort represents a mathematical set whose elements are all members of the sort. Where the sort has a finite number of ordered values, the `Powerset` of the sort shall be encoded as a bit string (a string of DIGIT ZERO and DIGIT 1 characters) enclosed in APOSTROPHE (') characters. Each bit position in this bit string represents if a value of the sort is present or absent in the set. A DIGIT 1 represents that the value is present and a DIGIT ZERO that the value is absent. The leftmost bit represents the smallest value of the element sort, and each other bit represents a value of the element sort larger than values for the bits to the left.

Example

```
value type Shortalpha {literals a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v};
/* note Shortalpha has 22 values */
value type Psa inherits Powerset<Shortalpha>;
dcl letters_used Psa;

task letters_used := (. h, c, u, r .);
```

Generated `Charstring`: '00100001000000000010010'

Where the sort does not have a finite number of ordered values, the `Powerset` of the sort shall be encoded as the encoding of each element value present separated by COMMA characters enclosed in a LEFT CURLY BRACKET and RIGHT CURLY BRACKET pair. The element values are allowed to be in any order.

Example

```
value type Pchrstr inherits Powerset<Charstring>;
dcl strings_used Pchrstr;

task strings_used := (. 'me', 'you', 'us', 'me', 'again', 'hey', 'you' .);
```

Generated `Charstring`: {'again','hey','you','me','us'}

B.11 Duration

`Duration` is used to denote 'a time interval'. Unless specified otherwise, the default value of the unit of `Duration` is one second.

Duration values shall be encoded as a pair of integers separated by a COMMA enclosed in a LEFT CURLY BRACKET and RIGHT CURLY BRACKET pair: the first integer gives the number of units and the second integer any fractional part in nano (10^{-9}) units. By default these are seconds and nano-seconds. A negative value is allowed, in which case the encoding of the negated value shall be used with a HYPHEN-MINUS inserted immediately before the first integer.

Example

```
dcl dvar Duration;

task dvar := -17.00000007;
```

Generated `Charstring`: {-17,700}

B.12 Time

`Time` is used to denote 'a point in time'. The value of a unit of `Time` shall be the same as the value of the unit of `Duration`. The origin of `Time` units is not specified by this Recommendation, but corresponds to the system clock being at zero: that is, NOW gives the value 0.0. It is allowed for `Time` values to be negative.

Time values shall be encoded in the same way as Duration.

Example

```
dcl tvar Time;

task tvar:= 17.0000017;
```

Generated `Charstring`: {17,17000}

B.13 Bag

`Bag` shall be encoded in the same way as a `Powerset` with an element sort that does not have a finite ordered set of values, but with each element value preceded by an integer followed by COLON. The integer is the number of times the element sort occurs in the `Bag` value.

Example

```
value type B1 inherits Bag <Integer>;

dcl Var_Bag B1;

task Var_Bag := (. 7, 4, 7 .);
```

Generated `Charstring`: {2:7,1:4}

B.14 Bit, Bitstring

`Bit` shall be encoded as a single DIGIT ZERO or DIGIT 1 representing a zero and one bit respectively.

Example

```
dcl Var_Bit Bit;

task Var_Bit := 1;
```

Generated `Charstring`: 1

`Bitstring` shall be encoded as sequence of bits represented by DIGIT ZERO and DIGIT 1 characters enclosed in APOSTROPHE (') characters.

Example

```
dcl Var_Bit Bit_String;

task Var_Bit := '01011'B;
```

Generated `Charstring`: '01011'

B.15 Octet, Octetstring

`Octet` shall be encoded as two characters corresponding to the hexadecimal notation of the `Octet`. The alphabetic characters shall be represented by lowercase letters (that is, LATIN SMALL LETER A to LATIN SMALL LETTER F).

Example

```
dcl oct Octet;

task oct := 62;
```

Generated `Charstring`: 3e

`Octetstring` is a sequence of `Octet` values and shall be encoded as a string of hexadecimal character pairs enclosed in APOSTROPHE (') characters. The alphabetic characters shall be represented by lowercase letters.

Example

```
dcl os Octetstring;

task os:= '12B32D'H;
```

Generated `Charstring`: '12b32d'

B.16 Pid, pid sorts

To allow flexibility between applications pid values shall be encoded as a CHOICE value corresponding to the choice definition:

```
{ choice
      0 ApplicationDefined;
      1 Integer;
      2 OctetString;
      3 BitString;
      4 Charstring;
      5 { struct
           identity Charstring;
           instance Natural;
        };
}
```

A value of a pid sort shall be encoded as the corresponding value of the `Pid` sort.

All pid values shall be encoded using the same choice field as defined above: that is, if one pid value in a model is encoded using the choice 1 Integer, all other pid values in the same model shall be encoded using choice 1 Integer. Otherwise, which of the above choices is selected is application dependent. The first choice allows an application defined encoding to be used. The `ApplicationDefined` sort is not defined by this Recommendation.

The same agent instance of a model shall always have the same pid value encoding. Two different agent instances shall always have different encoded pid values. Otherwise the derivation of the encoded pid values is not further defined by the Recommendation.

Example (using choice 5)

```
dcl agent_id Pid; /* in second instance of process IPS */

task agent_id := self;
```

Generated `Charstring`: {5, {IPS, 2}}

B.17 Null

The value Null (see [ITU-T Z.105]) shall be encoded as a DIGIT ZERO character.

Example

```
dcl Var_Null Null;

task Var_Null := Null;
```

Generated `Charstring`: 0

B.18 Enumerated (literal list)

An enumerated type defined by a literal list or as an ASN.1 ENUMERATED type shall be encoded as the Natural value given by the application of `num` operator of the data type to the literal representing the value to be encoded.

Example

```
value type Enum
{ literals e1, e2, e3;
};

dcl evar Enum;

task evar := e2;
```

Generated `Charstring`: 1

B.19 Structures

A structure data type value shall be encoded as a list of values for the fields within a LEFT CURLY BRACKET and RIGHT CURLY BRACKET pair separated by COMMA characters.

Example

```
value type Record { struct
f1 Integer;
f2 Charstring;
f3 Integer;};
dcl locat Record;
task locat:= (. 17, 'mid-field', 230125 .);
```

Generated `Charstring`: {17, 'mid-field', 230125}

B.20 Choice

A choice value shall be encoded as a pair of values separated by a COMMA enclosed in a LEFT CURLY BRACKET and RIGHT CURLY BRACKET pair. The first value is the selected field indicated by name of the choice. The second value is the `Charstring` for the field value according to the type of the field.

Example

```
value type C {choice
cs Charstring;
cb Boolean;
};

dcl ChoiceVar C;

task ChoiceVar.cb := true;
```

Generated `Charstring`: {cb,T}

B.21 Inherits and syntype

A **syntype** defines a new name for a data type with an optional constraint on the set values. The text encoding is therefore identical to that of the data type.

A data type that inherits from another data type has the same encoding as the parent data type, though in the case that additional literal or fields are added, these are added to the encoding.

Annex C

Language binding

(This annex forms an integral part of this Recommendation.)

This annex defines the use of the syntax of an alternative language within the syntax rules <variable definition>, <data definition>, <non terminating statement>, <terminating statement> and <expression>, which are taken as the points of syntax variation. By default an SDL-2010 specification is bound to the native syntax as defined in the main body of this Recommendation or other Recommendations for SDL-2010.

Each diagram or other definition that is allowed a <package use clause> is bound to a particular concrete syntax as defined in clause 7.2. The only allowed language <data binding> constructs are to the default binding **`package`** `Predefined`, which is the language binding for the native SDL-2010 concrete syntax, or to a package defined in this annex.

Though the syntax within the points of syntax variation looks like another language (such as C, C++, Java or some other language) the semantics are defined by SDL-2010. The binding to the SDL-2010 abstract grammar is permitted to invoke complex transformations to achieve the mapping, and constraints are allowed to exclude constructions permitted by the language concrete syntax that are not able to be reasonably mapped. A model that includes constructs that do not map to SDL-2010 does not conform to the SDL-2010 language. A tool that handles constructs that do not map to SDL-2010 abstract grammar and semantics shall provide an indication if such constructs are used.

Different languages not only vary in their syntax rules, but also have some differences in the lexical elements of the language. Therefore, within diagrams bound to the syntax of an alternative language for certain rules, it is allowed for the lexical rules of SDL-2010 to be extended to include lexemes of the alternative language. This annex also defines these variations.

To be concise, throughout this annex the phrase "SDL-2010 diagram" is used to mean a "diagram bound to the native SDL-2010 syntax for the points of syntax variation", and the phrase "SDL-2010 diagrams" to mean more than one such diagram.

C.1 C Language binding

This clause allows diagrams to be bound to a subset of [b-ISO/IEC 9899] by binding each such diagram to the **`package`** `C_Predefined` defined at the end of this clause.

To be concise, throughout this clause the phrase "C diagram" is used to mean a "diagram bound to a subset of [b-ISO/IEC 9899] for the points of syntax variation", and the phrase "C diagrams" to mean more than one such diagram.

C.1.1 Extensions to lexical rules

Lexical rules define lexical units. Lexical units are terminal symbols of the *Concrete grammar*.

<lexical unit> ::=
<name>
| <integer name>
| <real name>
| <character string>
| <hex string>
| <bit string>
| <note>
| <comment body>
| <composite special>
| <special>

| <semicolon>
| <other character>
| <quoted operation name>
| <c string literal>
| <keyword>

The rule <lexical unit> is extended to include <c string literal>.

NOTE – No qualifier is allowed for an identifier in the C language syntax, therefore identifiers and names have the same syntax. The universal character name (of the form **\u** hex-quad or **\U** hex-quad hex-quad) are not allowed, therefore names in the C language have the same syntax as names in SDL-2010.

C.1.1.1 C keywords

The following SDL-2010 lowercase <name> items are also keywords in C diagrams:

auto	**case**	**char**
const	**do**	**double**
enum	**extern**	**float**
for	**goto**	**inline**
int	**long**	**register**
restrict	**short**	**signed**
sizeof	**static**	**switch**
typedef	**union**	**unsigned**
void	**volatile**	**while**

These are keywords only in C diagrams. In SDL-2010 diagrams each of these is treated as a <name>. In accordance with C and to limit the impact on SDL-2010 of C lexical rules, corresponding uppercase keywords do not exist. An SDL-2010 <keyword> item as defined in [ITU-T Z.101] cannot be used as a <name> in C type or variable definitions.

C.1.1.2 C integer constants

<integer name> ::=
<decimal digit>+
| <c integer constant>

The lexical rule <integer name> is extended to include <c integer constant>. The form <decimal digit>+ applies to SDL-2010 diagrams, and the form <c integer constant> applies to C diagrams.

<c integer constant> ::=
<c decimal constant>
| <c octal constant>
| <c hexadecimal constant>

NOTE 1 – Clause 6.4.4.1 of [b-ISO/IEC 9899] but excluding the <c integer suffix>.

<c decimal constant> ::=
<c nonzero digit>
| <c decimal constant> <c digit>

NOTE 2 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c octal constant> ::=
0
| <c octal constant> <c octal digit>

NOTE 3 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c hexadecimal constant> ::=
<c hexadecimal prefix> <c hexadecimal digit>
| <c hexadecimal constant> <c hexadecimal digit>

NOTE 4 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c hexadecimal prefix> ::=
0x

NOTE 5 – Clause 6.4.4.1 of [b-ISO/IEC 9899] without the **0X** form.

<c nonzero digit> ::=
1 | **2** | **3** | **4** | **5**
| **6** | **7** | **8** | **9**

NOTE 6 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c octal digit> ::=
0 | **1** | **2** | **3** | **4** | **5** | **6** | **7**

NOTE 7 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

<c hexadecimal digit> ::=
0 | **1** | **2** | **3** | **4**
| **5** | **6** | **7** | **8** | **9**
| **a** | **b** | **c** | **d** | **e** | **f**
| **A** | **B** | **C** | **D** | **E** | **F**

NOTE 8 – Clause 6.4.4.1 of [b-ISO/IEC 9899].

A <c integer constant> has an unbounded integer value determined according to the rules given in [b-ISO/IEC 9899]. The *Name* of the <c integer constant> is the *Name* of the <decimal digit> string starting with a non-zero digit that has this value. The *Qualifier* for the *Identifier* of a <c integer constant> is the *Qualifier for* <<**package** Predefined **type** Integer>>.

C.1.1.3 C composite specials

<composite special> ::=
<result sign>
| <range sign>
| <composite begin sign>
| <composite end sign>
| <concatenation sign>
| <history dash sign>
| <greater than or equals sign>
| <implies sign>
| <is assigned sign>
| <less than or equals sign>
| <not equals sign>
| <qualifier begin sign>
| <qualifier end sign>
| <c increment operator>
| <c decrement operator>
| <c equality sign>
| <c inequality sign>
| <c logical and sign>
| <c logical or sign>
| <c multiplication assignment sign>
| <c division assignment sign>
| <c remainder assignment sign>
| <c addition assignment sign>
| <c subtraction assignment sign>
| <c shift left assignment sign>
| <c shift right assignment sign>
| <c bitwise and assignment sign>
| <c bitwise excl or assignment sign>
| <c bitwise incl or assignment sign>

The rule <composite special> is extended to include the additional composite symbols used in C except those symbols only used for pre-processing and alternatives for: square brackets, curly brackets and the number sign.

<c pointer operator> ::=
<result sign>

<c shift left sign> ::=
<qualifier begin sign>

<c shift right sign> ::=
<qualifier end sign>

The rule <c pointer operator>, <c shift left sign> and <c shift right sign> lexical units are the same as <result sign>, <qualifier begin sign> and <qualifier end sign>, respectively, and the lexical unit represented is determined by context. In syntax rules defined for C diagrams these lexical units always represent <c pointer operator>, <c shift left sign> and <c shift right sign>, respectively.

<c increment operator> ::=
<plus sign> <plus sign>

<c decrement operator> ::=
<hyphen> <hyphen>

<c equality sign> ::=
<equals sign> <equals sign>

<c inequality sign> ::=
<exclamation mark> <equals sign>

<c logical and sign> ::=
<ampersand> <ampersand>

<c logical or sign> ::=
<vertical line> <vertical line>

<c multiplication assignment sign> ::=
<asterisk> <equals sign>

<c division assignment sign> ::=
<solidus> <equals sign>

<c remainder assignment sign> ::=
<percent sign> <equals sign>

<c addition assignment sign> ::=
<plus sign> <equals sign>

<c subtraction assignment sign> ::=
<hyphen> <equals sign>

<c shift left assignment sign> ::=
<less than sign> <less than sign> <equals sign>

<c shift right assignment sign> ::=
<greater than sign> <greater than sign> <equals sign>

<c bitwise and assignment sign> ::=
<ampersand> <equals sign>

<c bitwise excl or assignment sign> ::=
<circumflex accent> <equals sign>

<c bitwise incl or assignment sign> ::=
<vertical line> <equals sign>

C.1.1.4 C character constants

<c character constant> ::=
<apostrophe> <c char sequence> <apostrophe>

NOTE 1 – Clause 6.4.4.4 of [b-ISO/IEC 9899] without the prefixes for types other than **unsigned char**.

<c char sequence> ::=
<c char>

NOTE 2 – Clause 6.4.4.4 of [b-ISO/IEC 9899] restricted to a single character, because the "value of an integer character constant containing more than one character, or containing a character or escape sequence that does not map to a single-byte execution character, is implementation-defined". (See clause 6.4.4.4 of [b-ISO/IEC 9899].)

<c char> ::=
<quotation mark> | <c other character> | <space> | <c escape sequence>

NOTE 3 – Clause 6.4.4.4 of [b-ISO/IEC 9899], modified to use SDL-2010 lexical items.

A <c escape sequence> (see clause C.1.1.5 below) shall map to a single byte character.

The <c character constant> represents the constant **unsigned char** value corresponding to the constant expression `to_Unsigned_char(<<package Predefined type Char>> num(c))`, where c is the character specified by the <c character constant>.

C.1.1.5 C string literals

<c string literal> ::=
<quotation mark> [<c s char sequence>] <quotation mark>

There is a clash between <quoted operation name> and <c string literal>. A lexical unit in an SDL-2010 diagram that starts with a <quotation mark> never represents a <c string literal>. A lexical unit in a C diagram that starts with a <quotation mark> always represents a <c string literal>.

A <c string literal> is pointer to the first element of an array of characters derived from the <character string> where the <c string literal> has a <c s char sequence> the same as the sequence of characters in a <character string> (after replacement of <c escape character> items in the <c s char sequence> and <apostrophe> pairs in the <character string>).

A <c string literal> represents a *Variable-access* for an implicit anonymous **unsigned char** pointer (to first **char** of an implicit **char** array) defined by

```
synonym anoncp Star_Unsigned_char = "&"(xstring[0]);
```

where

`xstring` is an implicit anonymous **char** array defined by

```
synonym xstring value inherits Cvector <char,n+1> = c_array_init;
```

where

`n` is the length of the <character string> (that is, the length in bytes of the <c s char sequence> after replacement of <c escape character> items), and

`c_array_init` is the initialization value using a call of `Make` and calls of `Modify` (as described for <c initialize> in clause C.1.3) with each element of the array initialized to the corresponding character of the string and the string terminates in a zero (a `NUL` character).

For example, if the <c string literal> is `"AB3"`, then `c_array_init` is `{'A', 'B', '3', endstr}` where `endstr` is `Integer_to_Unsigned_char(num(<<package Predefined type Character>>NUL))`.

The **unsigned char** array element `i` (<`n`) for the string is `Integer_to_Unsigned_char(num(zstring[i+1]))`, where `zstring` is the `Charstring` <character string>.

NOTE 1 – This definition allows a <c string literal> to be used with functions from the C library `string.h` (see clause 7.24 of [b-ISO/IEC 9899]), if these functions are defined in an additional package as external operations or procedures. The operator

```
<<package C_Predefined type Star_Unsigned_char>> Star_Unsigned_char_to_Charstring
```

allows a <c string literal> (or other `Unsigned_char` array values) to be converted to `Charstring` expressions.

<c s char sequence> ::=
<c s char>
| <c s char sequence> <c s char>

NOTE 2 – Clause 6.4.5 of [b-ISO/IEC 9899].

<c s char> ::=
<apostrophe> | <c other character> | <space> | <c escape sequence>

NOTE 3 – Clause 6.4.5 of [b-ISO/IEC 9899], modified to use SDL-2010 lexical items.

<c other character> ::=
<alphanumeric>
| <special> | <dollar sign> | <percent sign>
| <ampersand> | <question mark> | <commercial at>
| <circumflex accent> | <underline> | <grave accent>
| <vertical line> | <tilde>

<c escape sequence> ::=
<c simple escape sequence>
| <c octal escape sequence>
| <c hexadecimal escape sequence>

NOTE 4 – Clause 6.4.4.4 of [b-ISO/IEC 9899] excluding <c universal character name>.

<c simple escape sequence> ::=
<reverse solidus> <apostrophe>
| <reverse solidus> <quotation mark>
| <reverse solidus> <question mark>
| <reverse solidus> <reverse solidus>
| <reverse solidus> **a**
| <reverse solidus> **b**
| <reverse solidus> **f**
| <reverse solidus> **n**
| <reverse solidus> **r**
| <reverse solidus> **t**
| <reverse solidus> **v**

NOTE 5 – Clause 6.4.4.4 of [b-ISO/IEC 9899].

<c octal escape sequence> ::=
<reverse solidus> <c octal digit> <c octal digit> <c octal digit>

NOTE 6 – Clause 6.4.4.4 of [b-ISO/IEC 9899].

<c hexadecimal escape sequence> ::=
<reverse solidus> **x** <c hexadecimal digit> <c hexadecimal digit>

NOTE 7 – Clause 6.4.4.4 of [b-ISO/IEC 9899] but always with two hexadecimal digits.

The characters represented by the alternatives of <c escape sequence> are defined by [b-ISO/IEC 9899].

C.1.1.6 C identifiers, name resolution and visibility

Concrete grammar

<c identifier> ::=
<name>

NOTE – Clause 6.4.2.1 of [b-ISO/IEC 9899] is redefined as <name> because <c identifier> has the same syntax as <name>.

In a C diagram, a <c identifier> cannot have a <qualifier>. Identifiers in C language data definitions and expressions are in principle bound to names according to the SDL-2010 name binding rules in clause 6.6 of [ITU-T Z.101]. In a defining context for a *Name* in the abstract grammar, a <c identifier> represents the *Name*. In a non-defining context where a name is used, a <c identifier> represents an *Identifier* and the *Qualifier* is derived from the name binding. A <c identifier> that represents an *Identifier* may be marked as belonging to a subcategory: for example, <data type c identifier> requires the *Identifier* to represent a data type. When a <c identifier> represents an *Identifier*, if it cannot be uniquely bound to an *Identifier* it is ambiguous and therefore invalid.

C.1.1.7 Macros and pre-processing

It is assumed that pre-processing as defined by [b-ISO/IEC 9899] is applied before any C diagram is considered as part of an SDL-2010 model, and as a consequence that any C pre-processing tokens

or alternative composite symbols (such as the alternative left curly bracket) are removed from the contents of a C diagram. It is assumed that SDL-2010 macro processing is applied to the model after any C pre-processing.

C.1.2 Data type definition

Concrete grammar

<data definition> ::=
<entity in data type>
| <interface definition>
| <c type definition>

A <c type definition> shall only be used in a C diagram.

<c type definition> ::=
typedef <c type specifier> <c declarator> <semicolon>
| <c struct or union specifier> <semicolon>
| <c enum specifier> <semicolon>

NOTE 1 – This is a valid subset of <c declaration> from clause 6.7 of [b-ISO/IEC 9899]. This subset includes only alternatives starting with the keywords **typedef**, **struct**, **union** and **enum**. The <c init declarator list> is simplified to a <c declarator> and is not allowed to be omitted.

<c declarator> ::=
[<c pointer>] <c direct declarator>

NOTE 2 – Clause 6.7.6 of [b-ISO/IEC 9899].

If the <c type qualifier list> of a <c pointer> of a <c declarator> contains a <c type qualifier> that is **const**, it is allowed to modify the contents of the target of the pointer, but the pointer itself should always point to the same target.

The <c identifier> of a <c declarator> is the <c identifier> of the <c direct declarator> of the <c declarator>, or if the <c direct declarator> of the <c declarator> contains a <c declarator> enclosed in <left parenthesis> <right parenthesis>, the <c identifier> of this enclosed <c declarator>.

<c direct declarator> ::=
<c identifier>
{ <left square bracket> <integer c constant expression> <right square bracket> }*
| <left parenthesis> <c declarator> <right parenthesis>

NOTE 3 – Clause 6.7.6 of [b-ISO/IEC 9899] simplified so that it either starts with a <c identifier> or is a <c declarator> in parentheses, with the assignment expression constrained to a <c constant expression> without type qualifiers or **static**.

<c pointer> ::=
<asterisk> [<c type qualifier list>]
| <asterisk> [<c type qualifier list>] <c pointer>

NOTE 4 – Clause 6.7.6 of [b-ISO/IEC 9899].

<c type qualifier list> ::=
<c type qualifier>
| <c type qualifier list> <c type qualifier>

NOTE 5 – Clause 6.7.6 of [b-ISO/IEC 9899].

A <c type definition> starting with **typedef** represents a *Syntype-definition*. The <c identifier> of the <c declarator> of the <c type definition> represents the *Syntype-name* of the *Syntype-definition*. The *Range-condition* of the *Syntype-definition* is the predefined Boolean value `true`. There is no *Default-initialization* for the *Syntype-definition*. The *Parent-sort-identifier* of the *Syntype-definition* of the <c type definition> is denoted `PS` below and is determined as follows.

a) If the <c declarator> has a <c pointer>, its type is a typed pointer and `PS` is the *Identifier* of this data type. If the <c pointer> does not include another <c pointer> and the <c type specifier> is the <c type keywords> item **void**, `PS` is the *Identifier* for `Star_void`. Otherwise, if it not `Star_void` and there is a (possibly anonymous) visible *Data-type-definition* for a subtype of `Star_type` where `Atype` of `Star_type` is bound to the type (`target`) of a reduced <c declarator> (that is, the <c declarator> with removal of the <asterisk> and any adjacent <c type qualifier list> from the <c pointer>), `PS` is the *Identifier* of this data type. Otherwise (if it is not `Star_void` and no such type is visible), the <c type definition> represents an anonymous *Data-type-definition* that is a *Value-data-type-definition* in addition to the *Syntype-definition* represented by the <c type definition>, and this anonymous type is the sort defined by the concrete SDL-2010 **`value type`** `PS` **`inherits`** `Star_type < target >;`

where

`target` is the target type;

`Star_type` is defined as a parameterized data type in `C_Predefined`.

If the reduced <c declarator> contains a <c pointer>, `target` is a typed pointer to a further typed pointer with a type for either a visible or an anonymous *Data-type-definition* determined in the same way described above. For example, for

```
typedef Existing_type ***Ppp_existing_type;
```

defines a syntype *Name* `Ppp_existing_type` and *Parent-sort-identifier* identifying the anonymous type defined by

```
value type PS inherits
    Star_type<value inherits
        Star_type<value inherits Star_type<Existing_type>>>;
```

and intermediate anonymous types defined by

`value inherits` `Star_type<value` **`inherits`** `Star_type<Existing_type>>` and **`value inherits`** `Star_type<Existing_type>` using inline sort definitions for the two outer `Star_type` parameters.

Otherwise, the reduced <c declarator> does not contain a <c pointer> and `target` is the type identified for the `PS` of the reduced <c declarator> (that is, the type of a <c declarator> with no <c pointer>) as determined in b), c), d) and e) below.

b) If there is no <c pointer> and the <c identifier> in the <c declarator> of the <c type definition> is followed by one or more index specifications (<left square bracket> [<integer c constant expression>] <right square bracket> list in <c direct declarator>), the type is a C array. The C array has a derived anonymous *Data-type-definition* that is a *Value-data-type-definition* in addition to the *Syntype-definition* represented by the <c type definition>. The `PS` is the *Identifier* of this anonymous value type defined by

```
value type PS inherits Cvector <itemsort, arraysize>;
```

where

`arraysize` is the value of the <Integer c constant expression> for this index specification,

`itemsort` is sort of the following index specification if there is one, or the sort identified for PS from the <c type specifier> after the **typedef** as in (c), (d) and (e) below, and

`Cvector` is defined as a parameterized data type in `C_Predefined`.

If there is a following index specification, for the `itemsort` there is a further derived anonymous *Data-type-definition* that is a *Value-data-type-definition* in addition to the *Syntype-definition* represented by the <c type definition>. This is formed in the same way

as the *Data-type-definition* for a previous index specification, so that there are as many anonymous data types as index specifications. For example, for

```
typedef int x3d[3][5][7]
```

PS is defined by

```
value type PS inherits
   Cvector<value inherits Cvector<value inherits Cvector<int,7>,5>,3>
```

using inline sort definitions for the two outer `itemsort` parameters.

c) If there is no <c pointer> and no index specification after the <c identifier> in the <c declarator> of the <c type definition>, and the <c type specifier> is <c type keywords>, PS is the *Identifier* for the <c type keywords>. The identified type shall not be `Void`.

d) Otherwise, if there is no <c pointer> and no index specification after the <c identifier> in the <c declarator> of the <c type definition>, and the <c type specifier> is a <c identifier>, it shall identify a data type and PS is the *Identifier* of that data type.

e) Otherwise, if there is no <c pointer> and no index specification after the <c identifier> in the <c declarator> of the <c type definition>, and the <c type specifier> is <c struct or union specifier> or <c enum specifier>, PS is the (possibly anonymous) *Identifier* of the data type of the <c type specifier> that is the <c struct or union specifier> or <c enum specifier>. The <c type specifier> that is the <c struct or union specifier> or <c enum specifier> represents a *Data-type-definition* in addition to the *Syntype-definition* represented by the <c type definition>.

C.1.2.1 Data type specifier

Concrete grammar

<c type specifier> ::=
- <c type keywords>
- | <c struct or union specifier>
- | <c enum specifier>
- | <data type c identifier>

<c type keywords> ::=
- **void**
- | [**signed** | **unsigned**] **char**
- | [**signed** | **unsigned**] **short** [**int**]
- | [**signed** | **unsigned**] **int**
- | [**signed** | **unsigned**] **long** [**long**] [**int**]
- | **signed**
- | **unsigned**
- | **float**
- | [**long**] **double**

NOTE – This is a valid subset of <c type specifier> from clause 6.7.2 of [b-ISO/IEC 9899], modified to syntactically restrict the allowed lists of keywords used for data types and also removing **_Bool**, **_Complex** and <c atomic type specifier>.

If the <c type specifier> is <c type keywords>, the keyword list represents the *Sort* for the identified data type of <<**package** C_Predefined>> as follows:

void	identifies <<**package** C_Predefined>> Void
char	identifies <<**package** C_Predefined>> Signed_char
signed char	identifies <<**package** C_Predefined>> Unsigned_char
unsigned char	identifies <<**package** C_Predefined>> Unsigned_char
short	identifies <<**package** C_Predefined>> Signed_short
short int	identifies <<**package** C_Predefined>> Signed_short
signed short	identifies <<**package** C_Predefined>> Signed_short

signed short int	identifies <<**package** C_Predefined>> Signed_short
unsigned short	identifies <<**package** C_Predefined>> Unsigned_short
unsigned short int	identifies <<**package** C_Predefined>> Unsigned_short
int	identifies <<**package** C_Predefined>> Signed_int
signed	identifies <<**package** C_Predefined>> Signed_int
signed int	identifies <<**package** C_Predefined>> Signed_int
unsigned	identifies <<**package** C_Predefined>> Unsigned_int
unsigned int	identifies <<**package** C_Predefined>> Unsigned_int
long	identifies <<**package** C_Predefined>> Signed_long
long int	identifies <<**package** C_Predefined>> Signed_long
signed long	identifies <<**package** C_Predefined>> Signed_long
signed long int	identifies <<**package** C_Predefined>> Signed_long
unsigned long	identifies <<**package** C_Predefined>> Unsigned_long
unsigned long int	identifies <<**package** C_Predefined>> Unsigned_long
long long	identifies <<**package** C_Predefined>> Signed_long_long
long long int	identifies <<**package** C_Predefined>> Signed_long_long
signed long long	identifies <<**package** C_Predefined>> Signed_long_long
signed long long int	identifies <<**package** C_Predefined>> Signed_long_long
unsigned long long	identifies <<**package** C_Predefined>> Unsigned_long_long
unsigned long long int	identifies <<**package** C_Predefined>> Unsigned_long_long
float	identifies <<**package** C_Predefined>> Float
double	identifies <<**package** C_Predefined>> Double
long double	identifies <<**package** C_Predefined>> Long_double

C.1.2.2 Struct or union specifier

Concrete grammar

<c struct or union specifier>::
<c struct or union> [<c identifier>]
<left curly bracket> <c struct declaration list> <right curly bracket>
| <c struct or union> <struct or union data type c identifier>

NOTE 1 – Clause 6.7.2.1 of [b-ISO/IEC 9899].

<c struct or union> ::=
struct
| **union**

NOTE 2 – Clause 6.7.2.1 of [b-ISO/IEC 9899].

A <c struct or union specifier> that has a <c struct declaration list> represents a *Data-type-definition* that is a *Value-data-type-definition*. The <c identifier> before the <left curly bracket> represents a *Name* that is the *Sort* of the *Value-data-type-definition*. If the <c identifier> is omitted, the *Value-data-type-definition* has an anonymous unique *Name*. The optional *Data-type-identifier* of the *Value-data-type-definition* is omitted. The *Literal-signature-**set***, *Procedure-definition-**set***, *Data-type-definition-**set*** and *Syntype-definition-**set*** of the *Value-data-type-definition* are empty. The *Static-operation-signature-**set*** is derived from <c struct declaration list>.

A <c struct or union specifier> that does not have a <c struct declaration list> represents an *Identifier*. If <c struct or union> is **struct**, the following <struct or union data type c identifier> shall represent an *Identifier* that identifies a *Value-data-type-definition* for a structure. If <c struct or union> is **union**, the following <struct or union data type c identifier> shall represent an *Identifier* that identifies a *Value-data-type-definition* for a union (a choice in SDL-2010 terminology).

<c struct declaration list> ::=
<c struct declaration>
| <c struct declaration list> <c struct declaration>

NOTE 3 – Clause 6.7.2.1 of [b-ISO/IEC 9899].

<c struct declaration> ::=
<c specifier qualifier list> [<c struct declarator list>] <semicolon>

NOTE 4 – Clause 6.7.2.1 of [b-ISO/IEC 9899] without the static-assert-declaration alternative.

<c specifier qualifier list> ::=
<c type specifier>

NOTE 5 – Clause 6.7.2.1 of [b-ISO/IEC 9899] without <c type qualifier> items.

<c struct declarator list> ::=
<c struct declarator>
| <c struct declarator list> <comma> <c struct declarator>

NOTE 6 – Clause 6.7.2.1 of [b-ISO/IEC 9899].

<c struct declarator> ::=
<c declarator>
| <c declarator> <colon> <c constant expression>

NOTE 7 – Clause 6.7.2.1 of [b-ISO/IEC 9899], modified so that the <c declarator> before a <colon> is not optional because SDL-2010 requires a field name. The <c constant expression> has no SDL-2010 meaning.

A <c struct declarator> is a field of a structure or union with a field name that is the <c identifier> of the <c declarator> of the <c struct declarator>. Each field name of a structure or union (<c identifier> of the <c declarator> of a <c struct declarator> of a <c struct declarator list> of a <c struct declaration> of a <c struct declaration list>) shall be different from every other field name of the same structure or union (the <c struct declaration list>). A <c struct declarator> that has a <c declarator> that contains a <c pointer> is a pointer field. A <c struct declarator> is an array field if it has a <c declarator> that contains one or more index specifications (<left square bracket> <Integer c constant expression> <right square bracket> list in <c direct declarator>). A <c struct declarator> has a field sort that is the *Sort* determined in the same way as determining the *Parent-sort-identifier* of the *Syntype-definition* of a <c type definition> with a **typedef** (see clause C.1.2), except the <c type specifier> is the <c type specifier> of the <c specifier qualifier list> of the <c struct declaration> that encloses the <c struct declarator list> that encloses the <c struct declarator>. As a consequence, it is possible that a <c declarator> of a <c struct declarator> that contains a <c pointer> or index specification represents additional anonymous *Data-type-definition* items. The field sort shall not be `Void`.

The <c struct declaration list> for a structure `S` represents (in the *Static-operation-signature-**set*** of the *Data-type-definition* for `S`):

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 `S` structure sort, and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

b) An *Operation-signature* for a generic operator named `Make` with a non-empty *Formal-argument* list where each item is the *Sort-reference-identifier* of the corresponding (in order) field name, and an *Operation-result* that is the *Sort-reference-identifier* of the `S` structure sort, each formal parameter of the procedure identified by the *Operation-signature* has its *Parameter-aggregation* that is **PART**, 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 SDL-2010 <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 a **PART** aggregation kind,

and the result has the same aggregation kind as the field `fn`.

NOTE 8 – A special syntax is provided as described in clause 12.2.3. To use `fnExtract` to extract the value of field `fn` from structure variable `vs` in the context of an expression the notation is:

```
vs.fn
```

d) For each field, if the field name is `fn` and the field sort is `fs`, an *Operation-signature* for the SDL-2010 <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 a **PART** 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**.

NOTE 9 – A special syntax is provided as described in clause 12.3.3.1 modified to use `"="` for the assignment sign. To use `fnModify` to assign the value `fieldValue` (a value with the sort of field `fn`) to field `fn` of structure variable `vs`, the notation is:

```
vs.fn = fieldValue;
```

The <c struct declaration list> for a union sort `U` represents (in the *Operation-signature* set of the *Data-type-definition* for `U`):

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 `U` union sort, and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

b) For a unique field sort with the sort `ufs` (if there are two or more fields with the same sort there is one unique field sort, `ufs`), an *Operation-signature* for a generic operator

```
Make ( ufs ) -> U;
```

for a generic field initialization operator for the leftmost (in the union definition) field with sort `ufs` where

`ufs` is an **in** parameter with the aggregation kind **PART**

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 SDL-2010 operation signature

```
fn ( fs ) -> U;
```

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**.

d) For each field, if the field name is `fn` and the field sort is `fs`, an *Operation-signature* for the SDL-2010 <operation signature>

`fnExtract ( U ) -> fs;`
for a generic operator where
`fnExtract` is a *field-extract-name* formed from the concatenation of the field name and `"Extract"`,
`U` is an **in/out** parameter with a **PART** aggregation kind,
and the result has the same aggregation kind as the field `fn`.

NOTE 10 – A special syntax is provided as described in clause 12.2.3. To use `fnExtract` to extract the value of field `fn` from a choice variable `vu` in the context of an expression, the notation is:

```
vu.fn
```

e) For each field, if the field name is `fn` and the field sort is `fs`, an *Operation-signature* for the SDL-2010 <operation signature>
`fnModify ( U, fs ) -> U;`
for a generic operator where
`fnModify` is a *field-modify-name* formed from the concatenation of the field name and `"Modify"`,
`U` is an **in/out** parameter with a **PART** aggregation kind,
`fs` is an **in** parameter with a **PART** aggregation kind,
and the procedure identified by the *Operation-signature* has a *Result-aggregation* that is **PART**.

NOTE 11 – A special syntax is provided as described in clause 12.3.3.1 modified to use "=" for the assignment sign. To use `fnModify` to assign the value of `fieldValue` (a value with the sort of field `fn`) to field `fn` of a choice variable `vu`, the notation is:

```
vu.fn = fieldValue;
```

f) For each field, if the field name is `fn`, an *Operation-signature* for the SDL-2010 <operation signature>
`fnPresent ( U ) -> <<`**`package`** `Predefined>>Boolean;`
for a generic operator where
`fnPresent` is a *field-present-name* formed from the concatenation of the field name and `"Present"`,
`U` 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**.
Whether `fnPresent` is visible in a C diagram is implementation dependent.

g) An *Operation-signature* for a generic operator named `PresentExtract` based on the SDL-2010 <operation signature>
`PresentExtract ( U )-> AnonPresent;`
where `AnonPresent` is defined as a literal constructor data type that uses the field names of the choice as literals as described below,
`U` 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**.
Whether `PresentExtract` is visible in a C diagram is implementation dependent.

The <c struct or union specifier> for a union type `U` also represents an additional (anonymous) *Data-type-definition*, that for the description above is called `AnonPresent`, in the context that the *Data-type-definition* for `U` occurs. This is defined with a *Literal-signature-**set*** where each field name of the union `U` represents a *Literal-signature*. The order of the literals is the same as the order in which the field names are specified left to right in the union `U`. 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.

C.1.2.3 Enum specifier

Concrete grammar

<c enum specifier> ::=
    enum [<c identifier>]
        <left curly bracket> <c enumerator list> <right curly bracket>
  | **enum** [<c identifier>]
        <left curly bracket> <c enumerator list> <comma> <right curly bracket>
  | **enum** <enum data type c identifier>

NOTE 1 – Clause 6.7.2.2 of [b-ISO/IEC 9899].

A <c enum specifier> that has a <left curly bracket> followed by a <c enumerator list> followed by a <comma> followed by a <right curly bracket> has the same meaning as a <c enum specifier> with <left curly bracket> followed by the same <c enumerator list> followed by a <right curly bracket>.

A <c enum specifier> that does not have a <c enumerator list> represents an *Identifier* that identifies a *Value-data-type-definition* for an enumerated type.

A <c enum specifier> that has a <c enumerator list> represents a *Data-type-definition* that is a *Value-data-type-definition* for an enumerated type. The <c identifier> before the <left curly bracket> represents a *Name* that is the *Sort* of the *Value-data-type-definition*. If the <c identifier> is omitted, the *Value-data-type-definition* has an anonymous unique *Name*. The optional *Data-type-identifier* of the *Value-data-type-definition* is omitted. The *Procedure-definition-**set***, *Data-type-definition-**set*** and *Syntype-definition-**set*** of the *Value-data-type-definition* are empty. The *Literal-signature-**set*** and *Static-operation-signature-**set*** are derived from <c enumerator list> as described below.

Each *Literal-signature* has a *Result* that is the *Sort* of the *Value-data-type-definition*.

<c enumerator list> ::=
    <c enumerator>
  | <c enumerator list> <comma> <c enumerator>

NOTE 2 – Clause 6.7.2.2 of [b-ISO/IEC 9899].

The number of items in the <c enumerator list> shall be less than or equal to `INT_MAX`.

<c enumerator> ::=
    <c enumeration constant>
  | <c enumeration constant> <equals sign> <c constant expression>

NOTE 3 – Clause 6.7.2.2 of [b-ISO/IEC 9899].

The <c constant expression> shall evaluate to a `Natural` simple expression less than or equal to `INT_MAX`.

<c enumeration constant> ::=
    <c identifier>

NOTE 4 – Clause 6.4.4.3 of [b-ISO/IEC 9899].

Each <c enumerator> represents a *Literal-signature*.

Each *Literal-name* is unique within the defining scope unit in the abstract syntax even if the corresponding <c identifier> of the <c enumeration constant> of the <c enumerator> is not unique. The unique *Literal-name* is derived from:

a) the <c identifier> of the <c enumeration constant> of the <c enumerator>; plus

b) the *Result* of the *Literal-signature*.

The Natural value of the <c constant expression> after an <equals sign> represents the *Literal-natural* of the *Literal-signature.*

Each <c enumeration constant> not followed by an <equals sign> in a <c enumerator list> is given the lowest possible Natural value for the *Literal-natural* of the *Literal-signature* not occurring for any other <c enumeration constant> items of the same <c enumerator list>, considering the <c enumeration constant> items one by one from left to right.

C.1.2.4 Sizeof data types

As well as representing abstract grammar items described above, a <c type definition> also represents a synonym definition for the size of the data type as defined below. A synonym definition is a read-only *Variable-definition*. The *Variable-name* is the *Name* derived from prefixing the <c identifier> for the data type with "`sizeof_`" as further described below. The *Sort-reference-identifier* is the *Sort-reference-identifier* for the `Integer` data type. The *Constant-expression* is a *Literal* that is a *Literal-identifier* that is an *Identifier* for the `Integer` value for the size of the data type.

NOTE – If the data type has an anonymous name the synonym for the data type size is also anonymous.

A <c struct or union specifier> or <c enum specifier> that is not a <c type definition> also represents a synonym definition for the size of the data type as well as representing a data type definition. The synonym is named and defined in the same way as <c struct or union specifier> or <c enum specifier> that is a <c type definition>.

There is only a *Variable-definition* for the size of a data type, if the data type is defined in a C diagram and every element of the data type has a defined size. The size of the data types `Boolean`, `Character`, `Bit`, `Octet`, `NumericChar`, `PrintableChar`, `TeletexChar` and `IA5Char` of <<**package** `Predefined`>> have defined sizes and <<**package** `C_Predefined`>> defines the corresponding synonyms: `sizeof_Boolean`, `sizeof_Character`, `sizeof_Bit`, `sizeof_Octet`, `sizeof_NumericChar`, `sizeof_PrintableChar`, `sizeof_TeletexChar` and `sizeof_IA5Char`. Other data types of <<**package** `Predefined`>> such as `Integer` are unbounded, and although they can be used in C diagrams the size of these data types and any data type having elements of these types is undefined and the corresponding synonym *Variable-definition* for the size does not exist.

If the <c type definition> is a <c type definition> for a C array, the size for the *Constant-expression* is the value of the <Integer c constant expression> for the number of array elements multiplied by the size of the item sort of the array.

If the <c type definition> defines the data type as a syntype for another type (a named data type identified by a <c identifier> or by <c type keywords>), the *Constant-expression* is the same as the *Constant-expression* for the size of the other type.

If the <c type definition> defines a syntype based on a <c type specifier> that is <c struct or union specifier> or <c enum specifier>, the *Constant-expression* is the size of the structure or union or enumerated type defined by the <c struct or union specifier> or <c enum specifier>.

If the <c type definition> is a <c struct or union specifier> for a structure type with a <c struct declaration list>, the *Constant-expression* is the size for the struct, which is implementation dependent but at least the sum of the sizes of all <c struct declarator> items in the <c struct declaration list>. The size of a <c struct declarator> in a structure that is not a C array is the size of the *Sort* of the field sort (that is, the *Sort* represented by <c type specifier> of the <c specifier qualifier list> of the <c struct declaration> that encloses the <c struct declarator list> that encloses the <c struct declarator>). The size of a <c struct declarator> in a structure that is a C array, is the size of a <c struct declarator> of the array item sort multiplied by the <Integer c constant expression> for the number of array elements.

If the <c type definition> is a <c struct or union specifier> for a union type with a <c struct declaration list>, the *Constant-expression* is the size for the union, which is the maximum of each of the sizes of all <c struct declarator> items in the <c struct declaration list>. The size of each

<c struct declarator> in a union is the same as the size of the same <c struct declarator> in a structure.

If the <c type definition> is a <c enum specifier>, the *Constant-expression* is the same as the *Constant-expression* for the synonym <<**package** C_Predefined>>sizeof_Signed_int.

In <<**package** C_Predefined>> the following data types are defined with synonyms for their size:

```
Void                    has size <<package C_Predefined>> sizeof_Void
Signed_char             has size <<package C_Predefined>>
                        sizeof_Signed_char
Unsigned_char           has size <<package C_Predefined>>
                        sizeof_Unsigned_char
Signed_short            has size <<package C_Predefined>>
                        sizeof_Signed_short
Unsigned_short          has size <<package C_Predefined>>
                        sizeof_Unsigned_short
Signed_int              has size <<package C_Predefined>>
                        sizeof_Signed_int
Unsigned_int            has size <<package C_Predefined>>
                        sizeof_Unsigned_int
Signed_long             has size <<package C_Predefined>>
                        sizeof_Signed_long
Unsigned_long           has size <<package C_Predefined>>
                        sizeof_Unsigned_long
Signed_long_long        has size <<package C_Predefined>>
                        sizeof_Signed_long_long
Unsigned_long_long      has size <<package C_Predefined>>
                        sizeof_Unsigned_long_long
Float                   has size <<package C_Predefined>> sizeof_Float
Double                  has size <<package C_Predefined>> sizeof_Double
Long_double             has size <<package C_Predefined>>
                        sizeof_Long_double
Star_Void               has size <<package C_Predefined>> sizeof_Star_Void
Star_Signed_char        has size <<package C_Predefined>>
                        sizeof_Star_Signed_char
Star_Unsigned_char      has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_char
Star_Signed_short       has size <<package C_Predefined>>
                        sizeof_Star_Signed_short
Star_Unsigned_short     has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_short
Star_Signed_int         has size <<package C_Predefined>>
                        sizeof_Star_Signed_int
Star_Unsigned_int       has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_int
Star_Signed_long        has size <<package C_Predefined>>
                        sizeof_Star_Signed_long
Star_Unsigned_long      has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_long
Star_Signed_long_long   has size <<package C_Predefined>>
                        sizeof_Star_Signed_long_long
Star_Unsigned_long_long has size <<package C_Predefined>>
                        sizeof_Star_Unsigned_long_long
```

```
Star_Float              has size <<package C_Predefined>>
                        sizeof_Star_Float
Star_Double             has size <<package C_Predefined>>
                        sizeof_Star_Double
Star_Long_double        has size <<package C_Predefined>>
                        sizeof_Star_Long_double
```

Other subtypes of the `Star_Type` have the same size as `Star_void`.

C.1.3 Use of C variable definitions

Abstract grammar

The *Sort-identifier* of a *Variable-definition* shall not represent the `Void` data type.

Concrete grammar

<variable definition> ::=
dcl <variables of sort> {**,** <variables of sort> }* <end>
| **dcl exported** <exported variables of sort> {**,** <exported variables of sort> }* <end>
| <c declaration>

A <c declaration> shall only be used in a C diagram.

<c declaration> ::=
<c declaration specifiers> <c init declarator list> <semicolon>

NOTE 1 – Clause 6.7 of [b-ISO/IEC 9899] <c declaration> omitting the static assert alternative and modified to exclude alternatives used for <c type definition> as an alternative of <data definition>. The declaration specifiers of <c declaration> are restricted to the alternatives defined by <c type specifier> items that start with a data type name or specific keywords. Compared with [b-ISO/IEC 9899] it is not allowed to omit the <c init declarator list>.

A <c declaration> represents a *Variable-definition-**set*** in the enclosing scope.

A <c declaration specifiers> identifies a *Sort-identifier* for a sort used in the *Variable-definition* for each <c init declarator> of the <c init declarator list>. This is either the sort of the variable or the target sort for a pointer or the item sort for a C array, depending on the <c declarator> for <c init declarator> as described below.

<c declaration specifiers> ::=
<c type specifier>
| <c storage class specifier> [<c declaration specifiers>]
| <c type qualifier> [<c declaration specifiers>]

NOTE 2 – Clause 6.7 of [b-ISO/IEC 9899] excluding function-specifier and alignment-specifier alternative. The syntax is rewritten so that there is one and only one <c type specifier> (that is, <c declaration specifiers> is not allowed after <c type specifier>).

If the <c type specifier> is <c type keywords> it identifies a *Sort-identifier* according to the list given for the syntax of <c type keywords> above.

If the <c type specifier> is a <data type c identifier> or <c struct or union specifier> with a <c identifier>, or <c enum specifier> with a <c identifier>, this <c identifier> represents the *Sort-identifier*.

If the <c type specifier> is a <c struct or union specifier> without a <c identifier>, or <c enum specifier> without a <c identifier>, it identifies the *Sort-identifier* for an anonymous unique identifier of the inline type definition given by the <c struct or union specifier> or <c enum specifier>.

If the <c type specifier> is a <c struct or union specifier> with a <c struct declaration list> or a <c enum specifier> with a <c enumerator list>, the <c type specifier> is an inline definition of a data type (see the grammar for <c type specifier>).

<c storage class specifier> ::=
extern
| **auto**
| **register**

NOTE 3 – Clause 6.7.1 of [b-ISO/IEC 9899] excluding **typedef** (used in <c type definition>), **static** and **_Thread_local**. The variables specified as **static** are not considered, as they are difficult to map in SDL-2010 and can be replaced with system or block variables. The keywords **auto** and **register** have no meaning in SDL-2010 and are ignored.

<c type qualifier> ::=
const
| **restrict**
| **volatile**

NOTE 4 – Clause 6.7.3 of [b-ISO/IEC 9899] excluding **_Atomic**. The keywords **volatile** and **restrict** have no meaning in SDL-2010 and are ignored.

The keyword **const** means the item should be initialized and should not be subsequently changed, but has no mapping to the abstract grammar and is therefore treated as annotation with respect to the SDL-2010 semantics. It is permitted for a tool to treat the keyword **const** as defined in clause 6.7.3 of [b-ISO/IEC 9899].

<c init declarator list> ::=
<c init declarator>
| <c init declarator list> <comma> <c init declarator>

NOTE 5 – Clause 6.7 of [b-ISO/IEC 9899].

<c init declarator> ::=
<c declarator>
| <c declarator> <equals sign> <c initializer>

NOTE 6 – Clause 6.7 of [b-ISO/IEC 9899].

Each <c init declarator> represents a *Variable-definition*. The <c identifier> (of the <c declarator> of the <c init declarator>) represents the *Variable-name* of the *Variable-definition*. The *Aggregation-kind* of the *Variable-definition* is **PART**.

If the <c declarator> of a <c init declarator> contains no <c pointer> items and no index specifications (<left square bracket> <Integer c constant expression> <right square bracket> list), the *Sort-reference-identifier* of the *Variable-definition* is the *Sort-identifier* from the <c declaration specifiers> as described above.

If the <c declarator> of a <c init declarator> contains a <c pointer> or one or more index specifications, its type is a typed pointer, or the type of a C Array and *Sort-reference-identifier* of the *Variable-definition* is the *Identifier* of this data type. The data type is derived in the same way as the *Parent-sort-identifier* of the *Syntype-definition* of a <c type definition> with a **typedef** (see clause C.1.2), except the <c type specifier> is the <c type specifier> of the <c declaration specifiers> of the <c declaration> that encloses the <c init declarator list> that encloses the <c init declarator>. As a consequence, it is possible that a <c declarator> of a <c init declarator> that contains a <c pointer> or index specification represents additional anonymous *Data-type-definition* items.

If the <c init declarator> contains a <c initializer>, this represents the optional *Constant-expression* of the *Variable-definition*.

If a <c type qualifier> is **const** for the <c declaration specifiers> of a <c declaration>, each *Variable-definition* represented by the <c declaration> is a read-only variable and shall not be used for the target of an assignment or otherwise modified. In this case, each <c init declarator> shall contain a <c initializer> that initializes the variable.

<c initializer> ::=
<constant c conditional expression>
| <left curly bracket> <c initializer list> <right curly bracket>
| <left curly bracket> <c initializer list> <comma> <right curly bracket>

NOTE 7 – Clause 6.7.9 of [b-ISO/IEC 9899].

The trailing <comma> after a <c initializer list> is ignored so this alternative has the same meaning as the alternative without a <comma>.

<c initializer list> ::=
<c initializer>
<c initializer list> <comma> <c initializer>

NOTE 8 – Clause 6.7.9 of [b-ISO/IEC 9899] excluding designation items.

A <c initializer> shall only contain a <c initializer list> within curly brackets if the variable is a structure or a union or an array. In the case of a structure, there shall be a <c initializer> in the <c initializer list> for every field of the structure and each item shall be compatible with the sort of the corresponding field. The <c initializer list> represents a call of `Make` for the structure with constants represented by the <c initializer> items as actual parameters. In the case of a union, the <c initializer> represents a call of `Make` for the union with the constant represented by the <c initializer> items as the actual parameter. In the case of an array there shall be as many items in the <c initializer list> as there are elements in the array and the sort of each element shall be the same as the item sort of the array. The <c initializer list> represents a static evaluation of `Modify` for the array with the integer value 0 as the index and the constant represented by the first <c initializer> as the other 2 parameters. If this is last element of the array, the first parameter of `Modify` is a static evaluation of the `Make` operator for the array with one parameter as a default value for the element type; otherwise it is a static evaluation of `Modify` with the index value of the next array element and the constant represented by the next <c initializer>. Nested evaluations of `Modify` are repeated until `Modify` is evaluated for the last element. For example, in

```
enum ABC {A, B, C} a3[3] = {A, B, C};
```

the <c initializer list> `{A, B, C}` is evaluated

```
Modify(Modify(Modify(Make,C,2),B,1),A,0)
```

as the *Constant-expression* for the *Variable-definition* of `a3`.

If a field sort or array element sort is itself a structure or array, the <c initializer> for the field or array element that is a nested <c initializer list> within curly brackets initializes the nested structure fields or nested array elements.

C.1.4 Use of C expressions

Concrete grammar

<expression> ::=
<expression0>
| <range check expression>
| <c expression>

The alternatives <expression0> and <range check expression> are not valid in diagrams using C syntax. The alternative <c expression> is only valid in diagrams using C syntax.

<c expression> ::=
<c assignment expression>

NOTE – Clause 6.5.17 of [b-ISO/IEC 9899] without multiple assignments separated by commas.

C.1.4.1 Assignment expression

Concrete grammar

<c assignment expression> ::=
<c conditional expression>
| <c unary expression> <c assignment operator> <c assignment expression>

NOTE 1 – Clause 6.5.16 of [b-ISO/IEC 9899].

The <c unary expression> of a <c assignment expression> is valid if it is

a) a <c postfix expression> that

i) is a <c primary expression> that is <c identifier> for a variable, and has the type of this variable; or

ii) is a <c expression> in parentheses that is a valid <c unary expression> for a <c assignment expression>, and has the type of this <c unary expression>; or

iii) is a <c postfix expression> that identifies a C array item (variable, array element or field) followed by an index specification (<left square bracket> <c expression> <right square bracket>), and has the element type of the array; or

iv) is a <c postfix expression> that identifies a structure or union item (variable, array element or field) followed by field selection (<full stop> <c identifier>), and has the type of the field selected by <c identifier>;

or

b) an <asterisk> (dereference) <c unary operator> followed by a <c cast expression> that is a <c postfix expression> that is a valid <c unary expression> for a <c assignment expression>, and <asterisk> (dereference) is defined for the type of the <c unary expression>; or

c) an <asterisk> (dereference) <c unary operator> followed by a <c cast expression> that is a cast (<left parenthesis> <c type name> <right parenthesis> <c cast expression>), and <asterisk> (dereference) is defined for the type of the <c type name >.

A <c postfix expression> identifies C array if it:

a) is a <c primary expression> that is <c identifier> for a variable that is a C array, or

b) starts with an inner <c postfix expression> that identifies C array.

<c assignment operator> ::=
<equals sign>
| <c multiplication assignment sign>
| <c division assignment sign>
| <c remainder assignment sign>
| <c addition assignment sign>
| <c subtraction assignment sign>
| <c shift left assignment sign>
| <c shift right assignment sign>
| <c bitwise and assignment sign>
| <c bitwise excl or assignment sign>
| <c bitwise incl or assignment sign>

NOTE 2 – Clause 6.5.17 of [b-ISO/IEC 9899].

A <c assignment expression> that is not a <c conditional expression> represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for a method of **package** `C_Predefined` or an implicit type for a pointer (a subtype of `Star_type` of **package** `C_Predefined)`. The <c assignment operator> determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "=" if the <c assignment operator> is <equals sign>; "*=" for the <c multiplication assignment sign>; and so on). The type of the <c unary expression> determines the type that defines the method. For example, if this is

`Signed_int` the method is defined by `Signed_int` and `Signed_int` corresponds to the *Sort-reference-identifier* of the first *Formal-argument* of the *Operation-signature*. The type of the inner <c assignment expression> determines the *Sort-reference-identifier* of the second and last *Formal-argument* of the *Operation-signature*. These items identify at most one *Operation-signature*, a method of **`package`** `C_Predefined` or implicit types for pointers. If no *Operation-signature* is identified the <c assignment expression> is not valid. The <c assignment expression> has a type identified by the *Sort-reference-identifier* of the *Operation-result* of the *Operation-signature*.

The <c unary expression> represents the *Expression* for the first item of the *Actual-parameters* of the *Operation-application*.

The inner <c assignment expression> represents the *Expression* for the second item of the *Actual-parameters* of the *Operation-application*.

C.1.4.2 Conditional expression

Concrete grammar

```
<c conditional expression> ::=
                        <c logical or expression>
                |       <c logical or expression> <question mark>
                            <c expression> <colon> <c conditional expression>
```

NOTE – Clause 6.5.15 of [b-ISO/IEC 9899].

```
<c constant expression> ::=
                    <c conditional expression>
```

A <c conditional expression> containing a <question mark> represents a *Conditional-expression* (see clause 12.2.5 of [ITU-T Z.101]). In this case, the sort of the <c logical or expression> shall be scalar sort, which is defined as one of the following:

a) an integer sort (that is, `Char`, `Signed_char`, `Unsigned_char`, `Signed_short`, `Unsigned_short`, `Signed_int`, `Unsigned_int`, `Signed_long`, `Unsigned_long_long`, `Integer` or the sort of a data type that inherits from one of these); or

b) a floating sort (`Float`, `Double`, `Long_double`, `Real` or the sort of a data type that inherits from one of these); or

c) a pointer sort (`Star_void`, a subtype of `Star_void` or a subtype of `Star_type`); or

d) a boolean sort (the `Boolean` sort or the sort of a data type that inherits from `Boolean`); or

e) a pid sort.

An integer, floating, boolean sort (other than `Boolean`) or pid sort <c logical or expression> before the <question mark> represents the *First-operand* of a negative ("/=") *Equality-expression* that is the *Boolean-expression* of the *Conditional-expression*, and the *Second-operand* of this *Equality-expression* is:

a) the value of the sort equivalent to "0" if the sort of the <c logical or expression> is an integer sort; or

b) the value of the sort equivalent to "0.0" if the sort of the <c logical or expression> is a floating sort; or

c) the value of the sort equivalent to "`false`" if the sort of the <c logical or expression> is a boolean sort; or

d) the *Null-literal-signature* if the sort of the <c logical or expression> is a pointer sort; or

e) the *Null-literal-signature* if the sort of the <c logical or expression> is a pid.

If the sort of the <c logical or expression> before the <question mark> is `Boolean`, the <c logical or expression> directly represents the *Boolean-expression* of the *Conditional-expression*.

The *Consequence-expression* in the *Conditional-expression* is represented by the <c expression> in the <c conditional expression>. The *Alternative-expression* is represented by the inner <c conditional expression> after the <colon>.

C.1.4.3 Logical and bitwise operation expressions

For most operators the operator is treated in a similar way to operators in the native SDL-2010 syntax: that is, the operator is treated as if it were written as a prefix operator or as a quoted operator name such as "+" applied to the operands.

In the case of the logical operators (<c logical or sign> `"||"`, <c logical and sign> `"&&"`), in [b-ISO/IEC 9899] the left-hand operand is evaluated first and the right-hand operand is only evaluated if necessary. So these both have to be handled as a *Conditional-expression*.

Concrete grammar

<c logical or expression> ::=
<c logical and expression>
| <c logical or expression> <c logical or sign> <c logical and expression>

NOTE 1 – Clause 6.5.14 of [b-ISO/IEC 9899].

A <c logical or expression> before a <c logical or sign> of a <c logical or expression> shall not represent a pid sort.

A <c logical or expression> containing a <c logical or sign> represents a *Conditional-expression*. In this case, the <c logical or expression> represents the *Boolean-expression* of the *Conditional-expression* as described in clause C.1.4.2 for the <c logical or expression> of a <c conditional expression> containing a <question mark>. The sort and value of the *Consequence-expression* is:

a) the value of the sort equivalent to "1" if the sort of the <c logical or expression> is an integer sort; or

b) the value of the sort equivalent to "1.0" if the sort of the <c logical or expression> is a floating sort; or

c) the value of the sort equivalent to "`true`" if the sort of the <c logical or expression> is a boolean sort or `Boolean`.

The *Alternative-expression* of the *Conditional-expression* is represented by the <c logical and expression> after the <c logical or sign>.

<c logical and expression> ::=
<c inclusive or expression>
| <c logical and expression> <c logical and sign> <c inclusive or expression>

NOTE 2 – Clause 6.5.13 of [b-ISO/IEC 9899].

A <c logical and expression> containing a <c logical and sign> represents a *Conditional-expression*. In this case, the <c logical and expression> represents the *Boolean-expression* of the *Conditional-expression* as described in clause C.1.4.2 for the <c logical or expression> of a <c conditional expression> containing a <question mark>. The *Consequence-expression* of the *Conditional-expression* is represented by the <c inclusive or expression> after the <c logical and sign>. The sort and value of the *Alternative-expression* is:

a) the value of the sort equivalent to "0" if the sort of the <c logical and expression> is an integer sort; or

b) the value of the sort equivalent to "0.0" if the sort of the <c logical and expression> is a floating sort; or

c) the value of the sort equivalent to "`false`" if the sort of the <c logical and expression> is a boolean sort; or

d) the *Null-literal-signature*, if the sort of the <c logical and expression> is a pid.

<c inclusive or expression> ::=
<c exclusive or expression>
| <c inclusive or expression> <vertical line> <c exclusive or expression>

NOTE 3 – Clause 6.5.12 of [b-ISO/IEC 9899].

<c exclusive or expression> ::=
<c and expression>
| <c exclusive or expression> <circumflex accent> <c and expression>

NOTE 4 – Clause 6.5.11 of [b-ISO/IEC 9899].

<c and expression> ::=
<c equality expression>
| <c and expression> <ampersand> <c equality expression>

NOTE 5 – Clause 6.5.10 of [b-ISO/IEC 9899].

A <c inclusive or expression> containing a <vertical line>, <c exclusive or expression> containing a <circumflex accent> or <c and expression> containing an <ampersand> represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** C_Predefined. The infix operator (<vertical line>, <circumflex accent> or <ampersand>) determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "|" for <vertical line>; "^" for <circumflex accent>; "&" for <ampersand>). The sort of the context of the <c inclusive or expression>, <c exclusive or expression> or <c and expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the infix operator, the expression is not valid. The expression before the infix operator (<c inclusive or expression> for <vertical line>; <c exclusive or expression> for <circumflex accent>; <c and expression> for <ampersand>) represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The expression after the infix operator (<c exclusive or expression> for <vertical line>; <c and expression> for <circumflex accent>; <c equality expression> for <ampersand>) represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

C.1.4.4 Equality and relational expressions

Concrete grammar

<c equality expression> ::=
<c relational expression>
| <c equality expression> <c equality sign> <c relational expression>
| <c equality expression> <c inequality sign> <c relational expression>

NOTE 1 – Clause 6.5.9 of [b-ISO/IEC 9899].

A <c equality expression> containing an equality infix operator (<c equality sign> or <c inequality sign>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** C_Predefined. The equality infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "==" for <c equality sign>; "!=" for <c inequality sign>). The sort of the context of the <c equality expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the equality infix operator, the expression is not valid. The <c equality expression> before the equality infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c relational expression> after the equality infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

<c relational expression> ::=
<c shift expression>
| <c relational expression> <less than sign> <c shift expression>
| <c relational expression> <greater than sign> <c shift expression>
| <c relational expression> <less than or equals sign> <c shift expression>
| <c relational expression> <greater than or equals sign> <c shift expression>

NOTE 2 – Clause 6.5.8 of [b-ISO/IEC 9899].

A <c relational expression> containing a relational infix operator (<less than sign> or <greater than sign> or <less than or equals sign> or <greater than or equals sign>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** C_Predefined. The relational infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "<" for <less than sign>; ">" for <greater than sign>; "<=" for <less than or equals sign>; ">=" for <greater than or equals sign>). The sort of the context of the <c relational expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the relational infix operator, the expression is not valid. The <c relational expression> before the relational infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c shift expression> after the relational infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

C.1.4.5 Bit shift expressions

Concrete grammar

<c shift expression> ::=
<c additive expression>
| <c shift expression> <c shift left sign> <c additive expression>
| <c shift expression> <c shift right sign> <c additive expression>

NOTE – Clause 6.5.7 of [b-ISO/IEC 9899].

A <c shift expression> containing a shift infix operator (<c shift left sign> or <c shift right sign>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** C_Predefined. The shift infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "<<" for <c shift left sign>; ">>" for <c shift right sign>). The sort of the context of the <c shift expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the shift infix operator, the expression is not valid. The <c shift expression> before the shift infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c additive expression> after the shift infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

C.1.4.6 Binary operation expressions

Concrete grammar

<c additive expression> ::=
<c multiplicative expression>
| <c additive expression> <plus sign> <c multiplicative expression>
| <c additive expression> <hyphen> <c multiplicative expression>

NOTE 1 – Clause 6.5.6 of [b-ISO/IEC 9899].

A <c additive expression> containing an additive infix operator (<plus sign> or <hyphen>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature*

for an operator of **package** C_Predefined. The additive infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "+" for <plus sign>; "-" for <hyphen>). The sort of the context of the <c additive expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the additive infix operator, the expression is not valid. The <c additive expression> before the additive infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c multiplicative expression> after the additive infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

<c multiplicative expression> ::=
<c cast expression>
| <c multiplicative expression> <asterisk> <c cast expression>
| <c multiplicative expression> <solidus> <c cast expression>
| <c multiplicative expression> <percent sign> <c cast expression>

NOTE 2 – Clause 6.5.5 of [b-ISO/IEC 9899].

A <c multiplicative expression> containing a multiplicative infix operator (<asterisk> or <solidus> or <percent sign>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** C_Predefined. The multiplicative infix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "*" for <asterisk>; "/" for <solidus>; "%" for <percent sign>). The sort of the context of the <c multiplicative expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the multiplicative infix operator, the expression is not valid. The <c multiplicative expression> before the multiplicative infix operator represents the first *Expression* of the *Actual-parameters* list of the *Operation-application*. The <c cast expression> after the multiplicative infix operator represents the second (and last) *Expression* of the *Actual-parameters* list of the *Operation-application*.

C.1.4.7 Cast expressions

Concrete grammar

<c cast expression> ::=
<c unary expression>
| <left parenthesis> <c type name> <right parenthesis> <c cast expression>

NOTE 1 – Clause 6.5.4 of [b-ISO/IEC 9899].

<c type name> ::=
<c specifier qualifier list>

NOTE 2 – Clause 6.7.7 of [b-ISO/IEC 9899] without the abstract declarator suffix.

The representation of a <c cast expression> without a parenthesized <c type name> is given by the representation of the <c unary expression>. In the following text a <c cast expression> that contains a parenthesized <c type name> is casting expression, and <c cast expression> after the parenthesized <c type name> in a <c cast expression> is a casted expression.

The parenthesized <c type name> in a casting expression shall identify a C integer sort or a C floating sort or a C boolean sort or Star_void or a subtype of Star_type. This sort is the casting sort.

The sort of the casted expression (the casted sort) shall be the sort of <<**package** Predefined>> Integer or a C integer sort or a C boolean sort or Star_void or a subtype of Star_type.

The casting expression represents the *Operator-application* with *Operator-identifier* of the operator of the data type for the casting sort to convert a <<**package** Predefined>> Integer value to a value of the casting sort. For example, if <c type name> is **short int** the operator is to_Signed_short of the data type Signed_short; if <c type name> is Star_Unsigned_long the operator is Integer_to_Star_Unsigned_short of the data type Star_Unsigned_long; and if the <c type name> is **double** the operator is Integer_to_Float of the data type Float. The *Actual-parameters* list of the *Operator-application* has one *Expression*.

If the casted sort is <<**package** Predefined>> Integer sort, the *Expression* of the *Actual-parameters* list of the *Operator-application* is represented by the casted expression. Otherwise, *Expression* of the *Actual-parameters* list of the *Operator-application* is represented by another *Operator-application* to convert the value of the casted expression to the <<**package** Predefined>> Integer sort. The *Actual-parameters* list of this *Operator-application* has one *Expression* of the casted sort. For a casted sort that is a C integer sort or C boolean sort, the operator is the num operator of the type of the casted sort. For a casted sort that is Star_void, the operator is the Star_void_to_Integer operator of the type Star_void. For a casted sort that is a subtype of Star_type, the operator is renamed Star_type_to_Integer operator of the subtype of Star_type (for example, Star_Unsigned_int_to_Integer for the type Star_Unsigned_int). The *Expression* of the *Actual-parameters* list of the *Operator-application* (to convert to the <<**package** Predefined>> Integer sort) is represented by the casted expression.

C.1.4.8 Unary operation expression

Concrete grammar

<c unary expression> ::=
<c postfix expression>
| <c increment operator> <c unary expression>
| <c decrement operator> <c unary expression>
| <c unary operator> <c cast expression>
| **sizeof** <c unary expression>
| **sizeof** <left parenthesis> <c type name> <right parenthesis>

NOTE 1 – Clause 6.5.3 of [b-ISO/IEC 9899] excluding the **_Alignof** alternative.

<c unary operator> ::=
<ampersand>
| <asterisk>
| <plus sign>
| <hyphen>
| <tilde>
| <exclamation mark>

NOTE 2 – Clause 6.5.3 of [b-ISO/IEC 9899].

A <c unary expression> containing a <c increment operator> or <c decrement operator> represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** C_Predefined. The <c increment operator> or <c decrement operator> determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "++" for <c increment operator>; "--" for <c decrement operator>). The sort of the context of the <c unary expression> and sort of the parameter (the *Expression* item of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the <c increment operator> or <c decrement operator>, the expression is not valid. The <c unary expression> after the <c increment operator> or <c decrement operator> represents the *Expression* of the *Actual-parameters* list of the *Operation-application*.

A <c unary expression> containing a <c unary operator> represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **package** C_Predefined.

The <c unary operator> determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: "`&`" for <ampersand>; "*" for <asterisk>; "+" for <plus sign>; "-" for <hyphen>; "~" for <tilde>; "!" for <exclamation mark>). The sort of the context of the <c unary expression> and sort of the parameter (the *Expression* item of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the <c unary operator>, the expression is not valid. The <c cast expression> after the <c unary operator> represents the *Expression* of the *Actual-parameters* list of the *Operation-application*.

A <c unary expression> containing **sizeof** followed by a <c unary expression> represents an *Expression* that is a *Variable_access* for the `sizeof_` synonym for the data type of the sort of the <c unary expression>. For example, if the sort is `Signed_long`, the <c unary expression> represents a *Variable_access* for the `sizeof_Signed_long`.

A <c unary expression> containing **sizeof** followed by a parenthesized <c type name> represents an *Expression* that is a *Variable_access* for the `sizeof_` synonym for the data type of the <c type name>. For example, if the <c type name> is **`signed long`**, the <c unary expression> represents a *Variable_access* for the `sizeof_Signed_long`.

C.1.4.9 Postfix expression

Concrete grammar

<c postfix expression> ::=
<c primary expression>
| <c call expression>
| <c postfix expression> <left square bracket> <c expression> <right square bracket>
| <c postfix expression> <full stop> <c identifier>
| <c postfix expression> <c pointer operator> <c identifier>
| <c postfix expression> <c increment operator>
| <c postfix expression> <c decrement operator>
| <left parenthesis> <c type name> <right parenthesis>
<left curly bracket> <c initializer list> <right curly bracket>
| <left parenthesis> <c type name> <right parenthesis>
<left curly bracket> <c initializer list> <comma> <right curly bracket>

NOTE 1 – Clause 6.5.2 of [b-ISO/IEC 9899] excluding the alternative with a *postfix-expression* followed by a parenthesized *argument-expression-list* restricted to <c call expression>.

A <c postfix expression> before a square bracketed <c expression> or a <full stop> or a <c increment operator> or a <c decrement operator> shall not be a <c primary expression> that is a <c constant> or a <c string literal>.

A <c postfix expression> containing an additive postfix operator (<c increment operator> or <c decrement operator>) represents an *Operation-application* where the *Operator-identifier* denotes an *Operation-signature* for an operator of **`package`** `C_Predefined`. The additive postfix operator determines the *Operation-name* of the *Operation-signature* (that is, the *Operation-name* for operators defined by: `postfix_inc` for <c increment operator>; `postfix_dec` for <c decrement operator>). The sort of the context of the <c multiplicative expression> and sorts of the parameters (the *Expression* items of the *Actual-parameters* list) of the operator are used to identify the exact *Operation-signature* as described in clauses 6.6 and 12.2.6 of [ITU-T Z.101]. If no *Operation-signature* is found that matches the expression with the multiplicative infix operator, the expression is not valid. The <c postfix expression> before the additive postfix operator represents the *Expression* of the *Actual-parameters* list of the *Operation-application*.

A <c postfix expression> that starts with a parenthesized <c type name> (without or with a <comma> after the <c initializer list>) is a compound literal and shall not be followed by a <c increment operator> or <c decrement operator>. The compound literal represents the static

evaluation of the nested calls of the operators as described in clause C.1.3 and the <c postfix expression> represents an *Operator-application* for calling outermost `Modify` operator.

<c call expression> ::=
<c identifier> <left parenthesis> [<c argument expression list>] <right parenthesis>

The <c identifier> in a <c call expression> shall uniquely identify either a *Procedure-identifier* or an *Operation-identifier*. In the context of a <c postfix expression> procedures that do not have a *Result* are excluded. The *Procedure-identifier* or *Operation-identifier* is determined using the name of the <c identifier>, the sorts of the argument list and the sort of the context of the <c call expression> as described in clause 12.2.6 of [ITU-T Z.101]. If more than one procedure or operation is identified, the <c call expression> is invalid. The <c call expression> represents the *Call-node* if it is a <c call expression>; otherwise the <c call expression> represents *Value-returning-call-node* or *Operation-application* of the identified procedure or operation.

<c argument expression list> ::=
<c assignment expression>
| <c argument expression list> <comma> <c assignment expression>

NOTE 2 – Clause 6.5.2 of [b-ISO/IEC 9899].

Each <c assignment expression> in the <c argument expression list> represents an *Expression* in the *Actual-parameters* of the *Call-node* or *Value-returning-call-node* or *Operation-application*.

Model

A <c postfix expression> followed by a <c expression> between square brackets is derived concrete syntax for:

`Extract` (<c postfix expression> , <c assignment expression>)

treated as a <c call expression> where the <c argument expression list> is the <c postfix expression> followed by the <c expression>. If this <c postfix expression> is also a <c postfix expression> followed by a <c expression> between square brackets, this is similarly derived concrete syntax. For example, `a[i][j]` becomes `Extract(Extract(a,i),j)`. The abstract syntax is determined from the derived concrete expression.

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.

A <c postfix expression> followed by a <full stop> and a <c identifier> is derived concrete syntax for:

field-extract-name (<c postfix expression>)

where the *field-extract-name* is formed from the concatenation of the field name identified by the <c identifier> and "`Extract`" in that order. The abstract syntax is determined from the derived concrete expression.

A <c postfix expression> followed by <c pointer operator> {`->`) and a <c identifier> is transformed to the dereference operator (`*`) applied to the <c postfix expression> followed by a field extraction, so that it is derived concrete syntax for:

field-extract-name `(` `"*"` (<c postfix expression>))

where the *field-extract-name* is formed from the concatenation of the field name and "`Extract`" in that order. The field name given by <c identifier> shall be a field for the structure or union pointed at by the <c postfix expression>. The abstract syntax is determined from this concrete expression.

NOTE 3 – Expression `struct_or_union_pointer->field` is the same as `(*struct_or_union_pointer).field`.

C.1.4.10 Primary expression

Concrete grammar

<c primary expression> ::=
<c identifier>
| <c constant>
| <c string literal>
| <left parenthesis> <c expression> <right parenthesis>

NOTE 1 – Clause 6.5.1 of [b-ISO/IEC 9899] excluding generic selection.

NOTE 2 – The lexical rules in clause C.1.1 describe binding to abstract grammar for <c identifier> (see clause C.1.1.6), <c constant> (see clause C.1.1.4) and <c string literal> (see clause C.1.1.5).

A <c identifier> of a <c primary expression> is a variable access or a call of an operation (that has no parameters) or a call of a value-returning procedure (that has no parameters) or the literal (enumeration constant) of an enumerated type. If it is not possible to bind the <c identifier> to exactly one of these items, the <c primary expression> is invalid.

A c primary expression> represents an *Expression* that is either a *Constant-expression* or *Active-expression*.

If the <c identifier> is bound to the *Variable-name* of a *Variable-definition*, the <c identifier> represents an *Active-expression* that is a *Variable-access* of the identified variable.

If the <c identifier> is bound to the *Operation-name* of an *Operation-signature*, the <c identifier> represents an *Active-expression* that is an *Operation-application* of the identified operation.

If the <c identifier> is bound to the *Procedure-name* of a *Procedure-definition*, the <c identifier> represents an *Active-expression* that is a *Value-returning-call-node* of the identified procedure.

If the <c identifier> is bound to the *Identifier* for the *Literal-signature* for an enumeration, the <c identifier> represents a *Constant-expression* for the identified *Literal*.

A <c string literal> of a <c primary expression> represents an *Active-expression* that is a *Variable-access* for the `<<package Predefined>> Star_Unsigned_char` synonym of the <c string literal> as defined in clause C.1.1.5.

A parenthesized <c expression> of a <c primary expression> represents the *Expression* represented by the <c expression>.

<c constant> ::=
<c integer constant>
| <real name>
| <c character constant>

NOTE 3 – Clause 6.4.4 of [b-ISO/IEC 9899] but using <real name> for a floating constant and excluding enumeration constant, because <c identifier> covers this.

A <c integer constant> of a <c constant> represents a *Constant-expression* that is a `<<package Predefined>> Integer` *Literal* for the `Integer` value as in clause C.1.1.2.

A <real name> of a <c constant> represents a *Constant-expression* that is a `<<package Predefined>> Real` *Literal* for the `Real` value of the integer as determined by clause 14.7.1.

A <c character constant> represents a `<<package Predefined type Unsigned_char>>` *Constant-expression* that is the *Operation-application* defined in clause C.1.1.4.

C.1.5 Use of C statements

Statements are allowed in task bodies and in compound statements. A compound statement is itself a statement and the native SDL-2010 compound statement is extended to include C type definitions so that it matches the compound statement of clause 6.8 of [b-ISO/IEC 9899].

A statement is either a terminating statement that transfers the thread of control, or otherwise a non-terminating statement. In a C diagram a terminating statement or non-terminating statement is always a C statement (which includes the extended compound statement).

C.1.5.1 Compound statement

Abstract grammar

Compound-node :: *Connector-name*
*Data-type-definition-**set***
*Variable-definition-**set***
*Init-graph-node**
While-graph-node
Transition
*Step-graph-node**

Compound-node is extended compared with clause 11.14.1 of [ITU-T Z.102] to include a *Data-type-definition-**set***.

Concrete grammar

<compound statement> ::=
[<connector name> :] [<comment body>]
<left curly bracket>
{ <c type definition> <end> }*
[<variable definitions> <end>]
[<statements>] <end>*
<right curly bracket>

The syntax of <compound statement> is extended compared with clause 11.14.1 of [ITU-T Z.102] to allow <c type definition> items each of which represents an element of the *Data-type-definition-**set***.

A <comment body> of a <compound statement> is not allowed in C diagrams. A <c type definition> of a <compound statement> is only allowed in C diagrams.

Semantics

A *Compound-node* is a scope unit for each *Data-type-definition* of the *Compound-node*.

C.1.5.2 Non-terminating and terminating statement

<non terminating statement> ::=
<statement>
| <compound statement>
| <loop statement>
| <decision statement>
| <c labeled statement>
| <c expression statement>
| <c selection statement>
| <c iteration statement>

In <non terminating statement> the alternatives <statement>, <loop statement> and <decision statement> are not allowed in C diagrams. In <non terminating statement> the alternatives <c labeled statement>, <c expression statement>, <c selection statement> and <c iteration statement> are only allowed in C diagrams.

<terminating statement> ::=
<return statement>
| <stop statement>
| <break statement>
| <c jump statement>

In <terminating statement> the alternatives <return statement>, <stop statement> and <break statement> are not allowed in C diagrams. The alternative <c jump statement> of <terminating statement> is only allowed in C diagrams.

C.1.5.3 Labeled statement

<c labeled statement> ::=
<connector name> <colon> <c statement>
| **case** <c constant expression> <colon> <c statement>
| **default** <colon> <c statement>

NOTE – Clause 6.8.1 of [b-ISO/IEC 9899] with <connector name> instead of identifier before the colon.

A <c statement> with a <connector name> represents the *Compound-statement* where the <connector name> represents the *Connector-name*, the <c statement> represents the *Transition* and the *Variable-definition-**set***, *Init-graph-node* list, *While-graph-node Expression* list and *Step-graph-node* list are all empty.

A **case** <c constant expression> or **default** label shall appear only as part of the <c statement> of a **switch** <c selection> statement, and the binding to the abstract grammar for these is described in clause C.1.5.5. The <c constant expression> of each **case** label shall have an integer sort. No two of the **case** <c constant expression> items in the same **switch** <c selection> shall have the same value except any **case** <c constant expression> item for a **switch** <c selection> within the **switch** <c selection>. There shall be at most one **default** label for a **switch** <c selection> except a **default** label for a **switch** <c selection> within the **switch** <c selection>.

C.1.5.4 C statement

<c statement> ::=
<c labeled statement>
| <compound statement>
| <c expression statement>
| <c selection statement>
| <c iteration statement>
| <c jump statement>

NOTE 1 – Clause 6.8 of [b-ISO/IEC 9899] with <compound statement> for the C compound statement. <c jump statement> is a <terminating statement>; the other statements are non-terminating.

<c expression statement> ::=
[<c assignment statement> | <c call expression>] <semicolon>

NOTE 2 – Clause 6.8.4 of [b-ISO/IEC 9899] but with expression changed to [<c assignment statement> | <c call expression>] so that <c expression> is restricted to a <c assignment expression> with a <c assignment operator> or a <c postfix expression> that is <c identifier> <left parenthesis> [<c argument list>] <right parenthesis> (a <c call expression>).

<c assignment statement> ::=
<c postfix expression> <c assignment operator> <c assignment expression>

The <c postfix expression> of a <c assignment statement> shall start with a <c primary> that is a <c identifier> for a variable and shall not contain a <c increment operator> or <c decrement operator>.

A <c assignment statement> represents an *Assignment* or a *Call-node*, as further described below.

A <c assignment statement> where the <c assignment operator> is <equals> represents an *Assignment* (of a *Task-node* of a *Graph-node* of a *Transition*) as further described below. Otherwise the <c assignment statement> represents a *Call-node*. In this case the <c assignment operator>, the sort of the <c postfix expression> and sort of the <c assignment expression> identify an *Operation-signature* in the way described in clause C.1.4.1 (where the <c unary expression> in clause C.1.4.1 is the <c postfix expression> of the <c assignment statement>). If no such method exists, the <c assignment statement> is invalid. The *Call-node* invokes *Procedure-definition* identified by the

Procedure-identifier of the *Operation-signature*. The <c postfix expression> and <c assignment expression> represent the first and second *Expression* of the *Actual-parameters* list of the *Call-node*. If the <c postfix expression> contains <left square bracket> <c expression> <left square bracket> (an indexed variable) or <full stop> <c identifier> (a field variable), this is an extended variable used as the first parameter of the procedure.

The <c identifier> that starts the <c postfix expression> of a <c assignment statement> for an *Assignment* represents the *Variable-identifier* of the *Assignment*. If the <c postfix expression> contains <left square bracket> <c expression> <left square bracket> (an indexed variable) or <full stop> <c identifier> (a field variable), this is an extended variable and the *Expression* that the <c assignment expression> represents is derived in the same way as for an SDL-2010 extended variable as described in clause 12.3.3.1 of [ITU-T Z.101]. Otherwise the <c assignment expression> represents the *Expression* of the *Assignment*. However, if the sort of the *Expression* is not the same as the sort of the *Variable-identifier* in the following cases this *Expression* is the *Expression* of an *Actual-parameters* list of an *Operation-application* for an operator used to convert the right-hand side to the type of the left-hand side as described below (or for a `Boolean` <c assignment expression> a *Boolean-expression*).

The *Expression* of the *Assignment* is:

a) If the type of the *Variable-identifier* is `<<`**`package`** `Predefined>> Integer`, and the type of the right-hand side *Expression* is:

   i) a C integer type or a C boolean type, an *Operation-application* for the `num` operator for the C type with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the `num` operator;

   ii) a C floating type, an *Operation-application* for the `fix` operator for `<<`**`package`** `Predefined>> Real` with *Operation-application* of the `to_Real` operator of the C floating type as the *Expression* of the *Actual-parameters* list of the `fix` operator. The *Expression* from the <c assignment expression> is the *Actual-parameters* list of the `to_Real` operator;

   iii) `<<`**`package`** `C_Predefined>> Star_void`, an *Operation-application* for the `Star_void_to_Integer` operator of `Star_void` with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the `Star_void_to_Integer` operator;

   iv) a subtype of `<<`**`package`** `C_Predefined>> Star_type`, an *Operation-application* for the renamed `Star_type_to_Integer` operator of the subtype of `Star_type` with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the renamed `Star_type_to_Integer` operator;

   v) `<<`**`package`** `Predefined>> Real`, an *Operation-application* for the `fix` operator for `<<`**`package`** `Predefined>> Real` with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the `fix` operator;

   vi) `<<`**`package`** `Predefined>> Boolean`, a *Conditional-expression* with a *Boolean-expression* that is the *Expression* from the <c assignment expression>, a *Consequence-expression* that is the `<<`**`package`** `Predefined>> Integer` value 1, and a *Consequence-expression* that is the `<<`**`package`** `Predefined>> Integer` value 0.

b) If the type of the *Variable-identifier* is a C integer type, and the type of the right-hand side *Expression* is:

   i) `<<`**`package`** `Predefined>> Integer`, an *Operation-application* for the `integer` operator of the type `Integern` renamed for the C integer type with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

ii) a C integer type or a C boolean type or <<**package** C_Predefined>> Star_void or a subtype of <<**package** C_Predefined>> Star_type or <<**package** Predefined>> Real, an *Operation-application* for the integer operator of the type Integern renamed for the left-hand side C integer type with a <<**package** Predefined>> Integer *Expression* as in a) above as the *Actual-parameters* list of the operator.

c) If the type of the *Variable-identifier* is a C integer type, and the type of the right-hand side *Expression* is:

i) <<**package** Predefined>> Integer, an *Operation-application* for the integer operator of the type Integern renamed for the C integer type with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

ii) a C integer type or a C boolean type or <<**package** C_Predefined>> Star_void or a subtype of <<**package** C_Predefined>> Star_type or <<**package** Predefined>> Real, an *Operation-application* for the integer operator of the type Integern renamed for the left-hand side C integer type with a <<**package** Predefined>> Integer *Expression* as in a) above as the *Actual-parameters* list of the operator.

d) If the type of the *Variable-identifier* is a C floating type, and the type of the right-hand side *Expression* is:

i) <<**package** Predefined>> Integer, an *Operation-application* for the Integer_to_Float operator for <<**package** C_Predefined>> Float with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

ii) a C integer type or a C boolean type or <<**package** C_Predefined>> Star_void or a subtype of <<**package** C_Predefined>> Star_type or <<**package** Predefined>> Real, an *Operation-application* for the Integer_to_Float operator for <<**package** C_Predefined>> Float with a <<**package** Predefined>> Integer *Expression* as in a) above as the *Actual-parameters* list of the operator.

e) If the type of the *Variable-identifier* is <<**package** C_Predefined>> Star_void, and the type of the right-hand side *Expression* is:

i) <<**package** Predefined>> Integer, an *Operation-application* for the Integer_to_Star_void operator for <<**package** C_Predefined>> Star_void with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

ii) a C integer type or a C boolean type or a subtype of <<**package** C_Predefined>> Star_type or <<**package** Predefined>> Real, an *Operation-application* for the Integer_to_Star_void operator for <<**package** C_Predefined>> Star_void with a <<**package** Predefined>> Integer *Expression* as in a) above as the *Actual-parameters* list of the operator.

f) If the type of the *Variable-identifier* is a subtype of <<**package** C_Predefined>> Star_type, and the type of the right-hand side *Expression* is:

i) <<**package** Predefined>> Integer, an *Operation-application* for the renamed Integer_to_Star_type operator for the subtype of <<**package** C_Predefined>> Star_type with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator;

ii) a C integer type or a C boolean type or <<**package** C_Predefined>> Star_void or a subtype of <<**package** C_Predefined>> Star_type or <<**package** Predefined>> Real, an *Operation-application* for the renamed Integer_to_Star_type operator for the subtype of <<**package** C_Predefined>> Star_type with a <<**package**

`Predefined>> Integer` *Expression* as in a) above as the *Actual-parameters* list of the operator.

g) If the type of the *Variable-identifier* is the type of `<<`**`package`** `Predefined>> Charstring`, and the type of the right-hand side *Expression* is `<<`**`package`** `C_Predefined>> Star_Unsigned_char`, an *Operation-application* of the `<<`**`package`** `C_Predefined type Star_Unsigned_char>> Star_Unsigned_char_to_Charstring` operator with the *Expression* from the <c assignment expression> as the *Actual-parameters* list of the operator.

C.1.5.5 C selection statement

Concrete grammar

<c selection statement> ::=
if <left parenthesis> <c expression> <right parenthesis> <c statement>
| **if** <left parenthesis> <c expression> <right parenthesis> <c statement> **else** <c statement>
| **switch** <left parenthesis> <c expression> <right parenthesis> <c statement>

NOTE – Clause 6.8.4 of [b-ISO/IEC 9899].

The <c expression> shall be:

a) an integer sort (that is, `Char`, `Signed_char`, `Unsigned_char`, `Signed_short`, `Unsigned_short`, `Signed_int`, `Unsigned_int`, `Signed_long`, `Unsigned_long_long`, `Signed_long`, `Unsigned_long_long`, `Integer` or the sort of a data type that inherits from one of these); or

b) a floating sort (`Float`, `Double`, `Long_double`, `Real` or the sort of a data type that inherits from one of these); or

c) provided the <c expression> is not in a **switch** <c selection statement>, a pointer sort (`Star_void`, a subtype of `Star_void` or a subtype of `Star_type`); or

d) a boolean sort (the `Boolean` sort or the sort of a data type that inherits from `Boolean`); or

e) a pid sort.

A <c selection statement> represents a *Compound-node*. A newly created anonymous name represents the *Connector-name*. The *Variable-definition-**set***, *Init-graph-node* list, *Expression* list of the *While-graph-node*, and *Step-graph-node* list of the *Compound-node* are empty. There is no *Finalization-node* in the *While-graph-node*. The *Transition* is an empty *Graph-node* list followed by a *Decision-node* and in the case of a **switch** <c selection statement> a set of case *Free-action* items that follow the *Decision-node*. The <c expression> of the <c selection statement> represents the *Decision-question* of the *Decision-body* of the *Decision-node*. The *Decision-answer-**set*** of the *Decision-body* of the *Decision-node* is represented by the <c statement> after the <right parenthesis> in the <c selection statement> as described below.

For an **if** <c selection statement>, the *Decision-answer-**set*** has only one element and the <c statement> after the <right parenthesis> represents the *Transition* of the *Decision-answer* of the *Decision-answer-**set***. The *Range-condition* for this *Decision-answer* is represented by an *Open-range* where the *Operator-identifier* identifies the operator of a negative ("/=") *Equality-expression* of the sort of <c expression>, the <c expression> represents the *First-operand* of the *Equality-expression* and the *Second-operand* of the *Equality-expression* is:

a) the value of the sort equivalent to "0" if the sort of the <c logical or expression> is an integer sort; or

b) the value of the sort equivalent to "0.0" if the sort of the <c logical or expression> is a floating sort; or

c) the value of the sort equivalent to "`false`" if the sort of the <c logical or expression> is a boolean sort; or

d) the *Null-literal-signature* if the sort of the <c logical or expression> is a pointer sort; or

e) the *Null-literal-signature* if the sort of the <c logical or expression> is a pid.

If there is an **else** in an **if** <c selection statement>, the <c statement> after the **else** represents the *Transition* of the optional *Else-answer* of the *Decision-body* of the *Decision-node*.

For a **switch** <c selection statement>, the <c statement> after the <right parenthesis> represents the *Decision-answer-**set*** and also represents the set of *Free-action* items following the *Decision-node*.

Each **case** <c labeled statement> within this <c statement> (and not within an inner **switch** <c selection statement>) represents a *Decision-answer*. The *Range-condition* for this *Decision-answer* is represented by the <c constant expression> of the **case** <c labeled statement>. The *Transition* for this *Decision-answer* is a *Join-node* to an implicit anonymous *Connector-name* for the *Free-action* represented by the <c statement> of the **case** <c labeled statement>. The *Free-action* consists of this *Connector-name* followed by the *Transition* represented by the <c statement> of the **case** <c labeled statement>. If this <c statement> represents a *Graph-node* list without a *Terminator* or *Decision-node*, a *Join-node* to the *Free-action* for the following **case** <c labeled statement> (or **default** <c labeled statement>) is inserted. If there is no such following <c labeled statement>, a *Break-node* for the *Compound-statement* is inserted. A **default** <c labeled statement> (within the <c statement> after the <right parenthesis> of a **switch** <c selection statement> and not within an inner **switch** <c selection statement>) represents the optional *Else-answer* of the *Decision-body* of the *Decision-node*. The *Transition* for this *Else-answer* is a *Join-node* to an implicit anonymous *Connector-name* for the *Free-action* represented by the <c statement> of the **default** <c labeled statement>. The *Free-action* consists of this *Connector-name* followed by the *Transition* represented by the <c statement> of the **default** <c labeled statement>. The *Transition* is completed if necessary by a *Join-node* or *Break-node* in the same way as for a **case** <c labeled statement>.

C.1.5.6 C iteration and jump statements

Concrete grammar

```
<c iteration statement> ::=
                    while <left parenthesis> <c expression> <right parenthesis> <c statement>
                |   do <c statement> while
                        <left parenthesis> <c expression> <right parenthesis> <semicolon>
                |   for <left parenthesis>
                        [ <c expression> { <comma> <c expression> }*] <semicolon>
                        [ <c expression> ] <semicolon>
                        [ <c expression> { <comma> <c expression> }*]
                    <right parenthesis> <c statement>
```

NOTE 1 – Clause 6.8.5 of [b-ISO/IEC 9899] with **for** syntax modified to allow multiple expressions separated commas because multiple expressions separated by commas are excluded from <c expression>. The **for** alternative with <c declaration> is excluded.

A <c iteration statement> represents a *Compound-node*. The *Connector-name* of the *Compound-node* is a newly created anonymous name. The *Variable-definition-**set*** of the *Compound-node* is empty. The *Transition* of the *Compound-node* is the *Graph-node* represented by the <c statement> of the <c iteration statement> followed by a *Continue-node* with the *Connector-name* of the *Compound-node*. The *Finalization-node* of the *While-graph-node* is absent.

In a **while** <c iteration statement> the *Init-graph-node* list and *Step-graph-node list* are both empty. The parenthesized <c expression> after **while** shall be an expression of a scalar sort (see clause C.1.4.2). This scalar sort expression represents the Boolean *Expression* that forms the *Expression* list of the *While-graph-node* in the same way as the *Boolean-expression* of the *Conditional-expression* is represented in clause C.1.4.2. If the parenthesized <c expression> after **while** is a boolean expression it represents the Boolean *Expression* that forms the *Expression* list of the *While-graph-node*.

In a **do** <c iteration statement> the *Init-graph-node* list is empty and the *While-graph-node* has an empty *Expression* list. The parenthesized <c expression> after **while** shall be an expression of a scalar sort (see clause C.1.4.2). This scalar expression represents Boolean *Expression* in the same way as the *Boolean-expression* of the *Conditional-expression* is represented in clause C.1.4.2. The *Step-graph-node* list is a single *Decision-node* with the Boolean *Expression* as the *Decision-question* of the *Decision-body*, and the *Decision-answer-**set*** is a single *Decision-answer* with the *Range-condition* represented by the Boolean value `false`. The *Transition* of this *Decision-answer* is a *Break-node* with the *Connector-name* being the anonymous name of the *Compound-node* for the <c iteration statement>.

The list of <c expression> items before the <semicolon> of a **for** <c iteration statement> form an *Expression* list for the *Init-graph-node* list in the order of the <c expression> items left to right. Each *Expression* in the list is the *Expression* of an *Assignment-node* that is a *Task-node* in the *Init-graph-node* list of the *Compound-node*.

The <c expression> between the first and second <semicolon> of a **for** <c iteration statement> shall be an expression of a scalar sort and represents the *While-graph-node* of the *Compound-node* in the same way as described above for <c expression> of a **while** <c iteration statement>.

The list of <c expression> items after the second <semicolon> of a **for** <c iteration statement> form an *Expression* list for the *Step-graph-node* item in the order of the <c expression> items left to right. Each *Expression* in the list is the *Expression* of an *Assignment-node* that is a *Task-node* in the *Step-graph-node* list of the *Compound-node*.

Each *Assignment-node* that is a *Task-node* in an *Init-graph-node* or a *Step-graph-node* list has an anonymous implicit variable identified by its *Variable-identifier* of the same sort as the *Expression* of the *Assignment-node*.

<c jump statement> ::=
    goto <c identifier> <semicolon>
  | **break** <semicolon>
  | **continue** <semicolon>
  | **return** [<c expression>] <semicolon>

NOTE 2 – Clause 6.8.6 of [b-ISO/IEC 9899].

A **goto** <c jump statement> represents a *Join-node* with the *Connector-name* represented by the <c identifier> of the <c jump statement>.

A **break** <c jump statement> is only valid within the <c statement> in a **switch**, **while**, **do** or **for** <c iteration statement>. It is mapped to a **join** on the anonymous connector *anon-break* for the enclosing **switch**, **while**, **do** / **while** or **for** statement.

A **break** <c jump statement> represents a *Break-node* and the *Connector-name* is the name of the immediately enclosing *Compound-node*.

A **continue** <c jump statement> is only valid within the <c statement> in a **while**, **do** or **for** <c iteration statement>. It is mapped to a **join** on the anonymous connector *anon-continue* for the enclosing **while**, **do** / **while** or **for** statement.

A **continue** <c jump statement> represents a *Continue-node* and the *Connector-name* is the name of the immediately enclosing *Compound-node*.

If a **return** <c jump statement> with an <c expression> represents a *Value-return-node* and the <c expression> represents the *Expression* of the *Value-return-node*.

A **return** <c jump statement> without an <expression> represents an *Action-return-node*.

C.1.6 Package C_Predefined

In the following definitions, all references to names defined in the **`package`** `Predefined` are treated as prefixed by the qualification `<<`**`package`** `Predefined>>`. Similarly, all references to names

defined in the **package** C_Predefined are treated as prefixed by the qualification <<**package** C_Predefined>>. To increase readability, these qualifications are omitted.

The extensions defined in Annex A are used in this package.

There are no literals for the C Integer types. When a <c integer constant> is used, it represents an Integer literal defined in the **package** Predefined, so no ambiguity is introduced between various integer literals.

Every C Integer type defined below includes a cast operator that converts an Integer value to the C Integer type. The name of this operator is to_ concatenated with the name of the type.

This package assumes that C Integer types values do not have padding bits, have a sign bit for signed integers, use two's complement arithmetic and are wrapped when they overflow. For example, for Unsigned_char as defined below with 8 bits to_Unsigned_char(255) + to_Unsigned_char(1) = to_unsigned_char (0) and for Signed_char as defined below with 8 bits to_Signed_char(127) + to_Signed_char(1) = to_signed_char(-128).

Library functions of clause 7 of the C standard clause 5.2.4.2 of [b-ISO/IEC 9899] are not included in **package** C_Predefined. The mechanism for including such C library items is implementation defined, but one possibility is to use the standard header names (such as assert) as the names of packages. The C library item stdint defines some general cases of integers where the number of bits for the C Integer type is *N* with names uint*N*_t for the unsigned case and int*N*_t for the signed case, the value range for uint*N*_t is to_Uint*N*_t(0) to to_Uint*N*_t(power(2,N)-1) and to_Uint*N*_t(power(2,*N*)-1) + to_Uint*N*_t(1) = to_Uint*N*_t(0), and the value range for int*N*_t is to_int*N*_t(-power(2,*N*-1)) to to_intN_t(power(2,*N*-1)-1) and to_int*N*_t(power(2,*N*-1)-1) + to_int*N*_t(1) = to_intN_t(-power(2,*N*-1)).

```
/* */
package C_Predefined
/*
```

The following <package public> defines the synonyms and types that are visible wherever C_Predefined is visible.

It is not necessary to explicitly list operations as visible, because literals and operations defined by a type are visible where the type is visible except if the operation has **private** visibility (in which case it is only visible within the data type where it is defined) or **protected** visibility (in which case the operator is visible only within the data type where it is defined and within any specialization of this data type).

```
*/
public
/* synonyms */
synonym CHAR_BIT,
synonym CHAR_MAX,
synonym CHAR_MIN,
synonym DBL_DECIMAL_DIG,
synonym DBL_DIG,
synonym DBL_EPSILON,
synonym DBL_MANT_DIG,
synonym DBL_MAX_10_EXP,
synonym DBL_MAX_EXP,
synonym DBL_MAX,
synonym DBL_MIN_10_EXP,
synonym DBL_MIN_EXP,
synonym DBL_MIN,
synonym DBL_TRUE_MIN,
synonym DECIMAL_DIG,
synonym FLT_DECIMAL_DIG,
synonym FLT_DIG,
synonym FLT_EPSILON,
synonym FLT_MANT_DIG,
synonym FLT_MAX_10_EXP,
```

```
synonym FLT_MAX_EXP,
synonym FLT_MAX,
synonym FLT_MIN_10_EXP,
synonym FLT_MIN_EXP,
synonym FLT_MIN,
synonym FLT_RADIX,
synonym FLT_TRUE_MIN,
synonym INT_BIT,
synonym INT_MAX,
synonym INT_MIN,
synonym LDBL_DECIMAL_DIG,
synonym LDBL_DIG,
synonym LDBL_EPSILON,
synonym LDBL_MANT_DIG,
synonym LDBL_MAX_10_EXP,
synonym LDBL_MAX_EXP,
synonym LDBL_MAX,
synonym LDBL_MIN_10_EXP,
synonym LDBL_MIN_EXP,
synonym LDBL_MIN,
synonym LDBL_TRUE_MIN,
synonym LLONG_BIT,
synonym LLONG_MAX,
synonym LLONG_MIN,
synonym LONG_BIT,
synonym LONG_MAX,
synonym LONG_MIN,
synonym MB_LEN_MAX,
synonym SCHAR_MAX,
synonym SCHAR_MIN,
synonym SHRT_BIT,
synonym SHRT_MAX,
synonym SHRT_MIN,
synonym sizeof_Bit ,
synonym sizeof_Boolean ,
synonym sizeof_Character ,
synonym sizeof_Float,
synonym sizeof_IA5Char ,
synonym sizeof_NumericChar ,
synonym sizeof_Octet ,
synonym sizeof_PrintableChar ,
synonym sizeof_Signed_char,
synonym sizeof_Signed_int,
synonym sizeof_Signed_long_long,
synonym sizeof_Signed_long,
synonym sizeof_Signed_short,
synonym sizeof_Star_Float,
synonym sizeof_Star_Signed_char,
synonym sizeof_Star_Signed_int,
synonym sizeof_Star_Signed_long_long,
synonym sizeof_Star_Signed_long,
synonym sizeof_Star_Signed_short,
synonym sizeof_Star_Unsigned_char,
synonym sizeof_Star_Unsigned_int,
synonym sizeof_Star_Unsigned_long_long,
synonym sizeof_Star_Unsigned_long,
synonym sizeof_Star_Unsigned_short,
synonym sizeof_Star_void,
synonym sizeof_TeletexChar ,
synonym sizeof_Unsigned_char,
synonym sizeof_Unsigned_int,
synonym sizeof_Unsigned_long_long,
synonym sizeof_Unsigned_long,
synonym sizeof_Unsigned_short,
synonym sizeof_void,
synonym UCHAR_MAX,
synonym UINT_MAX,
synonym ULLONG_MAX,
synonym ULONG_MAX,
synonym USHRT_MAX,
/*types*/
```

```
type Cvector,
type Double,
type Float,
type Long_Double
type Signed_char,
type Signed_int,
type Signed_long_long,
type Signed_long,
type Signed_short,
type Star_Float
type Star_Signed_char,
type Star_Signed_int,
type Star_Signed_long_long,
type Star_Signed_long,
type Star_Signed_short,
type Star_type,
type Star_Unsigned_char,
type Star_Unsigned_int,
type Star_Unsigned_long_long,
type Star_Unsigned_long,
type Star_Unsigned_short,
type Star_void,
type Unsigned_char,
type Unsigned_char,
type Unsigned_int,
type Unsigned_long_long,
type Unsigned_long,
type Void;
/*
```

C.1.6.1 Numerical limits

A C implementation is required to document all the limits specified as synonyms in this clause (see clause 5.2.4.2 of [b-ISO/IEC 9899]).

C.1.6.1.1 Private limits

The following external synonyms are additional to limits specified in clause 5.2.4.2 of [b-ISO/IEC 9899] for use within **package** C_Predefined and are considered private to this package. Each of these synonyms can therefore be consistently renamed in this package to avoid name clashes with a C implementation library or items in the system being defined.

```
*/
/* number of bits for the type Short_int*/
synonym SHRT_BIT Integer = external;/* 16 */;
/* */
/* number of bits for the type int*/
synonym INT_BIT Integer = external;/* 16 */;
/* */
/* number of bits for the type Long_int*/
synonym LONG_BIT Integer = external;/* 32 */;
/* */
/* number of bits for the type Long_long_int*/
synonym LLONG_BIT Integer = external;/* 64 */;
/*
```

C.1.6.1.2 Sizes of integer types

The values given here vary according to the implementation, and therefore are given as external synonyms. The values in the comments are derived from clause 5.2.4.2.1 of [b-ISO/IEC 9899].

```
*/
/* number of bits for smallest item that is not a bit-field (byte)*/
synonym CHAR_BIT Integer = external; /* 8 */
/* */
/* minimum value for type Signed_char */
synonym SCHAR_MIN Integer = external; /* -128 that is –power(2, CHAR_BIT-1) */
/* */
/* maximum value for type Signed_char */
synonym SCHAR_MAX Integer = external; /* +127 that is power(2, CHAR_BIT-1) – 1 */
/* */
```

```
/* maximum value for type Unsigned_char */
synonym UCHAR_MAX Integer = external; /* 255 that is power(2, CHAR_BIT) - 1 */
/* */
/* minimum value for type denoted by char */
synonym CHAR_MIN Integer = external; /* 0 - char treated as a signed integer */
/* */
/* maximum value for type denoted by char */
synonym CHAR_MAX Integer = external; /* SCHAR_MAX - char treated as a signed integer */
/* */
/* maximum number of bytes in a multibyte character, for any supported locale */
synonym MB_LEN_MAX Integer = external; /* 1 that is multibyte characters are not
allowed*/
/* */
/* minimum value for type Signed_short */
synonym SHRT_MIN Integer = external; /* -32768 that is –power(2,SHRT_BIT-1) */
/* */
/* maximum value for type Signed_short */
synonym SHRT_MAX Integer = external; /* +32767 that is power(2,SHRT_BIT-1) - 1 */
/* */
/* maximum value for type Unsigned_short */
synonym USHRT_MAX Integer = external; /* 65535 that is power(2,SHRT_BIT) - 1 */
/* */
/* minimum value for type Signed_int */
synonym INT_MIN Integer = external; /* -32768 that is –power(2,INT_BIT-1) */
/* */
/* maximum value for type Signed_int */
synonym INT_MAX Integer = external; /* +32767 that is power(2,INT_BIT-1) - 1 */
/* */
/* maximum value for type Unsigned_int */
synonym UINT_MAX Integer = external; /* 65535 that is power(2,INT_BIT) - 1 */
/* */
/* minimum value for type Signed_long */
synonym LONG_MIN Integer = external; /* -2147483648 that is –power(2,LONG_BIT-1) */
/* */
/* maximum value for type Signed_long */
synonym LONG_MAX Integer = external; /* +2147483647 that is power(2,LONG_BIT-1) - 1 */
/* */
/* maximum value for type Unsigned_long */
synonym ULONG_MAX Integer = external; /* 4294967295 that is power(2,LONG_BIT) - 1 */
/* */
/* minimum value for type Signed_long_long */
synonym LLONG_MIN Integer = external; /* -9223372036854775808
                                         that is –power(2,LLONG_BIT-1) */
/* */
/* maximum value for type Signed_long_long */
synonym LLONG_MAX Integer = external; /* +9223372036854775807
                                         that is power(2,LLONG_BIT-1) - 1 */
/* */
/* maximum value for type Unsigned_long_long */
synonym ULLONG_MAX Integer = external; /* 18446744073709551615
                                          that is power(2,LLONG_BIT) - 1 */
/*
```

C.1.6.1.3 Characteristics of floating types

The values given here vary according to the implementation, and therefore are given as external synonyms. The values in the comments are derived from clause 5.2.4.2.2 of [b-ISO/IEC 9899].

```
*/
/*  number of decimal digits, n, such that any Double floating-point number with p radix
b digits can be rounded to a floating-point number with n decimal digits and back again
without change to the value */
synonym DBL_DECIMAL_DIG Integer = external; /* 10 */
/* */
/* number of decimal digits, q, such that any Double floating-point number with q decimal
digits can be rounded into a floating-point number with p radix b digits and back again
without change to the q decimal digit */
synonym DBL_DIG Integer  = external; /* 10 */
/* */
/* the difference between 1 and the least value greater than 1 that is representable in
the Double floating point type*/
synonym DBL_EPSILON Real = external; /*10.0E-5*/
```

```
/* */
/* number of base-FLT_RADIX digits in the Double floating-point significand, p */
synonym DBL_MANT_DIG Integer = external;
/* */
/* maximum integer such that 10 raised to that power is in the range of representable
finite Double floating-point numbers */
synonym DBL_MAX_10_EXP Integer = external; /* +37 */
/* */
/*maximum representable finite Double floating-point number*/
synonym DBL_MAX Real = external; /* 10.0E37 */
/* */
/* maximum integer such that FLT_RADIX raised to one less than that power is a
representable finite Double floating-point number */
synonym DBL_MAX_EXP Integer = external;
/* */
/* minimum normalized positive double floating-point number*/
synonym DBL_MIN Real = external; /* 10.0E-37 */
/* */
/* minimum negative integer such that 10 raised to that power is in the range of
normalized Double floating-point numbers */
synonym DBL_MIN_10_EXP Integer = external; /* -37 */
/* */
/* minimum negative integer such that FLT_RADIX raised to one less than that power is a
normalized Double floating-point number, emin */
synonym DBL_MIN_EXP Integer = external;
/* */
/* minimum positive Double floating-point number */
synonym DBL_TRUE_MIN Real = external; /* 10.0E-37 */
/* */
/* number of decimal digits, n, such that any floating-point number in the widest
supported floating type with pmax radix b digits can be rounded to a floating-point
number with n decimal digits and back again without change to the value, */
synonym DECIMAL_DIG Integer = external; /* 10 */
/* */
/* number of decimal digits, n, such that any Float floating-point number with p radix b
digits can be rounded to a floating-point number with n decimal digits and back again
without change to the value */
synonym FLT_DECIMAL_DIG Integer = external; /* 6 */
/* */
/* number of decimal digits, q, such that any Float floating-point number with q decimal
digits can be rounded into a floating-point number with p radix b digits and back again
without change to the q decimal digits */
synonym FLT_DIG Integer = external; /* 6 */
/* */
/* the difference between 1 and the least value greater than 1 that is representable in
the Float floating point type*/
synonym FLT_EPSILON Real = external; /*10.0E-5*/
/* */
/* number of base-FLT_RADIX digits in the Float floating-point significand, p*/
synonym FLT_MANT_DIG Integer = external;
/* */
/*maximum representable finite Float floating-point number*/
synonym FLT_MAX Real = external; /* 10.0E37 */
/* */
/* maximum integer such that 10 raised to that power is in the range of representable
finite Float floating-point numbers */
synonym FLT_MAX_10_EXP Integer = external; /* +37 */
/* */
/* maximum integer such that FLT_RADIX raised to one less than that power is a
representable Float finite floating-point number */
synonym FLT_MAX_EXP Integer = external;
/* */
/* minimum normalized positive Float floating-point number*/
synonym FLT_MIN Real = external; /* 10.0E-37 */
/* */
/* minimum negative integer such that 10 raised to that power is in the range of
normalized Float floating-point numbers */
synonym FLT_MIN_10_EXP Integer = external; /* -37 */
/* */
/* minimum negative integer such that FLT_RADIX raised to one less than that power is a
normalized Float floating-point number, emin */
```

```
synonym FLT_MIN_EXP Integer = external; /*
/* */
/* the radix (or base) for floating point numbers */
synonym FLT_RADIX Integer = external; /*2
/* */
/* minimum positive Float floating-point number */
synonym FLT_TRUE_MIN Real = external; /* 10.0E-37 */
/* */
/* number of decimal digits, n, such that any long double  floating-point number with p
radix b digits can be rounded to a floating-point number with n decimal digits and back
again without change to the value */
synonym LDBL_DECIMAL_DIG Integer = external; /* 10 */
/* */
/* number of decimal digits, q, such that any Long_double floating-point number with q
decimal digits can be rounded into a floating-point number with p radix b digits and back
again without change to the q decimal digits*/
synonym LDBL_DIG Integer = external; /*10*/
/* */
/* the difference between 1 and the least value greater than 1 that is representable in
the Long_double floating point type*/
synonym LDBL_EPSILON Real = external; /*10.0E-5 */
/* */
/* number of base-FLT_RADIX digits in the Long_double floating-point significand, */
synonym LDBL_MANT_DIG Integer = external;
/* */
/*maximum representable finite Long_double floating-point number*/
synonym LDBL_MAX Real = external; /* 10.0E37 */
/* */
/* maximum integer such that 10 raised to that power is in the range of representable
finite Long_double floating-point numbers */
synonym LDBL_MAX_10_EXP Integer = external; /* +37 */
/* */
/* maximum integer such that FLT_RADIX raised to one less than that power is a
representable finite Long_double floating-point number */
synonym LDBL_MAX_EXP Integer = external;
/* */
/* minimum normalized positive Long_double floating-point number*/
synonym LDBL_MIN Real = external; /* 10.0E-37 */
/* */
/* minimum negative integer such that 10 raised to that power is in the range of
normalized Long_double floating-point numbers */
synonym LDBL_MIN_10_EXP Integer = external; /* -37 */
/* */
/* minimum negative integer such that FLT_RADIX raised to one less than that power is a
normalized Long_double floating-point number, emin */
synonym LDBL_MIN_EXP Integer = external;
/* */
/* minimum positive Long_double floating-point number */
synonym LDBL_TRUE_MIN Real = external; /* 10.0E-37 */
/*
```

C.1.6.1.4 Sizes of package Predefined types

The following synonyms are defined for the sizes of some types defined in **package** Predefined. Other types defined in **package** Predefined are unbounded, therefore no synonym is defined for the size.

```
*/
synonym sizeof_Bit Integer = 1;
synonym sizeof_Boolean Integer = 1;
synonym sizeof_Character Integer = 1;
synonym sizeof_IA5Char Integer = 1;
synonym sizeof_NumericChar Integer = 1;
synonym sizeof_Octet Integer = 1;
synonym sizeof_PrintableChar Integer = 1;
synonym sizeof_TeletexChar Integer = 1;
/*
```

C.1.6.2 Parameterized types for integers

A C integer type is one of 5 signed types or 5 unsigned types: `Signed_char`, `Signed_short`, `Signed_int`, `Signed_long` or `Signed_long_long`, `Unsigned_char`, `Unsigned_short`, `Unsigned_int`, `Unsigned_long` or `Unsigned_long_long`.

The types for C integers inherit the parameterized types below with the parameters bound to the size in bits of the integer type and the other signed or unsigned integer types. The operation identifiers are overloaded. For a binary operator such as the "+" operator the first and second parameter is any of the 10 signed and unsigned C integer types or the `Integer` type. The types for C integers therefore define 100 binary `"+"` operators with different signatures. Because the result is always the `Integer` type, and the integer literals always represent values of the `Integer` type, it is always possible (providing no further overloading of `"+"` is introduced by the user) to determine the correct `"+"` operator from the sorts of the parameters.

Except for comparison operators in a context where a `Boolean` is required and the `"&"` operator to obtain a pointer to an object, each operation for a C integer types has an `Integer` result. As a consequence, it is not necessary to provide operators to convert between C integer types for expressions. When there is an explicit or implied casting (for example, when an `Unsigned_char` variable `c` is assigned to a `Signed_short` variable `s`), the appropriate `num` operator is used to convert to `Integer` and the resulting value converted to the target sort: in the example given `to_Signed_short(<<type Unsigned_char>>num(s))`.

C.1.6.2.1 Integern for a generic integer type with n bits

This type is defined as a generic type for both signed and unsigned integers. The parameters `Int_char`, `Int_short`, `Int_int`, `Int_long` and `Int_long_long` are bound to each of the corresponding unsigned integer types to produce a parameterized type `Integers` for signed Integer types. For unsigned integer types, the parameters `Int_char`, `Int_short`, `Int_int`, `Int_long` and `Int_long_long` are bound to each of the corresponding signed integer types to produce a parameterized type `Integeru`.

Each binary operator is defined for `Integern` with `Integer`, `Integer` with `Integern`, `Integern` with `Integern`, and `Integern` with each of the other 9 integer types, and that are given below in `Integern` as `Integer1`, `Integer2`, `Integer3`, `Integer4`, `Int_char`, `Int_short`, `int_int`, `Int_long` and `int_long_long`. Each signed integer type is defined using `Integern` with `n` bound to the number of bits in `Integern`, and the other context parameters bound to the other signed integer types.

There are no literals for `Integern` or types based on `Integern`. Instead the values of `Integern` are represented by `<<type Integern>>integer (i)` where `i` is an `Integer` value between `-power(2,n-1)` and `power(2,n-1)-1` for signed integer types and zero to `power(2,n)-1` for unsigned integer types. The implicit ordering of values in clause 12.1.6.1 of [ITU-T Z.101] does not apply, therefore `unordered` is given.

To shorten the description of `Integern` below, for each operation only the <operation definition> is given from which the <operation signature> is constructed (see the *Model* part of clause 12.1.7).

```
abstract value type Integern <
     value type Int_char; value type Int_short; value type Int_int;
     value type Int_long; value type Int_long_long;
     /* For signed integers these are given actual parameters for unsigned integers:
        Int_char=Unsigned_char, Int_sort=Unsigned_short, Int_int = Unsigned_int,
        Int_long=Unsigned_long, Int_sort=Unsigned_long_long.
        For unsigned integers these are given actual parameters for signed integers:
        Int_char=Signed_char, Int_sort=Signed_short, Int_int = Signed_int,
        Int_long=Signed_long, Int_sort=Signed_long_long */
     synonym n Natural; /*number of bits in Integern*/
     value type Integer1; value type Integer2; value type Integer3; value type Integer4
     /* These are given actual parameters for the other integer types.
        For exmaple when defininining Signed_long_long the values are:
        Integer1 = Signed_char, Integer2 = Signed_short,
        Integer3 = Signed_int,  Integer4 = Signed_long */ >
{
     literals unordered; /* "<", ">","<=",">=", first, last, pred, succ,
        and num are not implicitly defined.
     operators and methods signature lists are constructed from the definitions
        of the operations below except for operations used here but defined in subtypes.
*/
     operators /* Signatures for operations used here but only defined in subtypes. */
        integer ( Integer )-> this Integern; /* convert to a C integer */
        num (Integern )-> Integer; /* convert to a SDL-2010 Integer */
     /* shift left operators */
        "<<" (Integern, Integer ) -> Integer;
        "<<" (Integer, Integern ) -> Integer;
     /* shift right operators */
        ">>" (Integern, Integer ) -> Integer;
        ">>" (Integer, Integern ) -> Integer;
        /* */
        "&" (Integern, Integer ) -> Integer; /* bitwise 'and' */
        "^" (Integern, Integer ) -> Integer; /* bitwise 'xor' */
        "|" (Integern, Integer ) -> Integer /*bitwise 'or' */
/*
     operator integer ( i Integer )-> this Integern
        is defined for signed/unsigned integers with different body definitions,
        and with different names.
     operator num ( nv this Integern )-> Integer – for signed integers{}
     operator num ( nv this Integern )-> Natural– for unsigned integers{}
        are defined for signed/unsigned integers with different body definitions.
*/
     operator "&" ( in/out nv this Integern )-> Star_void external;
/*   The untyped 'address of' operator.
        The typed 'address of' operator
        "&"( Integern )-> the typed star type for the integer type, and
        the dereference operator
        "*" (the typed star type for the integer type) -> Integern are
           defined using Star_type with the integer type as the actual parameter.
*/
/*   Negation and Comparisons with Boolean results */
     operator "!"  ( nv1 Integern) -> Boolean {return num(nv1)= 0 }
     operator "<"  ( nv1 Integern, i Integer) -> Boolean {return num(nv1)< i }
     operator "<"  ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)< num(nv2)}
     operator "<"  ( i Integer, nv1 Integern) -> Boolean {return i < num(nv1)}
     operator "<"  ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)< num(nv2)}
     operator "<"  ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)< num(nv2)}
     operator "<"  ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)< num(nv2)}
     operator "<"  ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)< num(nv2)}
     operator "<"  (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)< num(nv2)}
     operator "<"  (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)< num(nv2)}
     operator "<"  (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)< num(nv2)}
     operator "<"  (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)< num(nv2)}
     operator "<"  (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)< num(nv2)}
     operator "<=" ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)<=num(nv2)}
     operator "<=" ( nv1 Integern, i Integer) -> Boolean {return num(nv1)<=i }
     operator "<=" ( i Integer, nv1 Integern) -> Boolean {return i<=num(nv1) }
     operator "<=" ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)<=num(nv2)}
     operator "<=" ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)<=num(nv2)}
     operator "<=" ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)<=num(nv2)}
```

```
     operator "<=" ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)<=num(nv2)}
     operator "<=" (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)<=num(nv2)}
     operator "<=" (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)<=num(nv2)}
     operator "<=" (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)<=num(nv2)}
     operator "<=" (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)<=num(nv2)}
     operator "<=" (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)<=num(nv2)}
     operator "==" ( nv1 Integern, i Integer) -> Boolean {return num(nv1)= i }
     operator "==" ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)= num(nv2)}
     operator "==" ( i Integer, nv1 Integern) -> Boolean {return i= num(nv1) }
     operator "==" ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)= num(nv2)}
     operator "==" ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)= num(nv2)}
     operator "==" ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)= num(nv2)}
     operator "==" ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)= num(nv2)}
     operator "==" (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)= num(nv2)}
     operator "==" (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)= num(nv2)}
     operator "==" (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)= num(nv2)}
     operator "==" (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)= num(nv2)}
     operator "==" (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)= num(nv2)}
     operator "!=" ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)/=num(nv2)}
     operator "!=" ( nv1 Integern, i Integer) -> Boolean {return num(nv1)/=i}
     operator "!=" ( i Integer, nv1 Integern) -> Boolean {return i/=num(nv1)}
     operator "!=" ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)/=num(nv2)}
     operator "!=" ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)/=num(nv2)}
     operator "!=" ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)/=num(nv2)}
     operator "!=" ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)/=num(nv2)}
     operator "!=" (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)/=num(nv2)}
     operator "!=" (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)/=num(nv2)}
     operator "!=" (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)/=num(nv2)}
     operator "!=" (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)/=num(nv2)}
     operator "!=" (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)/=num(nv2)}
     operator ">"  ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  ( nv1 Integern, i Integer) -> Boolean {return num(nv1)> i }
     operator ">"  ( i Integer, nv1 Integern) -> Boolean {return i> num(nv1) }
     operator ">"  ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)> num(nv2)}
     operator ">"  (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)> num(nv2)}
     operator ">=" ( nv1 Integern, i Integer) -> Boolean {return num(nv1)>=i }
     operator ">=" ( i Integer, nv1 Integern) -> Boolean {return i>=num(nv1) }
     operator ">=" ( nv1 Integern, nv2 Integern) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" ( nv1 Integern, nv2 Integer1) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" ( nv1 Integern, nv2 Integer2) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" ( nv1 Integern, nv2 Integer3) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" ( nv1 Integern, nv2 Integer4) -> Boolean {return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_char)->Boolean{return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_short)->Boolean{return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_int)->Boolean{return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_long)->Boolean{return num(nv1)>=num(nv2)}
     operator ">=" (nv1 Integern,nv2 Int_long_long)->Boolean{return num(nv1)>=num(nv2)}
/* unary operators "+", "-" and "~" */
     operator "+" ( nv Integern )-> Integer { return num(nv);}
     operator "-" ( nv Integern )-> Integer { return 0-num(nv)}
     operator "~" ( nv Integern )-> Integer { return -(num(nv)+1)}
/* binary arithmetic operators */
     operator "*" ( nv Integern, i Integer ) -> Integer {return num(nv) * i}
     operator "/" ( nv Integern, i Integer ) -> Integer {return num(nv) / i}
     operator "%" ( nv Integern, i Integer ) -> Integer {return num(nv) rem i}
     operator "+" ( nv Integern, i Integer ) -> Integer {return num(nv) + i}
     operator "-" ( nv Integern, i Integer ) -> Integer {return num(nv) - i}
     operator "*" ( i Integer, nv Integern ) -> Integer {return i * num(nv)}
     operator "/" ( i Integer, nv Integern ) -> Integer {return i / num(nv)}
     operator "%" ( i Integer, nv Integern ) -> Integer {return i rem num(nv)}
     operator "+" ( i Integer, nv Integern ) -> Integer {return i + num(nv)}
     operator "-" ( i Integer, nv Integern ) -> Integer {return i - num(nv)}
     operator "*" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1) * num(nv2)}
     operator "/" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1) / num(nv2)}
```

```
      operator "%" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1)rem num(nv2)}
      operator "+" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1) + num(nv2)}
      operator "-" ( nv1 Integern, nv2 Integern ) -> Integer {return num(nv1) - num(nv2)}
      operator "*" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1) * num(nv2)}
      operator "/" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1) / num(nv2)}
      operator "%" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1)rem num(nv2)}
      operator "+" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1) + num(nv2)}
      operator "-" ( nv1 Integern, nv2 Integer1 ) -> Integer {return num(nv1) - num(nv2)}
      operator "*" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1) * num(nv2)}
      operator "/" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1) / num(nv2)}
      operator "%" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1)rem num(nv2)}
      operator "+" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1) + num(nv2)}
      operator "-" ( nv1 Integern, nv2 Integer2 ) -> Integer {return num(nv1) - num(nv2)}
      operator "*" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1) * num(nv2)}
      operator "/" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1) / num(nv2)}
      operator "%" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1)rem num(nv2)}
      operator "+" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1) + num(nv2)}
      operator "-" ( nv1 Integern, nv2 Integer3 ) -> Integer {return num(nv1) - num(nv2)}
      operator "*" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1) * num(nv2)}
      operator "/" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1) / num(nv2)}
      operator "%" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1)rem num(nv2)}
      operator "+" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1) + num(nv2)}
      operator "-" ( nv1 Integern, nv2 Integer4 ) -> Integer {return num(nv1) - num(nv2)}
      operator "*" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1) * num(nv2)}
      operator "/" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1) / num(nv2)}
      operator "%" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1)rem num(nv2)}
      operator "+" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1) + num(nv2)}
      operator "-" (nv1 Integern,nv2 Int_char)->Integer{return num(nv1) - num(nv2)}
      operator "*" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1) * num(nv2)}
      operator "/" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1) / num(nv2)}
      operator "%" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1)rem num(nv2)}
      operator "+" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1) + num(nv2)}
      operator "-" (nv1 Integern,nv2 Int_short)->Integer{return num(nv1) - num(nv2)}
      operator "*" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1) * num(nv2)}
      operator "/" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1) / num(nv2)}
      operator "%" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1)rem num(nv2)}
      operator "+" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1) + num(nv2)}
      operator "-" (nv1 Integern,nv2 Int_int)->Integer{return num(nv1) - num(nv2)}
      operator "*" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1) * num(nv2)}
      operator "/" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1) / num(nv2)}
      operator "%" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1)rem num(nv2)}
      operator "+" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1) + num(nv2)}
      operator "-" (nv1 Integern,nv2 Int_long)->Integer{return num(nv1) - num(nv2)}
      operator "*" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1) * num(nv2)}
      operator "/" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1) / num(nv2)}
      operator "%" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1)rem num(nv2)}
      operator "+" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1) + num(nv2)}
      operator "-" (nv1 Integern,nv2 Int_long_long)->Integer{return num(nv1) - num(nv2)}
/* shift left  other than "<<"(Integern,Integer) and "<<"(Integer,Integern)*/
      operator "<<" (nv1 Integern,nv2 Integern     )->Integer {return nv1 << num(nv2)}
      operator "<<" (nv1 Integern,nv2 Integer1     )->Integer {return nv1 << num(nv2)}
      operator "<<" (nv1 Integern,nv2 Integer2     )->Integer {return nv1 << num(nv2)}
      operator "<<" (nv1 Integern,nv2 Integer3     )->Integer {return nv1 << num(nv2)}
      operator "<<" (nv1 Integern,nv2 Integer4     )->Integer {return nv1 << num(nv2)}
      operator "<<" (nv1 Integern,nv2 Int_char     )->Integer {return nv1 << num(nv2)}
      operator "<<" (nv1 Integern,nv2 Int_short    )->Integer {return nv1 << num(nv2)}
      operator "<<" (nv1 Integern,nv2 Int_int      )->Integer {return nv1 << num(nv2)}
      operator "<<" (nv1 Integern,nv2 Int_long     )->Integer {return nv1 << num(nv2)}
      operator "<<" (nv1 Integern,nv2 Int_long_long)->Integer {return nv1 << num(nv2)}
/* shift right  other than ">>"(Integern,Integer) and ">>"(Integer,Integern)*/
      operator ">>" (nv1 Integern,nv2 Integern     )->Integer {return nv1 >> num(nv2)}
      operator ">>" (nv1 Integern,nv2 Integer1     )->Integer {return nv1 >> num(nv2)}
      operator ">>" (nv1 Integern,nv2 Integer2     )->Integer {return nv1 >> num(nv2)}
      operator ">>" (nv1 Integern,nv2 Integer3     )->Integer {return nv1 >> num(nv2)}
      operator ">>" (nv1 Integern,nv2 Integer4     )->Integer {return nv1 >> num(nv2)}
      operator ">>" (nv1 Integern,nv2 Int_char     )->Integer {return nv1 >> num(nv2)}
      operator ">>" (nv1 Integern,nv2 Int_short    )->Integer {return nv1 >> num(nv2)}
      operator ">>" (nv1 Integern,nv2 Int_int      )->Integer {return nv1 >> num(nv2)}
      operator ">>" (nv1 Integern,nv2 Int_long     )->Integer {return nv1 >> num(nv2)}
      operator ">>" (nv1 Integern,nv2 Int_long_long)->Integer {return nv1 >> num(nv2)}
/*   Negation and Comparisons with Boolean results */
```

```
operator "!"  ( nv1 Integern) -> Integer
   return if num(nv1)= 0 then 1 else 0 fi}
operator "<"  ( nv1 Integern, i Integer) -> Integer
   return if num(nv1)< i then 1 else 0 fi}
operator "<"  ( nv1 Integern, nv2 Integern) -> Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<"  ( i Integer, nv1 Integern) -> Integer
   return if i < num(nv1)then 1 else 0 fi}
operator "<"  ( nv1 Integern, nv2 Integer1) -> Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<"  ( nv1 Integern, nv2 Integer2) -> Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<"  ( nv1 Integern, nv2 Integer3) -> Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<"  ( nv1 Integern, nv2 Integer4) -> Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<"  (nv1 Integern,nv2 Int_char)->Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<"  (nv1 Integern,nv2 Int_short)->Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<"  (nv1 Integern,nv2 Int_int)->Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<"  (nv1 Integern,nv2 Int_long)->Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<"  (nv1 Integern,nv2 Int_long_long)->Integer
   return if num(nv1)< num(nv2)then 1 else 0 fi}
operator "<=" ( nv1 Integern, nv2 Integern) -> Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" ( nv1 Integern, i Integer) -> Integer
   return if num(nv1)<=i then 1 else 0 fi}
operator "<=" ( i Integer, nv1 Integern) -> Integer
   return if i<=num(nv1) then 1 else 0 fi}
operator "<=" ( nv1 Integern, nv2 Integer1) -> Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" ( nv1 Integern, nv2 Integer2) -> Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" ( nv1 Integern, nv2 Integer3) -> Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" ( nv1 Integern, nv2 Integer4) -> Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_char)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_short)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_int)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_long)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "<=" (nv1 Integern,nv2 Int_long_long)->Integer
   return if num(nv1)<=num(nv2)then 1 else 0 fi}
operator "==" ( nv1 Integern, i Integer) -> Integer
   return if num(nv1)= i then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integern) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" ( i Integer, nv1 Integern) -> Integer
   return if i= num(nv1) then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integer1) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integer2) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integer3) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" ( nv1 Integern, nv2 Integer4) -> Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_char)->Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_short)->Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_int)->Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_long)->Integer
```

```
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "==" (nv1 Integern,nv2 Int_long_long)->Integer
   return if num(nv1)= num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integern) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, i Integer) -> Integer
   return if num(nv1)/=ithen 1 else 0 fi}
operator "!=" ( i Integer, nv1 Integern) -> Integer
   return if i/=num(nv1)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integer1) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integer2) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integer3) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" ( nv1 Integern, nv2 Integer4) -> Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_char)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_short)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_int)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_long)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator "!=" (nv1 Integern,nv2 Int_long_long)->Integer
   return if num(nv1)/=num(nv2)then 1 else 0 fi}
operator ">"  ( nv1 Integern, nv2 Integern) -> Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">"  ( nv1 Integern, i Integer) -> Integer
   return if num(nv1)> i then 1 else 0 fi}
operator ">"  ( i Integer, nv1 Integern) -> Integer
   return if i> num(nv1) then 1 else 0 fi}
operator ">"  ( nv1 Integern, nv2 Integer1) -> Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">"  ( nv1 Integern, nv2 Integer2) -> Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">"  ( nv1 Integern, nv2 Integer3) -> Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">"  ( nv1 Integern, nv2 Integer4) -> Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">"  (nv1 Integern,nv2 Int_char)->Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">"  (nv1 Integern,nv2 Int_short)->Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">"  (nv1 Integern,nv2 Int_int)->Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">"  (nv1 Integern,nv2 Int_long)->Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">"  (nv1 Integern,nv2 Int_long_long)->Integer
   return if num(nv1)> num(nv2)then 1 else 0 fi}
operator ">=" ( nv1 Integern, i Integer) -> Integer
   return if num(nv1)>=i then 1 else 0 fi}
operator ">=" ( i Integer, nv1 Integern) -> Integer
   return if i>=num(nv1) then 1 else 0 fi}
operator ">=" ( nv1 Integern, nv2 Integern) -> Integer
   return if num(nv1)>=num(nv2)then 1 else 0 fi}
operator ">=" ( nv1 Integern, nv2 Integer1) -> Integer
   return if num(nv1)>=num(nv2)then 1 else 0 fi}
operator ">=" ( nv1 Integern, nv2 Integer2) -> Integer
   return if num(nv1)>=num(nv2)then 1 else 0 fi}
operator ">=" ( nv1 Integern, nv2 Integer3) -> Integer
   return if num(nv1)>=num(nv2)then 1 else 0 fi}
operator ">=" ( nv1 Integern, nv2 Integer4) -> Integer
   return if num(nv1)>=num(nv2)then 1 else 0 fi}
operator ">=" (nv1 Integern,nv2 Int_char)->Integer
   return if num(nv1)>=num(nv2)then 1 else 0 fi}
operator ">=" (nv1 Integern,nv2 Int_short)->Integer
   return if num(nv1)>=num(nv2)then 1 else 0 fi}
operator ">=" (nv1 Integern,nv2 Int_int)->Integer
   return if num(nv1)>=num(nv2)then 1 else 0 fi}
```

```
     operator ">=" (nv1 Integern,nv2 Int_long)->Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
     operator ">=" (nv1 Integern,nv2 Int_long_long)->Integer
        return if num(nv1)>=num(nv2)then 1 else 0 fi}
/* bitwise logical operators */
/*   operator "&" ( nv Integern, i Integer ) -> Integer; bitwise 'and'
     operator "^" ( nv Integern, i Integer ) -> Integer; bitwise 'xor'
     operator "|" ( nv Integern, i Integer ) -> Integer; bitwise 'or'
        For bitwise 'and','xor' and 'or' are defined for signed/unsigned integers.
        However, the other bitwise 'and'/'xor',/'or' operators are defined using the
        Here using the signatures above. Bitwise 'xor' is defined here.
*/
/*   Bitwise 'and' with signatures other than "&" (Integern, Integer ) */
     operator "&" ( i Integer, nv Integern ) -> Integer {return nv & i}
     operator "&" ( nv1 Integern, nv2 Integern ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Integer1 ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Integer2 ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Integer3 ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Integer4 ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Int_char )->Integer{return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Int_short) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Int_Int  ) -> Integer {return nv1 & num(nv2)}
     operator "&" ( nv1 Integern, nv2 Int_long ) -> Integer {return nv1 & num(nv2)}
     operator "&" (nv1 Integern,nv2 Int_long_long )->Integer{return nv1 & num(nv2)}
/*   Bitwise 'xor' with signatures other than "^" (Integern, Integer ) */
     operator "^" ( i Integer, nv Integern )-> Integer {return nv ^ i}
     operator "^" ( nv1 Integern, nv2 Integern ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Integer1 ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Integer2 ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Integer3 ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Integer4 ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_char ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_short) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_int  ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_long ) -> Integer {return nv ^ num(nv2)}
     operator "^" ( nv1 Integern, nv2 Int_long_long ) -> Integer {return nv ^ num(nv2)}
/*   Bitwise 'or' with signatures other than "|" (Integern, Integer ) */
     operator "|" ( i Integer, nv Integern ) -> Integer {return nv | i}
     operator "|" ( nv Integern, i Integer ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integern ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integer1 ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integer2 ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integer3 ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integer4 ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Integern ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Int_char ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Int_short) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Int_int  ) -> Integer {return nv | num(nv2)}
     operator "|" ( nv1 Integern, nv1 Int_long ) -> Integer {return nv | num(nv2)}
     operator "|" (nv1 Integern, nv1 Int_long_long) -> Integer{return nv | num(nv2)}
/* Real/Float arithemetic */
     operator "+"(i this Integern, r  Real)->Real{return r+float(num(i))}
     operator "+"(r  Real, i this Integern)->Real{return r+float(num(i))}
     operator "+"(i this Integern, f Float)->Real{return to_Real(f)+float(num(i))}
     operator "+"(f Float, i this Integern)->Real{return to_Real(f)+float(num(i))}
     operator "-"(i this Integern, r  Real)->Real{return float(num(i))-r}
     operator "-"(r  Real, i this Integern)->Real{return r-float(num(i))}
     operator "-"(i this Integern, f Float)->Real{return float(num(i))-to_Real(f)}
     operator "-"(f Float, i this Integern)->Real{return to_Real(f)-float(num(i))}
     operator "*"(i this Integern, r  Real)->Real{return r*float(num(i))}
     operator "*"(r  Real, i this Integern)->Real{return r*float(num(i))}
     operator "*"(i this Integern, f Float)->Real{return to_Real(f)*float(num(i))}
     operator "*"(f Float, i this Integern)->Real{return to_Real(f)*float(num(i))}
     operator "/"(i this Integern, r  Real)->Real{return float(num(i))/r}
     operator "/"(r  Real, i this Integern)->Real{return r/float(num(i))}
     operator "/"(i this Integern, f Float)->Real{return float(num(i))/to_Real(f)}
     operator "/"(f Float, i this Integern)->Real{return to_Real(f)/float(num(i))}
     /* equal to Real/Float - Boolean and Integer result*/
     operator "=="(i this Integern, r  Real)->Boolean{return float(num(i))=r}
```

```
     operator "=="(r  Real, i this Integern)->Boolean{return r=float(num(i))}
     operator "=="(i this Integern, f Float)->Boolean{return float(num(i))=to_Real(f)}
     operator "=="(f Float, i this Integern)->Boolean{return to_Real(f)=float(num(i))}
     operator "=="(i this Integern, r  Real)->Integer{return if i==r then 1 else 0 fi}
     operator "=="(r  Real, i this Integern)->Integer{return if r==i then 1 else 0 fi}
     operator "=="(i this Integern, f Float)->Integer{return if i==f then 1 else 0 fi}
     operator "=="(f Float, i this Integern)->Integer{return if r==f then 1 else 0 fi}
/* not equal to Real/Float - Boolean and Integer result*/
     operator "!="(i this Integern, r  Real)->Boolean{return float(num(i))/=r}
     operator "!="(r  Real, i this Integern)->Boolean{return r/=float(num(i))}
     operator "!="(i this Integern, f Float)->Boolean{return float(num(i))/=to_Real(f)}
     operator "!="(f Float, i this Integern)->Boolean{return to_Real(f)/=float(num(i))}
     operator "!="(i this Integern, r  Real)->Integer{return if i!=r then 1 else 0 fi}
     operator "!="(r  Real, i this Integern)->Integer{return if r!=i then 1 else 0 fi}
     operator "!="(i this Integern, f Float)->Integer{return if i!=f then 1 else 0 fi}
     operator "!="(f Float, i this Integern)->Integer{return if r!=f then 1 else 0 fi}
/* less than Real/Float - Boolean and Integer result*/*/
     operator "<"(i this Integern, r  Real)->Boolean{return float(num(i))<r}
     operator "<"(r  Real, i this Integern)->Boolean{return r<float(num(i))}
     operator "<"(i this Integern, f Float)->Boolean{return float(num(i))<to_Real(f)}
     operator "<"(f Float, i this Integern)->Boolean{return to_Real(f)<float(num(i))}
     operator "<"(i this Integern, r  Real)->Integer{return if i<r then 1 else 0 fi}
     operator "<"(r  Real, i this Integern)->Integer{return if r<i then 1 else 0 fi}
     operator "<"(i this Integern, f Float)->Integer{return if i<f then 1 else 0 fi}
     operator "<"(f Float, i this Integern)->Integer{return if r<f then 1 else 0 fi}
/* less than or equal to Real/Float - Boolean and Integer result*/
     operator "<="(i this Integern, r  Real)->Boolean{return float(num(i))<=r}
     operator "<="(r  Real, i this Integern)->Boolean{return r<=float(num(i))}
     operator "<="(i this Integern, f Float)->Boolean{return float(num(i))<=to_Real(f)}
     operator "<="(f Float, i this Integern)->Boolean{return to_Real(f)<=float(num(i))}
     operator "<="(i this Integern, r  Real)->Integer{return if i<=r then 1 else 0 fi}
     operator "<="(r  Real, i this Integern)->Integer{return if r<=i then 1 else 0 fi}
     operator "<="(i this Integern, f Float)->Integer{return if i<=f then 1 else 0 fi}
     operator "<="(f Float, i this Integern)->Integer{return if r<=f then 1 else 0 fi}
/* greater than Real/Float - Boolean and Integer result*/*/
     operator ">"(i this Integern, r  Real)->Boolean{return float(num(i))>r}
     operator ">"(r  Real, i this Integern)->Boolean{return r>float(num(i))}
     operator ">"(i this Integern, f Float)->Boolean{return float(num(i))>to_Real(f)}
     operator ">"(f Float, i this Integern)->Boolean{return to_Real(f)>float(num(i))}
     operator ">"(i this Integern, r  Real)->Integer{return if i>r then 1 else 0 fi}
     operator ">"(r  Real, i this Integern)->Integer{return if r>i then 1 else 0 fi}
     operator ">"(i this Integern, f Float)->Integer{return if i>f then 1 else 0 fi}
     operator ">"(f Float, i this Integern)->Integer{return if r>f then 1 else 0 fi}
/* greater than or equal to Real/Float - Boolean and Integer result*/*/
     operator ">="(i this Integern, r  Real)->Boolean{return float(num(i))>=r}
     operator ">="(r  Real, i this Integern)->Boolean{return r>=float(num(i))}
     operator ">="(i this Integern, f Float)->Boolean{return float(num(i))>=to_Real(f)}
     operator ">="(f Float, i this Integern)->Boolean{return to_Real(f)>=float(num(i))}
     operator ">="(i this Integern, r  Real)->Integer{return if i>=r then 1 else 0 fi}
     operator ">="(r  Real, i this Integern)->Integer{return if r>=i then 1 else 0 fi}
     operator ">="(i this Integern, f Float)->Integer{return if i>=f then 1 else 0 fi}
     operator ">="(f Float, i this Integern)->Integer{return if r>=f then 1 else 0 fi}
/* Prefix increment/decrement */
     method "++"  -> Integer {this := <<type Integern>>integer(this+1); return this }
     method "--"  -> Integer {this := <<type Integern>>integer(this-1); return this }
/* Postfix increment/decrement */
     method postfix_inc-> r Integer
     {  r:=num(this);
        this:=<<type Integern>>integer(r+1);
        return r
     }
     method postfix_dec-> r Integer
     {  r:=num(this);
        this:=<<type Integern>>integer(r-1);
        return r
     }
/* simple assignment */
     method "="   (i Integern     ) -> Integer {this:=i;              return num(this)}
```

```
     method "="   (i Integer      ) -> Integer {this:=integer(i);     return num(this)}
     method "="   (i Integer1     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Integer2     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Integer3     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Integer4     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_char     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_short    ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_int      ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_long     ) -> Integer {this:=integer(num(i));return num(this)}
     method "="   (i Int_long_long) -> Integer {this:=integer(num(i));return num(this)}
/* multiplication assignment */
     method "*="  (i Integern     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer      ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer1     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer2     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer3     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Integer4     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_char     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_short    ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_int      ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_long     ) -> Integer {this:=this  * i; return num(this)}
     method "*="  (i Int_long_long) -> Integer {this:=this  * i; return num(this)}
/* division assignment */
     method "/="  (i Integern     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer      ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer1     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer2     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer3     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Integer4     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Int_char     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Int_short    ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Int_int      ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Int_long     ) -> Integer {this:=this  / i; return num(this)}
     method "/="  (i Int_long_long) -> Integer {this:=this  / i; return num(this)}
/* remainder assignment */
     method "%="  (i Integern     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer      ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer1     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer2     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer3     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Integer4     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_char     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_short    ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_int      ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_long     ) -> Integer {this:=this  % i; return num(this)}
     method "%="  (i Int_long_long) -> Integer {this:=this  % i; return num(this)}
/* addition assignment */
     method "+="  (i Integern     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer      ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer1     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer2     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer3     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Integer4     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_char     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_short    ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_int      ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_long     ) -> Integer {this:=this  + i; return num(this)}
     method "+="  (i Int_long_long) -> Integer {this:=this  + i; return num(this)}
/- subtraction assignment -/
     method "-="  (i Integern     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer      ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer1     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer2     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer3     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Integer4     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_char     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_short    ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_int      ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_long     ) -> Integer {this:=this  - i; return num(this)}
     method "-="  (i Int_long_long) -> Integer {this:=this  - i; return num(this)}
/* shift left assignment */
```

```
     method "<<=" (i Integern     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer      ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer1     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer2     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer3     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Integer4     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_char     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_short    ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_int      ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_long     ) -> Integer {this:=this  << i; return num(this)}
     method "<<=" (i Int_long_long) -> Integer {this:=this  << i; return num(this)}
/* shift right assignment */
     method ">>=" (i Integern     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer      ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer1     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer2     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer3     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Integer4     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_char     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_short    ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_int      ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_long     ) -> Integer {this:=this  >> i; return num(this)}
     method ">>=" (i Int_long_long) -> Integer {this:=this  >> i; return num(this)}
/* bitwise and assignment */
     method "&="  (i Integern     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer      ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer1     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer2     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer3     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Integer4     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Int_char     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Int_short    ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Int_int      ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Int_long     ) -> Integer {this:=this  & i; return num(this)}
     method "&="  (i Int_long_long) -> Integer {this:=this  & i; return num(this)}
/* bitwise xor assignment */
     method "^="  (i Integern     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer      ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer1     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer2     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer3     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Integer4     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_char     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_short    ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_int      ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_long     ) -> Integer {this:=this  ^ i; return num(this)}
     method "^="  (i Int_long_long) -> Integer {this:=this  ^ i; return num(this)}
/* bitwise or assignment */
     method "|="  (i Integern     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer      ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer1     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer2     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer3     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Integer4     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_char     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_short    ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_int      ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_long     ) -> Integer {this:=this  | i; return num(this)}
     method "|="  (i Int_long_long) -> Integer {this:=this  | i; return num(this)}
     default integer(0);
} /* end parameterized value type Integern;
```

C.1.6.2.2 Integers for a signed integer type with n bits

The parameters `n`, `Integer1`, `Integer2`, `Integer3` and `Integer4` from `Integern` are left unbound. Each signed integer type is defined using `Integern` with `n` bound to the number of bits, and the other context parameters bound to the other C signed integer types.

```
*/
abstract value type Integers inherits Integern
```

```
     < /* Int_char      = */ Unsigned_char,
       /* Int_short     = */ Unsigned_short,
       /* Int_int       = */ Unsigned_int
       /* Int_long      = */ Unsigned_long,
       /* Int_long_long = */ Unsigned_long_long >
     /* this leaves the following parameters unbound:
       synonym n Natural;
       value type Integer1; value type Integer2;
       value type Integer3; value type Integer4 */
{
     operator integer ( i Integer )-> this Integers
     synonym p Integer = power(2,n-1);
     { /* integer(i) of Integers is the unique constructor for Integers values. */
        return if i > p-1 then integer((i - (i/p)*p)
               /* reduce to integer in range of Integers */
       else
          if i < -p then integer(i - ((i+1)/p)*p)
                /* increase to integer in range of Integers */
          else integer(i) /* which represents the Integers value */
          fi
       fi}
     operator num ( nv this Integers )-> Integer {
     /* return the integer value for which integer(num(nv)) = nv */
     /* Although this is expressed here as an iterative algorithm,
       it is expected that the implementation does the conversion
       as an atomic action,
       or null action if the effect is to just to do a type conversion. */
       return if integer(0) = nv then 0
       else
          if (nv > integer (0)) then 1 + num(nv-<<type Integers>>integer(1));
          else /*(nv < integer (0))*/ -1 + num(nv+<<type Integers>>integer(1))
          fi
       fi}
/* shift left */
     operator "<<" ( nv Integers, i Integer ) -> Integer
     /* sign bit propagated left for << */
       {return num(nv) * power(2, if i > 0 then i else 0 fi)}
     operator "<<" ( i Integer, nv Integers ) -> Integer
       {return i * power(2, if num(nv) > 0 then num(nv) else 0 fi)}
/* shift right */
     operator ">>"( nv Integers, i Integer ) -> Integer {
     /* In clause 6.5.7 of [b-ISO/IEC 9899] for a negative number
       the resulting value is implementation-defined.
       A negative number is -power(2,n) for the sign bit
       plus power(2,m) for each of other 1 bit
       where the bits are numbered m to 0 (most to least significant). */
       return if i <= 0 then num(nv)
       else
          if num(nv) > 0 then num(nv)/power(2,i)
          else /* num(nv) < 0 */ (num(nv)+power(2,n))/ power(2,i)
          fi
       fi}
     operator ">>"( i Integer, nv Integers ) -> Integer {
       return if num(nv) <= 0 then i
       else
          if i >= 0 then i/power(2,num(nv))
          else /* i < 0 */
             (num(<<type Integers>>integer(i))+power(2,n))/ power(2,num(nv));
          fi
       fi}
/* bit wise and */
     operator private bitwise_and ( i1, i2, m Integer ) -> Integer
     dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
     {  if (m=1) return (i1 > 0) and (i2 > 0) then 1 else 0;
       p := power(2,m-1);/*value of bit m */
       r1 = i1 rem p;
       r2 = i2 rem p;
       /* bit m is 1 in i1 if i1-r1 > 0 */
       /* bit m is 1 in i2 if i2-r2 > 0 */
       return if ((i1-r1) > 0) and ((i2-r2) > 0) then p else 0 fi +
```

```
          bitwise_and (r1, r2, m-1)
      }
      operator "&" ( nv Integers, i Integer ) -> Integer
      synonym p Integer = power(2,n);
      dcl nvi Integer;
      {  nvi := num(nv);
         i := num(<<type Integers>>integer(i));
         return if (nvi < 0) and (i <0) then p else 0 /* the sign bit */ fi
            /* add the other bits, in each case removing the sign */
            + bitwise_and( nvi + if nvi < 0 then p else 0 fi,
                  i + if i < 0 then p else 0 fi, n-2)
      }
/* bitwise xor*/
      operator private bitwise_xor ( i1, i2, m Integer ) -> Integer
      dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
      {  if (m=1) return if ((i1 > 0) xor (i2 > 0)) then 1 else 0 fi
         p := power(2,m-1); /*value of bit m */
         r1 = i1 rem p;
         r2 = i2 rem p;
         /* bit m is 1 in i1 if i1-r1 > 0 */
         /* bit m is 1 in i2 if i2-r2 > 0 */
         return if ((i1-r1) > 0) xor ((i2-r2) > 0)  then p else 0 fi +
            bitwise_xor (r1, r2, m-1)
      }
      operator "^" ( nv Integers, i Integer ) -> Integer
      synonym p Integer = power(2,n-1);
      dcl nvi, i_only_bits Integer;
      {  nvi := num(nv);
         i_only_bits := i
            /* subtract bits in range to neutralise these*/
             - num(<<type Integers>>integer(i));
         i := num(<<type Integers>>integer(i));
         return  i_only_bits +
                 if (nvi < 0) xor (i <0 )
                 then p else 0 /* the sign bit */
                 fi + /* add the other bits, in each case removing the sign */
                 bitwise_xor( nvi + if nvi < 0 then p else 0 fi,
                  i+ if i < 0 then p else 0 fi, n-1);
      }
/* bitwise or*/
      operator private bitwise_or ( i1, i2, m Integer ) -> Integer
      dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
      {  if (m=1) return if ((i1 > 0) or (i2 > 0)) then 1 else 0 fi;
         p := power(2,m-1); /*value of bit m*/
         r1 = i1 rem p;
         r2 = i2 rem p;
         /* bit m is 1 in i1 if i1-r1 > 0 */
         /* bit m is 1 in i2 if i2-r2 > 0 */
         return if ((i1-r1) > 0) or ((i2-r2) > 0)  then p else 0 fi
            + bitwise_or (r1, r2, m-1)
      }
      operator "|" ( nv Integers, i Integer ) -> Integer
      synonym p Integer = power(2,n);
      dcl nvi, i_only_bits Integer;
      {  nvi := num(nv);
         i_only_bits := i
            /* subtract bits in range to neutralise these*/
             - num(<<type Integers>>integer(i));
         i := num(<<type Integers>>integer(i));
         return  i_only_bits +
                 if (nvi < 0) or (i <0 )
                 then p else 0 /* the sign bit */
                 fi + /* add the other bits, in each case removing the sign */
                 bitwise_or( nvi + if nvi < 0 then p else 0 fi,
                  i+ if i < 0 then p else 0 fi, n-1);
      }
} /* end parameterized value type Integers;
```

C.1.6.2.3 Integeru for an unsigned integer type with n bits

The parameters `n`, `Integer1`, `Integer2`, `Integer3` and `Integer4` from `Integern` are left unbound. Each signed integer type is defined using `Integeru` with `n` bound to the number of bits, and the other context parameters bound to the other C unsigned integer types.

```
*/
abstract value type Integeru inherits Integern
     <  /* Int_char      = */ Signed_char,
        /* Int_short     = */ Signed_short,
        /* Int_int       = */ Signed_int
        /* Int_long      = */ Signed_long,
        /* Int_long_long = */ Signed_long_long >
     /* this leaves the following parameters unbound:
        synonym n Natural;
        value type Integer1; value type Integer2;
        value type Integer3; value type Integer4 */
{
     operator integer ( i Integer )-> this Integeru
     synonym p Integer = power(2,n-1);
        /* integer(i) of Integeru is the unique constructor for Integeru values. */
     {  return  if i > p-1 then integer((i - (i/p)*p)
                /* reduce to integer in range of Integeru */
        else
           if i < -p then integer(i - ((i+1)/p)*p)
                 /* increase to integer in range of Integeru */
           else integer(i) /* which represents the Integeru value */
           fi
        fi}
     operator num ( nv this Integeru )-> Natural {
     /* return the positive integer value for which integer(num(nv)) = nv */
     /* Although this is expressed here as an iteration algorithm,
        it is expected that the implementation does the conversion
        as an atomic action,
        or null action if the effect is to just to do a type conversion. */
        if (integer(0) = nv) then return 0;
        return 1 + num(nv-<<type Integeru>>integer(1))
     }
/* shift left */
     operator "<<" ( nv Integeru, i Integer ) -> Integer
        {return num(nv) * power(2, if i > 0 then i else 0 fi)}
     operator "<<" ( i Integer, nv Integeru ) -> Integer {return i * power(2 num(nv))}
/* shift right */
     operator ">>"( nv Integeru, i Integer ) -> Integer
        {return if i <= 0 then num(nv) else num(nv)/power(2,i) fi}
     operator ">>"( i Integer, nv Integeru ) -> Integer {
        return if i >= 0
        then i/power(2,num(nv))
        else (num(<<type Integeru>>integer(i))+power(2,n))/ power(2,num(nv))fi
     }
/*bitwise and*/
     operator private bitwise_and ( i1, i2, m Integer ) -> Integer
     dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
     {  if (m=1) return if (i1 > 0) and (i2 > 0) then 1 else 0 fi;
        p := power(2,m-1); /*value of bit m */
        r1 = i1 rem p;
        r2 = i2 rem p;
        /* bit m-1 is 1 in i1 if i1-r1 > 0 */
        /* bit m-1 is 1 in i2 if i2-r2 > 0 */
        return if ((i1-r1) > 0) and ((i2-r2) > 0) then p else 0 fi
          + bitwise_and (r1, r2, m-1)
     }
     operator "&" ( nv Integeru, i Integer ) -> Integer;
        {return bitwise_and(num(nv), num(<<type Integeru>>integer(i)), n)}
/* bitwise xor*/
     operator private bitwise_xor ( i1, i2, m Integer ) -> Integer
     dcl p, r1, r2 Integer; /* least significant bit is bit 1 */
     {  if (m=1) return if ((i1 > 0) xor (i2 > 0)) then 1 else 0 fi;
        p := power(2,m-1); /*value of bit m */
```

```
        r1 = i1 rem p;
        r2 = i2 rem p;
        /* bit m is 1 in i1 if i1-r1 > 0 */
        /* bit m is 1 in i2 if i2-r2 > 0 */
        return if ((i1-r1) > 0) xor ((i2-r2) > 0)  then p else 0 fi
          + bitwise_xor (r1, r2, m-1)
     }
     operator "^" ( nv Integeru, i Integer ) -> Integer;
     dcl i_only_bits Integer;
     {  i_only_bits := i
          /* subtract bits in range to neutralise these*/
           - num(<<type Integeru>>integer(i));
        i := num(<<type Integeru>>integer(i));
        return  i_only_bits +  bitwise_xor(num(nv), i, n);
     }
/* bitwise or*/
     operator private bitwise_or ( i1, i2, m Integer ) -> Integer
     dcl p, r1, r2 Integer;
     {  if (m=1) return if ((i1 > 0) or (i2 > 0)) then 1 else 0 fi;
        p := power(2,m-1); /*value of bit m, least significant bit is bit 1*/
        r1 = i1 rem p;
        r2 = i2 rem p;
        /* bit m is 1 in i1 if i1-r1 > 0 */
        /* bit m is 1 in i2 if i2-r2 > 0 */
        return if ((i1-r1) > 0) or ((i2-r2) > 0)  then p else 0 fi
          + bitwise_or (r1, r2, m-1)
     }
     operator "|" ( nv Integeru, i Integer ) -> Integer;
     dcl i_only_bits Integer;
     {  i_only_bits := i
          /* subtract bits in range to neutralise these*/
           - num(<<type Integeru>>integer(i));
        i := num(<<type Integeru>>integer(i));
        return  i_only_bits + bitwise_or(num(nv), i, n);
     }
} /* end parameterized value type Integeru;
```

C.1.6.3 Types for void and pointers

The `Void` type comprises an empty set of values; it is an incomplete type that cannot be completed. The `Void` type is denoted by a <c type keywords> item **void**. It is not allowed to define a variable or parameter with the type `Void`.

```
*/
abstract value type Void
     /* Has an implicit literal signature denoted by null – see [ITU-T Z.107 12.1] */
endvalue type Void;
synonym sizeof_Void Integer = 0;
/*
```

The `Star_void` type represents a 'pointer to `Void`' (an untyped pointer) such as that introduced by a <c declaration> of the form:

```
void *id;
```

The operators `Star_void_to_Integer` and `Integer_to_Star_void` of `Star_void` convert to and from `Integer`. It is required that:

`Star_void_to_Integer (Integer_to_Star_void (i)) == i` and
`Integer_to_Star_void (Star_void_to_Integer (ptr) == ptr`

Arithmetic operators are not defined for `Star_void` because the size of the pointer is not known.

The dereference operator is not defined for `Star_void` because the actual type of the pointer is not known and therefore the type of the result is not known.

```
*/
value type Star_void {
     /* Has an implicit literal signature denoted by null – see [ITU-T Z.107 12.1] */
     operator Star_void_to_Integer (ptr Star_void) -> Integer external;
     operator Integer_to_Star_void (i Integer) -> Star_void external;
     default null;
} /* end value type Star_void; */
synonym sizeof_Star_void Integer = external /*Depends on the architecture.*/
/*
```

The parameterized type `Star_type` with a type as an actual parameter defines the typed pointer type for the type.

It is possible to convert a `Star_void` (a pointer to **`void`**) to or from a pointer of any type. If a pointer to a type is converted to a `Star_void` (a pointer to **`void`**) and back again, the result compares equal to the original pointer. The operators to convert to each type from `Star_void` and from each type to `Star_void` are defined in the typed pointer type for the type.

The operators `Star_type_to_Integer` and `Integer_to_Star_type` of `Star_type` convert to and from `Integer`. It is required that:

```
       Star_type_to_Integer(Integer_to_Star_type(i))==i      and
       Integer_to_Star_type(Star_type_to_Integer(p))==p      and
       Star_void_to_Integer(Star_type_to_Star_void(p))==Star_type_to_Integer(p)
       and
       Star_void_to_Star_type(Integer_to_Star_void(i))== Integer_to_Star_type(i)
*/
abstract value type Star_type <type Atype, synonym sizeof_type Integer>{
     operator "&" (in/out x Atype) -> Star_type /* address of */
        {return Star_void_to_Star_type(<<type Atype>>"&"(x)}
     operator "*" (p Star_type) -> Atype external; /*dereference the Star_type */
     operator Star_type_to_Integer (p Star_type) -> Integer external;
     operator Integer_to_Star_type (i Integer) -> Star_type external;
     operator Star_type_to_Star_void (p Star_type) -> Star_void
        { return Integer_to_Star_void(Star_type_to_Integer(p)) }
     operator Star_void_to_Star_type (p Star_void) -> Star_type
        { return Integer_to_Star_type(Star_void_to_Integer(p))}
     operator "+" (p Star_type, i Integer) -> Star_type /* pointer arithmetic - add */
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_char   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_short  ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_int    ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_long   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Signed_long_long) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_char   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_short  ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_int    ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_long   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "+" (p Star_type, i Unsigned_long_long) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) + i*sizeof_type) }
     operator "-" (p Star_type, i Integer) -> Star_type /* pointer arithmetic - sub */
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Signed_char   ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Signed_short  ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
     operator "-" (p Star_type, i Signed_int    ) -> Star_type
        { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
```

```
      operator "-" (p Star_type, i Signed_long   ) -> Star_type
         { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
      operator "-" (p Star_type, i Signed_long_long) -> Star_type
         { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
      operator "-" (p Star_type, i Unsigned_char   ) -> Star_type
         { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
      operator "-" (p Star_type, i Unsigned_short  ) -> Star_type
         { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
      operator "-" (p Star_type, i Unsigned_int    ) -> Star_type
         { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
      operator "-" (p Star_type, i Unsigned_long   ) -> Star_type
         { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
      operator "-" (p Star_type, i Unsigned_long_long) -> Star_type
         { return Integer_to_Star_type(Star_type_to_Integer(p) - i*sizeof_type) }
      method "++"  -> Star_type {this := this+1; return this }
      method "--"  -> Star_type {this := this-1; return this }
      method postfix_inc-> r Star_type {r:= this; this:= r+1; return r }
      method postfix_dec-> r Star_type {r:= this; this:= r-1; return r }
      /* simple assignment */
      method "=" (p Star_type)->Star_type {this:=p;return this}
      method "=" (i Integer)->Star_type{this:=Integer_to_Star_type(i);return this}
      method "=" (i Signed_char)->Star_type{this:=Integer_to_Star_type(num(i));return
this}
      method "=" (i Signed_short)->Star_type
         {this:=Integer_to_Star_type(num(i));return this}
      method "=" (i Signed_int)->Star_type{this:=Integer_to_Star_type(num(i));return this}
      method "=" (i Signed_long)->Star_type{this:=Integer_to_Star_type(num(i));return
this}
      method "=" (i Signed_long_long)->Star_type
         {this:=Integer_to_Star_type(num(i));return this}
      method "=" (i Unsigned_char)->Star_type
         {this:=Integer_to_Star_type(num(i));return this}
      method "=" (i Unsigned_short)->Star_type
         {this:=Integer_to_Star_type(num(i));return this}
      method "=" (i Unsigned_int)->Star_type
         {this:=Integer_to_Star_type(num(i));return this}
      method "=" (i Unsigned_long)->Star_type
         {this:=Integer_to_Star_type(num(i));return this}
      method "=" (i Unsigned_long_long)->Star_type
         {this:=Integer_to_Star_type(num(i));return this}
      /* addition assignment */
      method "+=" (i Integer)            -> Star_type {this:=this + i; return this}
      method "+=" (i Signed_char)        -> Star_type {this:=this + i; return this}
      method "+=" (i Signed_short)       -> Star_type {this:=this + i; return this}
      method "+=" (i Signed_int)         -> Star_type {this:=this + i; return this}
      method "+=" (i Signed_long)        -> Star_type {this:=this + i; return this}
      method "+=" (i Signed_long_long)   -> Star_type {this:=this + i; return this}
      method "+=" (i Unsigned_char)      -> Star_type {this:=this + i; return this}
      method "+=" (i Unsigned_short)     -> Star_type {this:=this + i; return this}
      method "+=" (i Unsigned_int)       -> Star_type {this:=this + i; return this}
      method "+=" (i Unsigned_long)      -> Star_type {this:=this + i; return this}
      method "+=" (i Unsigned_long_long) -> Star_type {this:=this + i; return this}
      /* subtraction assignment */
      method "-=" (i Integer)            -> Star_type {this:=this - i; return this}
      method "-=" (i Signed_char)        -> Star_type {this:=this - i; return this}
      method "-=" (i Signed_short)       -> Star_type {this:=this - i; return this}
      method "-=" (i Signed_int)         -> Star_type {this:=this - i; return this}
      method "-=" (i Signed_long)        -> Star_type {this:=this - i; return this}
      method "-=" (i Signed_long_long)   -> Star_type {this:=this - i; return this}
      method "-=" (i Unsigned_char)      -> Star_type {this:=this - i; return this}
      method "-=" (i Unsigned_short)     -> Star_type {this:=this - i; return this}
      method "-=" (i Unsigned_int)       -> Star_type {this:=this - i; return this}
      method "-=" (i Unsigned_long)      -> Star_type {this:=this - i; return this}
      method "-=" (i Unsigned_long_long) -> Star_type {this:=this - i; return this}
      /* Other Assignment methods not defined for pointers. */
      default null;
}/* end parameterized value type Star_type
*/
```

C.1.6.4 C signed and unsigned char types

A C char is an 8-bit integer, also used to represent character.

```
*/
value type Signed_char
     inherits Integers <
        /*n*/ CHAR_BIT /*external, usually 8 */,
        /*Integer1*/ Signed_short,
        /*Integer2*/ Signed_int,
        /*Integer3*/ Signed_long,
        /*Integer4*/ Signed_long_long >
        (to_Signed_char = integer );
synonym sizeof_Signed_char Integer = (CHAR_BIT+7)/8 /*usually 1*/;
value type Star_Signed_char
     inherits Star_type < Signed_char, sizeof_Signed_char>
     {  Star_Signed_char_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Signed_char = Star_void_to_Star_type,
        Star_Signed_char_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Signed_char = Integer_to_Star_type);
synonym sizeof_Star_Signed_char Integer = sizeof_Star_void;
value type Unsigned_char
     inherits Integeru <
        /*n*/ CHAR_BIT /*external, usually 8 */,
        /*Integer1*/ Unsigned_short,
        /*Integer2*/ Unsigned_int,
        /*Integer3*/ Unsigned_long,
        /*Integer4*/ Unsigned_long_long >
        (to_Unsigned_char = integer );
synonym sizeof_Unsigned_char Integer = sizeof_Signed_char;
value type Star_Unsigned_char
     inherits Star_type < Unsigned_char, sizeof_Unsigned_char>
     {  Star_Unsigned_char_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Unsigned_char = Star_void_to_Star_type,
        Star_Unsigned_char_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Unsigned_char = Integer_to_Star_type)
{
     operator Star_Unsigned_char_to_Charstring (sc Star_Unsigned_char)-> cs
Charstring
     {  cs := ""; /*empty string*/
        loop ("*"(sc)<>0) /* while character is not NUL */
        {  cs := cs//mkstring(chr(num("*"(sc))); /* convert and add to result */
           sc := (Star_Unsigned_char_Integer(sc) + sizeof_Signed_char); /*next char
*/
        } /* end of loop body*/
        return cs
     } /*end of Star_Unsigned_char_to_Charstring*/
     operator length(sc Star_Unsigned_char)-> Integer
     { return length(Star_Unsigned_char_to_Charstring(sc))}
} /*end of Star_Unsigned_char
synonym sizeof_Star_Unsigned_char Integer = sizeof_Star_void;
/*
```

C.1.6.5 C signed and unsigned short types

A C short is a 16-bit integer.

```
*/
value type Signed_short
     inherits Integers <
        /*n*/ SHRT_BIT /*external, usually 16 */,
        /*Integer1*/ Signed_char,
        /*Integer2*/ Signed_int,
        /*Integer3*/ Signed_long,
        /*Integer4*/ Signed_long_long >
        (to_Signed_short = integer );
synonym sizeof_Signed_short Integer = (SHRT_BIT+7)/8 /*usually 2*/;
value type Star_Signed_short
     inherits Star_type < Signed_short, sizeof_Signed_short>
```

```
     {  Star_Signed_short_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Signed_short = Star_void_to_Star_type,
        Star_Signed_short_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Signed_short = Integer_to_Star_type);
synonym sizeof_Star_Signed_short Integer = sizeof_Star_void;
value type Unsigned_short
     inherits Integeru <
        /*n*/ SHRT_BIT /*external, usually 16 */,
        /*Integer1*/ Unsigned_char,
        /*Integer2*/ Unsigned_int,
        /*Integer3*/ Unsigned_long,
        /*Integer4*/ Unsigned_long_long >
        (to_Unsigned_short = integer );
synonym sizeof_Unsigned_short Integer = sizeof_Signed_short;
value type Star_Unsigned_short
     inherits Star_type < Unsigned_short, sizeof_Unsigned_short>
     {  Star_Unsigned_short_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Unsigned_short = Star_void_to_Star_type,
        Star_Unsigned_short_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Unsigned_short = Integer_to_Star_type);
synonym sizeof_Star_Unsigned_short Integer = sizeof_Star_void;
/*
```

C.1.6.6 C signed and unsigned int types

A C int is a minimum of 16 bits.

```
*/
value type Signed_int
     inherits Integers <
        /*n*/ INT_BIT /*external, minimum 16 - same as short */,
        /*Integer1*/ Signed_char,
        /*Integer2*/ Signed_short,
        /*Integer3*/ Signed_long,
        /*Integer4*/ Signed_long_long >
        (to_Signed_int = integer );
synonym sizeof_Signed_int Integer = (INT_BIT+7)/8 /*minimum 2*/;
value type Star_Signed_int
     inherits Star_type < Signed_int, sizeof_Signed_int>
     {  Star_Signed_int_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Signed_int = Star_void_to_Star_type,
        Star_Signed_int_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Signed_int = Integer_to_Star_type);
synonym sizeof_Star_Signed_int Integer = sizeof_Star_void;
value type Unsigned_int
     inherits Integeru <
        /*n*/ INT_BIT /*external, minimum 16 */,
        /*Integer1*/ Unsigned_char,
        /*Integer2*/ Unsigned_short,
        /*Integer3*/ Unsigned_long,
        /*Integer4*/ Unsigned_long_long >
        (to_Unsigned_int = integer );
synonym sizeof_Unsigned_int Integer = sizeof_Signed_int;
value type Star_Unsigned_int
     inherits Star_type < Unsigned_int, sizeof_Unsigned_int>
     {  Star_Unsigned_int_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Unsigned_int = Star_void_to_Star_type,
        Star_Unsigned_int_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Unsigned_int = Integer_to_Star_type);
synonym sizeof_Star_Unsigned_int Integer = sizeof_Star_void;
/*
```

C.1.6.7 C signed and unsigned long types

A C long is a minimum of 32 bits.

```
*/
value type Signed_long
     inherits Integers <
        /*n*/ LONG_BIT /*external, minimum 32 - twice int */,
        /*Integer1*/ Signed_char,
```

```
        /*Integer2*/ Signed_short,
        /*Integer3*/ Signed_int,
        /*Integer4*/ Signed_long_long >
        (to_Signed_long = integer );
synonym sizeof_Signed_long Integer = (LONG_BIT+7)/8 /*minimum 4*/;
value type Star_Signed_long
     inherits Star_type < Signed_long, sizeof_Signed_long>
     {  Star_Signed_long_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Signed_long = Star_void_to_Star_type,
        Star_Signed_long_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Signed_long = Integer_to_Star_type);
synonym sizeof_Star_Signed_long Integer = sizeof_Star_void;
value type Unsigned_long
     inherits Integeru <
        /*n*/ LONG_BIT /*external, minimum 32 */,
        /*Integer1*/ Unsigned_char,
        /*Integer2*/ Unsigned_short,
        /*Integer3*/ Unsigned_int,
        /*Integer4*/ Unsigned_long_long >
        (to_Unsigned_long = integer );
synonym sizeof_Unsigned_long Integer = sizeof_Signed_ilong;
value type Star_Unsigned_long
     inherits Star_type < Unsigned_long, sizeof_Unsigned_long>
     {  Star_Unsigned_long_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Unsigned_long = Star_void_to_Star_type,
        Star_Unsigned_long_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Unsigned_long = Integer_to_Star_type);
synonym sizeof_Star_Unsigned_long Integer = sizeof_Star_void;
/*
```

C.1.6.8 C signed and unsigned long long types

A C long long is a minimum of 64 bits.

```
*/
value type Signed_long_long
     inherits Integers <
        /*n*/ LLONG_BIT /*external, minimum 64 - twice long */,
        /*Integer1*/ Signed_char,
        /*Integer2*/ Signed_short,
        /*Integer3*/ Signed_int,
        /*Integer4*/ Signed_long >
        (to_Signed_long_long = integer );
synonym sizeof_Signed_long_long Integer = (LLONG_BIT+7)/8 /*minimum 8*/;
value type Star_Signed_long_long
     inherits Star_type < Signed_long_long, sizeof_Signed_long_long>
     {  Star_Signed_long_long_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Signed_long_long = Star_void_to_Star_type,
        Star_Signed_long_long_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Signed_long_long = Integer_to_Star_type);
synonym sizeof_Star_Signed_long_long Integer = sizeof_Star_void;
value type Unsigned_long_long
     inherits Integeru <
        /*n*/ LLONG_BIT /*external, minimum 64 */,
        /*Integer1*/ Unsigned_char,
        /*Integer2*/ Unsigned_short,
        /*Integer3*/ Unsigned_int,
        /*Integer4*/ Unsigned_long >
        (to_Unsigned_long = integer );
synonym sizeof_Unsigned_long_long Integer = sizeof_Signed_long_long;
value type Star_Unsigned_long_long
     inherits Star_type < Unsigned_long_long, sizeof_Unsigned_long_long>
     {  Star_Unsigned_long_long_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Unsigned_long_long = Star_void_to_Star_type,
        Star_Unsigned_long_long_to_Integer = Star_type_to_Integer,
        Integer_to_Star_Unsigned_long_long = Integer_to_Star_type);
synonym sizeof_Star_Unsigned_long_long Integer = sizeof_Star_void;
/*
```

C.1.6.9 C floating numbers types

C has three different floating types: `Float`, `Double` and `Long_double` with potentially different levels of precision and range of the values. By contrast, `<<package Predefined>>Real` in theory can represent values to any level of precision and an unbounded range. In practice the implementation of floating numbers is (usually) dependent on hardware support and therefore is implementation dependent. The 3 types (`Float`, `Double` and `Long_double`) are therefore considered formally equivalent in this Recommendation except the size of each as defined by synonyms below, so that `Double` and `Long_double` are each defined as a syntype of `Float`.

```
*/
syntype Double = Float {} /* alternative name for Float */
syntype Long_double = Float {} /* alternative name for Float */
synonym sizeof_Float Integer = external /* at least 4 */;
synonym sizeof_Double Integer = external /* at least 8 */;
synonym sizeof_Long_double Integer = external /* at least 16 */;
/*
```

A new value type is introduced for `Float` (rather than define a syntype of `Real`) to define arithmetic operators (such as `"+"` between each of the C integer types and `Float`. An operator such as `"+"` is defined between `Float` and each of the following: `Real`, `Integer`, `Signed_char`, `Unsigned_char`, `Signed_short`, `Unsigned_short`, `Signed_int`, `Unsigned_int`, `Signed_long`, `Unsigned_long`, `Signed_long_long` and `Unsigned_long_long`. The corresponding operator is also defined in each case where the sorts of the parameters are interchanged so the second parameter sort is `Float` and for the case where both parameter sorts are `Float`. The result sort of each arithmetic operator is `Real`. For each such operator name there are therefore 25 signatures.

Comparison operators (`"=="`,`"!="`,`"<"`,`">"`,`"<="`,`">="`) are defined between the pairs of sorts as above, but also for both `Boolean` and `Integer` results. For each such operator name there are therefore 50 signatures.

Assignment expression methods (`"="`,`"*="`,`"/="`,`"+="`,`"-="`) are defined where the right-hand side is a `Float`, `Integer Real` or C integer type and the result is the `Real` value of the updated `Float` variable.

The operators between C integer types and `Float` are defined with the type for the integer rather than below, to shorten the description of `Float`.

Similar to integers, it is assumed that the result of all operators (except comparisons or `"&"` for pointers) involving floating numbers is a `Real`. Also, there are no literals for `Float`: the constructor is the operator

```
to_Float ( Real) -> Float;
```

To shorten the description of `Float` below, for each operation only the <operation definition> is given from which the <operation signature> is constructed (see the *Model* part of clause 12.1.7).

```
value type Float {
     literals unordered; /* "<", ">","<=",">=", first, last, pred, succ,
        and num are not implicitly defined
     operators and methods signature lists are constructed from the definitions
        of the operations below */
     operator to_Real (f Float) -> Real external;
     operator to_Float(r Real) ->Float external; /* constructor */
     /* the casting operators are not formally defined.
        It is assumed that:
           to_Real (to_Float(r)) has the value r, and
           to_Float (to_Real(f)) has the value f. */
     operator Integer_to_Float(i Integer) ->Float {return to_Float(float(i))}
     operator "&" ( f Float )-> Void_star external; /* 'address of' operator.*/
     operator "+"(f Float)-> Real{return  to_Real(f)} /*unary plus */
     operator "-"(f Float)-> Real{return -to_Real(f)} /*unary minus*/
     /* addition */
     operator "+"(f1 Float, f2 Float)-> Real{return to_Real(f1) + to_Real(f2)}
```

```
operator "+"(f Float, r Real)-> Real{return to_Real(f) + r}
operator "+"(r Real, f Float)-> Real{return to_Real(f) + r}
operator "+"(f Float, i Integer)-> Real{return to_Real(f) + float(i)}
operator "+"(i Integer, f Float)-> Real{return to_Real(f) + float(i)}
/* subtraction */
operator "-"(f1 Float, f2 Float)-> Real{return to_Real(f1) - to_Real(f2)}
operator "-"(f Float, r Real)-> Real{return to_Real(f) - r}
operator "-"(r Real, f Float)-> Real{return r - to_Real(f) }
operator "-"(f Float, i Integer)-> Real{return to_Real(f) - float(i)}
operator "-"(i Integer, f Float)-> Real{return float(i) - to_Real(f)}
/* multiplication */
operator "*"(f1 Float, f2 Float)-> Real{return to_Real(f1) * to_Real(f2)}
operator "*"(f Float, r Real)-> Real{return to_Real(f) * r}
operator "*"(r Real, f Float)-> Real{return to_Real(f) * r}
operator "*"(f Float, i Integer)-> Real{return to_Real(f) * float(i)}
operator "*"(i Integer, f Float)-> Real{return to_Real(f) * float(i)}
/* division */
operator "/"(f1 Float, f2 Float)-> Real{return to_Real(f1) / to_Real(f2)}
operator "/"(f Float, r Real)-> Real{return to_Real(f) / r}
operator "/"(r Real, f Float)-> Real{return r / to_Real(f)}
operator "/"(f Float, i Integer)-> Real{return to_Real(f) / float(i)}
operator "/"(i Integer, f Float)-> Real{return to_Real(i) / float(f)}
/* equal to - Boolean result */
operator "=="(f1 Float,f2 Float)-> Boolean{return to_Real(f1)=to_Real(f2)}
operator "=="(f Float, r Real)-> Boolean{return to_Real(f) = r}
operator "=="(r Real, f Float)-> Boolean{return r = to_Real(f)}
operator "=="(f Float, i Integer)-> Boolean{return to_Real(f) = float(i)}
operator "=="(i Integer, f Float)-> Boolean{return float(i) = to_Real(f)}
/* not equal to - Boolean result */
operator "!="(f1 Float,f2 Float)->Boolean{return to_Real(f1)/=to_Real(f2)}
operator "!="(f Float, r Real)-> Boolean{return to_Real(f) /= r}
operator "!="(r Real, f Float)-> Boolean{return r /= to_Real(f)}
operator "!="(f Float, i Integer)-> Boolean{return to_Real(f) /= float(i)}
operator "!="(i Integer, f Float)-> Boolean{return float(i) /= to_Real(f)}
/* less than - Boolean result */
operator "<"(f1 Float,f2 Float)-> Boolean{return to_Real(f1)<to_Real(f2)}
operator "<"(f Float, r Real)-> Boolean{return to_Real(f) < r}
operator "<"(r Real, f Float)-> Boolean{return r < to_Real(f)}
operator "<"(f Float, i Integer)-> Boolean{return to_Real(f) < float(i)}
operator "<"(i Integer, f Float)-> Boolean{return float(i) < to_Real(f)}
/* greater than - Boolean result */
operator ">"(f1 Float,f2 Float)-> Boolean{return to_Real(f1)>to_Real(f2)}
operator ">"(f Float, r Real)-> Boolean{return to_Real(f) > r}
operator ">"(r Real, f Float)-> Boolean{return r > to_Real(f)}
operator ">"(f Float, i Integer)-> Boolean{return to_Real(f) > float(i)}
operator ">"(i Integer, f Float)-> Boolean{return float(i) > to_Real(f)}
/* less than or equal - Boolean result */
operator"<="(f1 Float,f2 Float)-> Boolean{return to_Real(f1)<=to_Real(f2)}
operator"<="(f Float, r Real)-> Boolean{return to_Real(f) <= r}
operator"<="(r Real, f Float)-> Boolean{return r <= to_Real(f)}
operator"<="(f Float, i Integer)-> Boolean{return to_Real(f) <= float(i)}
operator"<="(i Integer, f Float)-> Boolean{return float(i) <= to_Real(f)}
/* greater than or equal - Boolean result */
operator">="(f1 Float,f2 Float)-> Boolean{return to_Real(f1)>=to_Real(f2)}
operator">="(f Float, r Real)-> Boolean{return to_Real(f) >= r}
operator">="(r Real, f Float)-> Boolean{return r >= to_Real(f)}
operator">="(f Float, i Integer)-> Boolean{return to_Real(f) >= float(i)}
operator">="(i Integer, f Float)-> Boolean{return float(i) >= to_Real(f)}
/* equal to - Integer result */
operator "=="(f1 Float,f2 Float)-> Integer
   {return if to_Real(f1)=to_Real(f2)then 1 else 0 fi}
operator "=="(f Float, r Real)-> Integer{return if to_Real(f) = rthen 1 else 0 fi}
operator "=="(r Real, f Float)-> Integer{return if r = to_Real(f)then 1 else 0 fi}
operator "=="(f Float, i Integer)-> Integer
   {return if to_Real(f) = float(i)then 1 else 0 fi}
```

```
      operator "=="(i Integer, f Float)-> Integer
         {return if float(i) = to_Real(f)then 1 else 0 fi}
      /* not equal to - Integer result */
      operator "!="(f1 Float,f2 Float)->Integer
         {return if to_Real(f1)/=to_Real(f2)then 1 else 0 fi}
      operator "!="(f Float, r Real)-> Integer{return if to_Real(f) /= rthen 1 else 0
fi}
      operator "!="(r Real, f Float)-> Integer{return if r /= to_Real(f)then 1 else 0
fi}
      operator "!="(f Float, i Integer)-> Integer
         {return if to_Real(f) /= float(i)then 1 else 0 fi}
      operator "!="(i Integer, f Float)-> Integer
         {return if float(i) /= to_Real(f)then 1 else 0 fi}
      /* less than - Integer result */
      operator "<"(f1 Float,f2 Float)-> Integer
         {return if to_Real(f1)<to_Real(f2) then 1 else 0 fi}
      operator "<"(f Float, r Real)-> Integer{return if to_Real(f) < r then 1 else 0 fi}
      operator "<"(r Real, f Float)-> Integer{return if r < to_Real(f) then 1 else 0 fi}
      operator "<"(f Float, i Integer)-> Integer
         {return if to_Real(f) < float(i) then 1 else 0 fi}
      operator "<"(i Integer, f Float)-> Integer
         {return if float(i) < to_Real(f) then 1 else 0 fi}
      /* greater than - Integer result */
      operator ">"(f1 Float,f2 Float)-> Integer
         {return if to_Real(f1)>to_Real(f2) then 1 else 0 fi}
      operator ">"(f Float, r Real)-> Integer{return if to_Real(f) > r then 1 else 0 fi}
      operator ">"(r Real, f Float)-> Integer{return if r > to_Real(f) then 1 else 0 fi}
      operator ">"(f Float, i Integer)-> Integer
         {return if to_Real(f) > float(i) then 1 else 0 fi}
      operator ">"(i Integer, f Float)-> Integer
         {return if float(i) > to_Real(f) then 1 else 0 fi}
      /* less than or equal - Integer result */
      operator"<="(f1 Float,f2 Float)-> Integer
         {return if to_Real(f1)<=to_Real(f2) then 1 else 0 fi}
      operator"<="(f Float, r Real)-> Integer{return if to_Real(f) <= r then 1 else 0
fi}
      operator"<="(r Real, f Float)-> Integer{return if r <= to_Real(f) then 1 else 0
fi}
      operator"<="(f Float, i Integer)-> Integer
         {return if to_Real(f) <= float(i) then 1 else 0 fi}
      operator"<="(i Integer, f Float)-> Integer
         {return if float(i) <= to_Real(f) then 1 else 0 fi}
      /* greater than or equal - Integer result */
      operator">="(f1 Float,f2 Float)-> Integer
         {return if to_Real(f1)>=to_Real(f2) then 1 else 0 fi}
      operator">="(f Float, r Real)-> Integer{return if to_Real(f) >= r then 1 else 0
fi}
      operator">="(r Real, f Float)-> Integer{return if r >= to_Real(f) then 1 else 0
fi}
      operator">="(f Float, i Integer)-> Integer
         {return if to_Real(f) >= float(i) then 1 else 0 fi}
      operator">="(i Integer, f Float)-> Integer
         {return if float(i) >= to_Real(f) then 1 else 0 fi}
      /* Simple assignment */
      method "="   (f Float  ) -> Real {this:=f;                  return to_Real(this)}
      method "="   (r Real   ) -> Real {this:=to_Float(r);        return to_Real(this)}
      method "="   (i Integer) -> Real {this:=to_Float(float(i)); return to_Real(this)}
      method "="   (i Signed_char) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method "="   (i Signed_short) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method "="   (i Signed_int) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method "="   (i Signed_long) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
      method "="   (i Signed_long_long) -> Real
         {this:=to_Float(float(num(i))); return to_Real(this)}
```

```
    method "="  (i Unsigned_char) -> Real
       {this:=to_Float(float(num(i))); return to_Real(this)}
    method "="  (i Unsigned_short) -> Real
       {this:=to_Float(float(num(i))); return to_Real(this)}
    method "="  (i Unsigned_int) -> Real
       {this:=to_Float(float(num(i))); return to_Real(this)}
    method "="  (i Unsigned_long) -> Real
       {this:=to_Float(float(num(i))); return to_Real(this)}
    method "="  (i Unsigned_long_long) -> Real
       {this:=to_Float(float(num(i))); return to_Real(this)}
    /* multiplication assignment */
    method "*=" (f Float)              -> Real {this:=this * f; return to_Real(this)}
    method "*=" (r Real)               -> Real {this:=this * r; return to_Real(this)}
    method "*=" (i Integer)            -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Signed_char)        -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Signed_short)       -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Signed_int)         -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Signed_long)        -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Signed_long_long)   -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Unsigned_char)      -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Unsigned_short)     -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Unsigned_int)       -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Unsigned_long)      -> Real {this:=this * i; return to_Real(this)}
    method "*=" (i Unsigned_long_long) -> Real {this:=this * i; return to_Real(this)}
    /* division assignment */
    method "/=" (f Float)              -> Real {this:=this / f; return to_Real(this)}
    method "/=" (r Real)               -> Real {this:=this / r; return to_Real(this)}
    method "/=" (i Integer)            -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Signed_char)        -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Signed_short)       -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Signed_int)         -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Signed_long)        -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Signed_long_long)   -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Unsigned_char)      -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Unsigned_short)     -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Unsigned_int)       -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Unsigned_long)      -> Real {this:=this / i; return to_Real(this)}
    method "/=" (i Unsigned_long_long) -> Real {this:=this / i; return to_Real(this)}
    /* addition assignment */
    method "+=" (f Float)              -> Real {this:=this + f; return to_Real(this)}
    method "+=" (r Real)               -> Real {this:=this + r; return to_Real(this)}
    method "+=" (i Integer)            -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Signed_char)        -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Signed_short)       -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Signed_int)         -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Signed_long)        -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Signed_long_long)   -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Unsigned_char)      -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Unsigned_short)     -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Unsigned_int)       -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Unsigned_long)      -> Real {this:=this + i; return to_Real(this)}
    method "+=" (i Unsigned_long_long) -> Real {this:=this + i; return to_Real(this)}
    /* subtraction assignment */
    method "-=" (f Float)              -> Real {this:=this - f; return to_Real(this)}
    method "-=" (r Real)               -> Real {this:=this - r; return to_Real(this)}
    method "-=" (i Integer)            -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Signed_char)        -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Signed_short)       -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Signed_int)         -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Signed_long)        -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Signed_long_long)   -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Unsigned_char)      -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Unsigned_short)     -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Unsigned_int)       -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Unsigned_long)      -> Real {this:=this - i; return to_Real(this)}
    method "-=" (i Unsigned_long_long) -> Real {this:=this - i; return to_Real(this)}
    /* Assignment methods with names "%=" (remainder), "<<=" (shift left),
       ">>=" (shift right), "&=" (logical and>, "^=" (logical xor),
       and "|=" (logical or) are not defined for Float */
    default to_Float(0.0);
} /* end value type Float*/
```

```
value type Star_Float
     inherits Star_type < Float, sizeof_Float >
     {  Star_Float_to_Star_void = Star_type_to_Star_void,
        Star_void_to_Star_Float = Star_void_to_Star_type);
synonym sizeof_Star_Float Integer = sizeof_Star_void;
/*
```

C.1.6.10 Parameterized type for arrays Cvector

Cvector is used to define the types of arrays.

```
*/
value type Cvector < type Itemsort; synonym MaxIndex Integer >
     inherits Array< Indexsort, Itemsort >
     {syntype Indexsort = Integer constants 0:MaxIndex-1;}
/* end value type Cvector
```

C.1.6.11 Package end

```
*/
endpackage C_Predefined;
/* */
```

Bibliography

[b-ISO/IEC 9899] ISO/IEC 9899:2011, *Information technology – Programming languages – C.*

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	Tariff and accounting principles and international telecommunication/ICT economic and policy issues
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant
Series M	Telecommunication management, including TMN and network maintenance
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling, and associated measurements and tests
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2019