International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100
Annex F3
(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 – Overview of SDL 2010

Annex F3: SDL-2010 formal definition: Dynamic semantics

Recommendation ITU-T Z.100 – Annex F3

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

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

Recommendation ITU-T Z.100

Specification and Description Language – Overview of SDL-2010

Annex F3

SDL-2010 formal definition: Dynamic semantics

Summary

This annex defines the SDL-2010 dynamic semantics.

History

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

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

3.3	ITU-T Z.100 Annex F1	1993-03-12	X	11.1002/1000/3157
3.4	ITU-T Z.100 Annex F2	1993-03-12	X	11.1002/1000/3158
3.5	ITU-T Z.100 Annex F3	1993-03-12	X	11.1002/1000/3159
3.6	ITU-T Z.100 App. I	1993-03-12	X	11.1002/1000/3160
3.7	ITU-T Z.100 App. II	1993-03-12	X	11.1002/1000/3161
4.0	ITU-T Z.100	1993-03-12	X	11.1002/1000/3154
4.1	ITU-T Z.100 (1993) Add. 1	1996-10-18	10	11.1002/1000/3917
5.0	ITU-T Z.100	1999-11-19	10	11.1002/1000/4764
5.1	ITU-T Z.100 (1999) Cor. 1	2001-10-29	17	11.1002/1000/5567
6.0	ITU-T Z.100	2002-08-06	17	11.1002/1000/6029
6.1	ITU-T Z.100 (2002) Amd. 1	2003-10-29	17	11.1002/1000/7091
6.2	ITU-T Z.100 (2002) Cor. 1	2004-08-29	17	11.1002/1000/356
7.0	ITU-T Z.100	2007-11-13	17	11.1002/1000/9262
8.0	ITU-T Z.100	2011-12-22	17	11.1002/1000/11387
8.1	ITU-T Z.100 Annex F1	2000-11-24	10	11.1002/1000/5239
8.2	ITU-T Z.100 Annex F2	2000-11-24	10	11.1002/1000/5576
8.3	ITU-T Z.100 Annex F3	2000-11-24	10	11.1002/1000/5577
8.4	ITU-T Z.100 Annex F1	2015-01-13	17	11.1002/1000/12354
8.5	ITU-T Z.100 Annex F2	2015-01-13	17	11.1002/1000/12355
8.6	ITU-T Z.100 Annex F3	2015-01-13	17	11.1002/1000/12356
9.0	ITU-T Z.100	2016-04-29	17	11.1002/1000/12846
9.1	ITU-T Z.100 Annex F1	2016-10-29	17	11.1002/1000/13040
9.2	ITU-T Z.100 Annex F2	2016-10-29	17	11.1002/1000/13041
9.3	ITU-T Z.100 Annex F3	2016-10-29	17	11.1002/1000/13042
9.4	ITU-T Z.100 Annex F1	2018-11-13	17	11.1002/1000/13732
9.5	ITU-T Z.100 Annex F2	2018-11-13	17	11.1002/1000/13733
9.6	ITU-T Z.100 Annex F3	2018-11-13	17	11.1002/1000/13734
10.0	ITU-T Z.100	2019-10-14	17	11.1002/1000/14048
10.1	ITU-T Z.100 Annex F1	2019-10-14	17	11.1002/1000/14049
10.2	ITU-T Z.100 Annex F2	2019-10-14	17	11.1002/1000/14050
10.3	ITU-T Z.100 Annex F3	2019-10-14	17	11.1002/1000/14051

Keywords

Behaviour semantics, compilation function, data semantics, dynamic semantics, formal definition, SDL-2010, SDL 2010 abstract machine, specification and description language, SAM, SAM programs.

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

Table of Contents

Recommendation ITU-T Z.100

Specification and Description Language – Overview of SDL-2010

Annex F3

SDL-2010 formal definition: Dynamic semantics

F3.1 General information

An overview of the formal semantics is described in clause F1.2 (Annex F1).

F3.1.1 Definitions from Annex F1

The following definitions for the syntax and semantics of ASMs are used within Annex F3. The domains and functions are defined in Annex F1 and listed here for cross-referencing reasons.

> The keywords **case**, **choose**, **constraint**, **controlled**, **derived**, **do**, **domain**, **else**, **elseif**, **endcase**, **endchoose**, **enddo**, **endextend**, **endif**, **endlet**, **endwhere**, **extend**, **forall**, **if**, **initially**, **let**, **monitored**, **of**, **shared**, **static**, **then**, **where**, **with.**
>
> The domains *AGENT*, *BOOLEAN*, *DEFINITIONAS1*, *INT*, *NAT*, *REAL*, *TOKEN*, *X*.
>
> The functions *empty*, *false*, *head*, *isAncestorAS1*, *last*, *length*, *parentAS1*, *parentAS1ofKind*, *program*, *rootNodeAS1*. *Self*, *substring*, *tail*, *take*, *toSet*, *true*, *undefined*.
>
> The operation symbols *, +, **-set**, **-seq**, =, ≠, ∧, ∨, ⇒, ⇔, ¬, ∃, ∀, >, ≥, <, ≤, +, -, *, /, **in**, ×, ⁀, ∪, ∩, \, ∈, ∉, ⊆, ⊂, | |, ⋃, ∅, **mk-**, **s-**, **s1-**, **s2-**, **s3-**.

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

F3.1.2 Definitions from Annex F2

The function *asSignal1* makes the data type *Identifier* for <as signal> allocated in F2 available.

> *asSignal1*: *Identifier* → *Identifier* // signal Identifier to sort Identifier

ENTITYDEFINITION1: the domain defined in F2 that is the union of all the entity definitions in AS1 and it is therefore a subset of *DEFINITIONAS1*.

ENTITYDEFINITION1$=_{def}$ *Agent-definition*
- ∪ *Agent-type-definition*
- ∪ *Channel-definition*
- ∪ *Composite-state-type-definition*
- ∪ *Data-type-definition*
- ∪ *Gate-definition*
- ∪ *Interface-definition*
- ∪ *Literal-signature*
- ∪ *Operation-signature*
- ∪ *Package-definition*
- ∪ *Parameter*
- ∪ *Procedure-definition*
- ∪ *Signal-definition*
- ∪ *State-node*
- ∪ *Syntype-definition*
- ∪ *Timer-definition*
- ∪ *Variable-definition*
- ∪ *Value-data-type-definition*

ENTITYKIND1: the domain defined in F2 that is a set of all the entity kinds in AS1.

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

The function *gcd* (used to normalise Real values) returns the natural number that is the greatest common divisor of the two integer numbers given as parameters.

gcd: *INT* × *INT* → *NAT*

The function *getEntityDefinition1* from Annex F2 gets the entity AS1 definition for an identifier and entity kind:

getEntityDefinition1: *Identifier*, *ENTITYKIND1*→*ENTITYDEFINITION1*

Given an *ENTITYDEFINITION1*, the corresponding *Identifier* is retrieved using the function *identifier1* from Annex F2:

identifier1: *ENTITYDEFINITION1*→ *Identifier*

The function *idKind1* from Annex F2 determines the kind of the entity from the identifier:

idKind1:*Identifier*→*ENTITYKIND1*

The function *isConstant1* is used to determine whether an *EXPRESSION* is a *Constant-expression* (note: *Expression* in F2 is equivalent to the domain *EXPRESSION* in F3).

isConstant1: *EXPRESSION* → *BOOLEAN*

The Boolean function *isevenNat* determines if the given natural is even.

isevenNat : *NAT* → *BOOLEAN* $=_{def}$

Given two definitions, whether one is a supertype of the other is determined using the function *isSuperType* from F2:

isSuperType: *ENTITYDEFINITION1* × *ENTITYDEFINITION1* → *BOOLEAN*

F3.1.3 Function definitions on AS1

Given an *Identifier*, the function *qualifierWithinId1* provides the sequence of qualifier names for any item defined directly within the identified item. The domain *PATHITEM* is defined in clause F3.3.1 and represents one of the constructors *Agent-qualifier*, *Agent-type-qualifier*, *State-type-qualifier*, *Compound-node-qualifier*, *Interface-qualifier*, *Package-qualifier*, *State-qualifier* or *Data-type-qualifier*.

qualifierWithinId1(*id*: *Identifier*): *PATHITEM** $=_{def}$
id.**s**-*PATHITEM*-**seq**
⌢

case *id.refersto1* **of**
| *Agent-definition* **then mk**-*Agent-qualifier*(*id*.**s**-*Name*)
| *Agent-type-definition* **then mk**-*Agent-type-qualifier*(*id*.**s**-*Name*)
| *Composite-state-type-definition* **then mk**-*State-type-qualifier*(*id*.**s**-*Name*)
| *Compound-node* **then mk**-*Compound-node-qualifier*(*id*.**s**-*Name*)
| *Interface-definition* **then mk**-*Interface-qualifier*(*id*.**s**-*Name*)
| *Package-definition* **then mk**-*Package-qualifier*(*id*.**s**-*Name*)
| *Procedure-definition* **then mk**-*Procedure-qualifier*(*id*.**s**-*Name*)
| *State-machine* **then mk**-*State-qualifier*(*id*.**s**-*Name*)
| *State-node* **then mk**-*State-qualifier*(*id*.**s**-*Name*)
| *State-partition* **then mk**-*State-qualifier*(*id*.**s**-*Name*)
| *Value-data-type-definition* **then mk**-*Data-type-qualifier*(*id*.**s**-*Name*)
endcase

Given an *Identifier*, the corresponding *ENTITYDEFINITION1* is retrieved using the function *refersto1*: that is derived from the application of functions *getEntityDefinition1* and *idKind1* from Annex F2.

refersto1(*id*: *Identifier*): [*ENTITYDEFINITION1*]=$_{\text{def}}$
getEntityDefinition1(*id, idKind1*(*id*))

F3.1.4 Status of Annex F3 (this annex)

The ASM in the (01/2015) edition had been updated to correct errors in the earlier (01/2000) edition and to reflect the features of SDL-2010 compared with SDL-2000. The ASM was not complete in the (01/2000) edition. For example, the (01/2000) edition mentions the function *objectsAssign* and the macro SETOBJECTS, but the definitions of these items were not included. While the (01/2015) edition was an improvement on the previous edition, some items still needed further work, in particular adding the treatment of an *Aggregation-kind* of **REF** (see [ITU-T Z.107]) that replaces **object** data types.

The work on Annex F between the (01/2015) and the (11/2018) edition focused on the static semantics in Annex F2, therefore there was not been much change in either the (10/2016) or (11/2018) editions of Annex F3 and further study was needed (denoted by "Further study needed …" items in the (11/2018) text).

When interpreting the abstract grammar, the difference between SDL-2000 and SDL-2010 is not very significant, because many of added SDL-2010 features are either in *Model* clauses that disappear in transformations in Annex F2, or map from the SDL-2010 concrete grammar (from AS0 in Annex F2) to abstract grammar that is unchanged from SDL-2000. Therefore the work to update Annex F3 for SDL-2010 was relatively simple compared to updating Annex F2. In addition to **REF** *Aggregation-kind*, some other SDL-2010 features handled are state timers, gates on input nodes, handling of availability time, and encode/decode with expressions/output. In the handling of inputs some of the description of the dynamic semantics was still based on SDL-2000 in the (10/2016) or (11/2018) editions. For the current edition, checks were made:

- that the functions *compile* and *compileExpr* cover the AS1 rules defined using :: (the ones that add syntax nodes to an abstract syntax tree) and that the syntax nodes they refer to have the current structure;
- that domain definitions used in F3: if not defined in F3, are defined in F1/F2 and they have the expected meanings;
- on the use of *uniqueLabel*; that the index picks out one syntax node from several of the same kind within a given pattern;
- that signals are received at the appropriate destinations.

A decision was made not to include the ASN1 related data types (NumericString, PrintableString, VideotexString, UniversalChar, UniversalCharString, GeneralCharString, VisibleString, BMPCharString and the syntypes NumericChar, PrintableChar, TeletexChar, VideotexChar, IA5Char, IA5String, UTF8String, GraphicChar, VisibleChar, and BMPChar) in F3. The rationale for this decision was that formalisation of these additional data types would follow the patterns used for data types that are included, and the resources (in terms of time, effort and size of the F3) were not justified by the marginal benefit of adding these data types.

F3.2 Behaviour semantics

This clause defines the following parts of the dynamic semantics:

- the SAM (SDL-2010 Abstract Machine): clause F3.2.1;
- the compilation function: clause F3.2.2; and
- SAM programs: clause F3.2.3.

An overview of the dynamic semantics is given in clause F1.2.4 (Annex F1).

F3.2.1 SDL-2010 abstract machine definition (SAM)

The SAM constitutes a generic behaviour model for SDL-2010 specifications. According to an abstract operational view, the possible computations of a given SDL-2010 specification are defined in terms of ASM runs. The underlying semantic model of distributed real-time ASMs is explained in Annex F1. The SAM definition consists of the following four main building blocks:

- signal flow related definitions: clause F3.2.1.1;
- SDL-2010 agent-related definitions: clause F3.2.1.2;
- the interface to the data semantics: clause F3.2.1.3; and
- behaviour primitives: clause F3.2.1.4.

These definitions, in particular, also state explicitly the various constraints on initial SAM states complementing the behaviour model.

F3.2.1.1 Signal flow model

This clause introduces the signal flow model as part of the SAM. The main focus here is on a uniform treatment of signal flow aspects, in particular, on defining how *agents* communicate through *signals* via *gates*. Also, *timers* (clause F3.2.1.1.5), which are modelled as special kinds of signals, are treated here.

F3.2.1.1.1 Signals

PLAINSIGNAL represents the set of *signal types* as declared by an SDL-2010 specification.

PLAINSIGNAL $=_{\text{def}}$ *Identifier* $\cup$ **NONE**

In an SDL-2010 specification, also timers (clause F3.2.1.1.5) are considered as signals; they are contained in a common domain *SIGNAL*

SIGNAL $=_{\text{def}}$ *PLAINSIGNAL* $\cup$ *TIMER*

Dynamically created *plain signal instances* (*plain signals* for short) are elements of a dynamic domain *PLAINSIGNALINST*. Since plain signals can also be created and sent by the environment, this domain is shared. The function *plainSignalType* gives the *signal type* for a given *plain signal instance*.

shared domain *PLAINSIGNALINST*
initially *PLAINSIGNALINST* = ∅

shared *plainSignalType*: *PLAINSIGNALINST* → *PLAINSIGNAL*

The domain *SIGNALINST* contains all kinds of signal instances (*signals* for short). Each element of *SIGNALINST* is uniquely related to an element of *SIGNAL*, as defined by the derived function *signalType*.

SIGNALINST $=_{\text{def}}$ *PLAINSIGNALINST* $\cup$ *TIMERINST*

signalType(*si*:*SIGNALINST*): *SIGNAL* $=_{\text{def}}$
if *si* ∈ *PLAINSIGNALINST* **then** *si*.*plainSignalType*
elseif *si* ∈ *TIMERINST* **then** *si*.**s**-*TIMER*
endif

The functions *plainSignalSender* (giving the sender agent) and *signalSender* (giving the sender of the signal or the agent for the timer) are defined:

shared *plainSignalSender*: *PLAINSIGNALINST* → *PID*

signalSender(*si*:*SIGNALINST*): *PID* $=_{\text{def}}$

if *si* ∈ *PLAINSIGNALINST* **then** *si.plainSignalSender*
elseif *si* ∈ *TIMERINST* **then** *si*.**s**-*PID*
endif

With each signal a (possibly empty) list of signal values is associated and is given by the function *plainSignalValues*. Values are represented in a uniform way as elements of the static domain *VALUE* (see clause F3.2.1.3):

shared *plainSignalValues*: *PLAINSIGNALINST* → *VALUE**

Each plain signal instance has an associated availability time derived from the activation delay specified in the output of the signal and given by the function *availabilityTime*.

shared *availabilityTime*: *PLAINSIGNALINST* → *TIME*

Each plain signal instance has an associated signal priority from the signal priority specified in the output of the signal that is used to decide between signals that have the same availability time, and is given by the function *signalPriority*.

shared *signalPriority*: *PLAINSIGNALINST* → *NAT*

SDL-2010 provides for two forms of indicating the receiver of a message, where the receiver may also remain unspecified.

VIAARG =$_{\text{def}}$ *Identifier*-**set**

TOARG =$_{\text{def}}$ *PID* ∪ *Identifier*

Additional functions on plain signals are *toArg* (giving the destination) and *viaArg* (giving optional constraints on admissible communication paths).

Signals received at an input gate of an agent set are appended to the input port of an agent instance depending on the value of *toArg*. Signals are discarded whenever no matching receiver instance exists.

The value of type *PID* is evaluated dynamically and associated with the label.

shared *toArg*: *PLAINSIGNALINST* → [*TOARG*]

shared *viaArg*: *PLAINSIGNALINST*→ *VIAARG*

A plain signal arriving at an SDL-2010 agent is initially placed in the arrival *GATE* at the end of the communication path. When the signal instance is delivered to the input port (*inport GATE*) of the agent, the signal instance is deleted from the arrival *GATE*, and the value of function *arrivalGate* for the plain signal is set to the arrival *GATE* (see clause F3.2.3.2.1, Agent set execution), so that the correct transition is selected if the input has a via gate.

controlled *arrivalGate*: *PLAINSIGNALINST*→ *GATE*

F3.2.1.1.2 Gates

Exchange of signals between SDL-2010 agents (such as processes, blocks or a system) and the environment is modelled by means of *gates* from a controlled domain *GATE*.

controlled domain *GATE*
initially *GATE* = ∅

A gate forms an interface for *serial* and *unidirectional* communication between two or more agents. Accordingly, gates are either classified as *input gates* or *output gates* (see clause F3.2.1.2.4).

DIRECTION =$_{\text{def}}$ { *inDir*, *outDir* }

controlled *direction*: *GATE* → *DIRECTION*

controlled *myAgent*: *GATE* → *AGENT*

Global system time

In SDL-2010, the *global system time* is represented by the expression **now** assuming that values of **now** increase monotonically over system runs. In particular, SDL-2010 allows having the same value of **now** in two or more consecutive system states. Building on the concept of distributed real-time ASM, this behaviour is modelled using a nullary, dynamic, monitored function *now*. Intuitively, *now* refers to internally observable values of the global system time.

TIME =$_{def}$ *REAL*

monitored *now*: → *TIME*

There are two integrity constraints on the behaviour of *now*:

1. *now* values change monotonically, increasing over ASM runs;
2. *now* values do not increase as long as a signal is in transit on a non-delaying channel.

Discrete delay model

Signals need not reach their destination instantaneously, but may be subject to delays, which means, it is possible to send signals to arrive in the future. Although those signals are not available at their destination before their arrival time has come, they are to be associated with their destination gates. A gate has to be capable of holding signals that are in transit (not yet arrived). Hence, to each gate a possibly empty *signal queue* is assigned, as detailed below.

To model signal arrivals at specified destination gates, each signal instance *si* has an individual arrival time (*si.arrival*) determining the time at which *s* eventually reaches a certain gate.

shared *arrival*: *SIGNALINST* → *TIME*

The relation between signals and gates in a given SAM state is represented by means of a dynamic function *schedule* defined on gates:

shared *schedule*: *GATE* → *SIGNALINST**

where *schedule* specifies, for each gate *g* in *GATE*, the corresponding *signal arrivals* at *g*.

An integrity constraint on *g.schedule* is that signals in *g.schedule* are linearly ordered by their arrival times. That is, if *g.schedule* contains signals *si*, *si'*, and *si.arrival* < *si'.arrival*, then *si* < *si'* in the order as imposed by *g.schedule*. This condition is assured by the *insert* function below.

Waiting signals

A signal instance *si* in *g.schedule* does not arrive "physically" at gate *g* before *now* ≥ *si.arrival*. Intuitively, that means that *s* remains "invisible" at *g* as long as it is in transit. Thus, in every given SAM state, part of *g.schedule* forms a possibly empty signal queue *g.queue*, where *g.queue* represents those signal instances *si* in *g.schedule* that have already arrived at *g* but are still waiting to be removed from *g.schedule*. The *g.queue* is formally defined as follows.

queue(*g*: *GATE*): *SIGNALINST** =$_{def}$ < *si* **in** *g.schedule*: (*now* ≥ *si.arrival*) >

See also Figure F3-1 below for an overview of the functions on schedules.

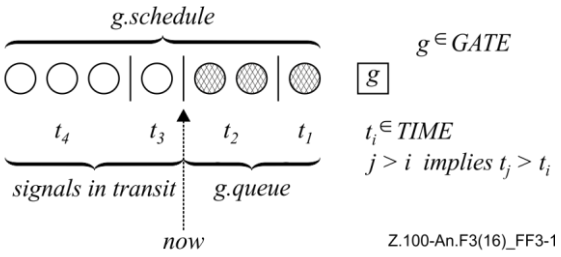

Figure F3-1 – Signal instances at a gate

Operations on schedules

The function *insert* defines the result of inserting some signal instance *si* with the intended arrival time *t* into a finite signal instance list *siSeq*, representing (for example) the schedule of a gate. Except when called from DELIVERSIGNALS in clause F3.2.3.2.1, the time parameter is the arrival time at the given gate. From DELIVERSIGNALS in clause F3.2.3.2.1, the time parameter is the availability time if this is later than the arrival time at the given gate.

As defined in clause 9 *Semantics* of [ITU-T Z.101]: Two or more signals are arbitrarily ordered, if they have the same availability time and same signal priority. For signals that do not convey an availability time and arrive on different paths it is possible that the sequence of the arrival events is not determined (they are "simultaneous") and therefore the signals have the same availability time. For a signal that does not convey an availability time it is also possible that the arrival time (and therefore availability time) of the signal has the same availability time of a signal that previously arrived with an availability time. It is also possible that two signals with availability time have the same availability time. Signal instances that are "simultaneous" and convey different signal priorities are ordered according to the signal priority value, so that signals with lower values are before signals with higher values.

insert(*si*:*SIGNALINST*, *t*:*TIME*, *siSeq*:*SIGNALINST**): *SIGNALINST** =$_{def}$
if *siSeq* = *empty* **then**
< *si* >
elseif *t* < *siSeq.head.arrival* // note that siSeq is not empty
∨ (*t* = *siSeq.head.arrival* ∧ *si.signalPriority* < *siSeq.head.signalPriority*)
then < *si* > ⌢ *siSeq* // add to front of queue
else < *siSeq.head* > ⌢ *insert*(*si*, *t*, *siSeq.tail*) // insert after head item
endif

Analogously, a function *delete* is used to remove a signal from a finite signal instance list *siSeq*.

delete(*si*:*SIGNALINST*, *siSeq*:*SIGNALINST**): *SIGNALINST** =$_{def}$
if *siSeq* = *empty* **then** *empty*
elseif *siSeq.head* = *si* **then** *siSeq.tail*
else < *siSeq.head* > ⌢ *delete*(*si*, *siSeq.tail*)
endif

The macros INSERT and DELETE update the schedule of a gate *g* by a revised signal list to *g.schedule*. The signal arrival time in INSERT is the time *t* derived from channel delays except when called from DELIVERSIGNALS in clause F3.2.3.2.1, the time parameter is the availability time if this is later than the arrival time at the given gate.

INSERT(*si*:*SIGNALINST*, *t*:*TIME*, *g*:*GATE*) ≡
g.schedule := *insert*(*si*,*t*,*g.schedule*)
si.arrival := *t* // arrival time at gate

DELETE(*si*:*SIGNALINST*, *g*:*GATE*) ≡
g.schedule := *delete*(*si*,*g.schedule*)

The function *nextSignal* yields, for a given signal instance and a sequence of signal instances, the next signal instance in the sequence after the given instance, or the value *undefined*, if the sequence is empty, or there is no instance after the given instance, or the given signal instance is not in the sequence.

nextSignal(*si*: *SIGNALINST*, *siSeq*:*SIGNALINST**): [*SIGNALINST*] =$_{def}$
if *siSeq* = *empty* **then** *undefined*
elseif *siSeq.head* = *si*
then
if *siSeq.tail* = *empty* **then** *undefined*
else *siSeq.tail.head*
endif
else *nextSignal*(*si*, *siSeq.tail*)
endif

The function *selectContinuousSignal* yields, for a set of continuous signal transitions and a set of natural numbers, an element of the transition set with a priority not contained in the set of natural numbers, such that this priority is the maximum priority of all transitions not having priorities in this set of natural numbers.

selectContinuousSignal(*tSet*: *SEMTRANSITION*-**set**, *nSet*: *NAT*-**set**): [*SEMTRANSITION*] =$_{def}$
if $\forall t1 \in tSet$: *t1*.**s**-*NAT* $\in$ *nSet* **then** *undefined*
else *take*({$t \in tSet$: *t*.**s**-*NAT* $\notin$ *nSet* $\wedge$ $\forall t1 \in tSet$: (*t1*.**s**-*NAT* $\notin$ *nSet* $\Rightarrow$ *t*.**s**-*NAT* $\leq$ *t1*.**s**-*NAT*)})
endif

F3.2.1.1.3 Channels

Channels, as declared in a given SDL-2010 specification, consist of either one or two unidirectional *channel paths*. In the SAM model, each channel path is identified with an object of a derived domain *LINK*. The element of a *LINK* is a SAM agent, such that the behaviour is defined through LINK-PROGRAM.

LINK =$_{def}$ { $a \in$ *AGENT* : *a.program* = LINK-PROGRAM}

LINKSEQ =$_{def}$ *LINK** // used for the sequence of links in functions *allConnections* and *commPathIds*

Intuitively, elements of *LINK* are considered as point-to-point connection primitives for the transport of signals. More specifically, each *l* element of *LINK* is able to convey certain signal types, as specified by *l.with*, from an originating gate *l.from* to a destination gate *l.to*, and *l.nodelay* indicating if *l* is non-delaying.

controlled *with*: *LINK* $\rightarrow$ *SIGNAL*-**set**

controlled *from*: *LINK* $\rightarrow$ [*GATE*]
// need to have optional result here, because function is also called within *allConnections* with general *AGENT*

controlled *to*: *LINK* $\rightarrow$ *GATE*

controlled *noDelay*: *LINK* $\rightarrow$ [**NODELAY**]

Signal delays

SDL-2010 considers channels as reliable and order-preserving communication links. A channel is able to delay the transport of a signal for an *indeterminate* and *non-constant* time interval. Although the exact delaying behaviour is not further specified, the fact that channels are reliable implies that all delays are finite.

Signal delays are modelled through a monitored function *delay* stating the dependency on external conditions and events. In a given SAM state, *delay* associates finite time intervals from a domain

DURATION to the elements of *LINK*, where the duration of a particular signal delay appears to be chosen non-deterministically.

DURATION =$_{def}$ *REAL*

monitored *delay*: *LINK* → *DURATION*

Integrity constraints

There are two important integrity constraints on the function *delay*:

1. Taking into account that there are also non-delaying channels, the only admissible value for non-delaying channel paths is *0*.
2. For every link agent *l*, the value of (*now + l.delay)* increases monotonically (with respect to *now*).

The second integrity constraint is needed in order to ensure that channel paths are *order-preserving*: that is, signals transported via the same channel path (and therefore are inserted into the same destination schedule) cannot overtake each other.

Channel behaviour

A link agent *l* performs a single operation: signals received at gate *l.from* are forwarded to gate *l.to*. That means, *l* permanently watches *l.from* waiting for the next deliverable signal in *l.from.queue*. Whenever *l* is applicable to a waiting signal *si* (as identified by the *l.from.queue.head*), it attempts to remove *si* from *l.from.queue* in order to insert it into *l.to.schedule*. This attempt need not necessarily be successful as, in general, there may be several link agents competing for the same signal *si*.

The following determines whether a link agent *l* is applicable to a signal *si*. The determination depends on the values of *si.toArg*, *si.viaArg*, *si.signalType* and *l.with*: *l* is a legal choice for the transportation of *si* only, if both the following two conditions hold:

1. *si.signalType* ∈ *l.with*, and
2. there exists an applicable path connecting *l.to* to some final destination that matches with the address information and the path constraints of *si*.

Abstractly, this decision can be expressed using a predicate function *applicable*, defined below.

F3.2.1.1.4 Reachability

When signals are sent, it has to be determined whether there currently is an applicable communication path: a path consisting of a sequence of links that can transfer the signal, and that satisfies further constraints as specified by the optional to- and via-arguments. The predicate function *applicable* formally states all conditions to be satisfied. The domain *TOARG* is defined in clause F3.2.1.1.1. The function *applicable* is invoked from the macro SIGNALOUTPUT in F3.2.1.4.13 for the primitive *OUTPUT* and in the macro LINK-PROGRAM in F3.2.3.1.7. SIGNALOUTPUT is called via EVALOUTPUT and EVAL from *compile* for an *Output-node* in F3.2.2.3 where the destination is resolved to a *PID* in the function *toPid*. In the case of LINK-PROGRAM the destination is derived from the destination of the signal at the head of the queue *from Self*. This destination is set for the signal by a previous call of SIGNALOUTPUT.

applicable(*s*: *SIGNAL*, *toArg*: [*TOARG*], *viaArg*: *VIAARG*, *g*: *GATE*, *l*: [*LINK*]): *BOOLEAN* =$_{def}$
∃ *commPath* ∈ *allConnections* (*g*): // set of all connections from g – each of which is a *LINKSEQ*
(∀ *lnk* ∈ *commPath*: *s* ∈ *lnk.with* ∧ *lnk.owner* ≠ *undefined*) // each *LINK* carries signal and owned
∧
if *commPath* = *empty* **then** // there are no connection paths
l = *undefined* // the *LINK* shall be undefined
∧ ((*g.direction* = *outDir*) ⇒ (*toArg* = *undefined* // the destination shall be undefined

```
                ∧  s ∈ g.gateAS1.s2-Identifier-set))  // the signal shall be in the Out-signal-identifier-set
            ∧  ((g.direction = inDir) ⇒ (validDestinationGate(g, toArg)  // to self
                ∧  s ∈ g.gateAS1.s1-Identifier-set)  // the signal shall be in In-signal-identifier-set)
            ∧ viaArg = ∅
        else // there are connection paths
                if l ≠ undefined then commPath.head = l else true endif // path starts with LINK if given
            ∧  ¬ ∃lnk ∈ LINK: (lnk.from = commPath.last.to ∧ s ∈ lnk.with) // the path is complete
            ∧  viaArg ⊆ commPath.commPathIds // destination is one on the path
            ∧  validDestinationGate(commPath.last.to, toArg)
        endif

validDestinationGate(g: GATE, toArg: [ TOARG ]): BOOLEAN =def
    case toArg of
    | Identifier then
        if toArg.refersto1 ∈ Agent-definition // toArg identifies an agent
        then g.myAgent.agentAS1.identifier1 = toArg // valid if gate of agent
        else false // identifier not an agent – path not valid
        endif
    | PID then
        if toArg ≠ nullPid // toArg is Pid other than null
        then ∃sa ∈ AGENT: (sa.owner = g.myAgent ∧ sa.selfPid = toArg)
            //there is an agent owned by the agent with the gate and the given Pid.
        else false // the destination is null – therefore path not valid
        endif
    endcase

allConnections(g: GATE): LINKSEQ-set =def
        ⋃ ({ { < l > ⌢ list | list ∈ allConnections(l.to) } | l ∈ LINK: l.from = g })
    ∪  { empty }

commPathIds(lSeq: LINKSEQ): Identifier-set =def
        { g.gateAS1.identifier1 | g ∈ GATE: ∃lnk in lSeq: (g = lnk.from ∨ g = lnk.to) }
    ∪  { lnk.agentAS1.identifier1 | lnk ∈ LINK: (lnk in lSeq) }
```

F3.2.1.1.5 Timers

A particular concise way of modelling timers is by identifying timer objects with respective timer signals. More precisely, each *active* timer is represented by a corresponding timer signal in the schedule associated with the input port of the related agent instance.

```
TIMER =def Identifier
TIMERINST =def PID × TIMER × VALUE*
```

Active timers

To indicate whether a timer instance *tmi* is active or not, there is a corresponding derived predicate function *active*:

```
active(tmi:TIMERINST): BOOLEAN =def tmi ∈ Self.inport.schedule
```

Timer operations

The macros below model the SDL-2010 actions *Set-node* and *Reset-node* on timers as executed by a corresponding SDL-2010 agent. A function (*duration*) gives default duration values as defined by an SDL-2010 specification under consideration.

```
 duration(t: TIMER ): DURATION=def
    t.refersto1.s-Timer-definition.s-EXPRESSION // Expression for Timer-default-initialization

SETTIMER(tm:TIMER, vSeq :VALUE*, t:[TIME]) ≡
let tmi = mk-TIMERINST(Self.selfPid, tm, vSeq ) in
```

```
    if t = undefined then
        Self.inport.schedule := insert(tmi, now + tm.duration, delete(tmi, Self.inport.schedule))
        tmi.arrival := now + tm.duration
    else
        Self.inport.schedule := insert(tmi, t, delete(tmi, Self.inport.schedule))
        tmi.arrival := t
    endif
endlet

RESETTIMER(tm:TIMER, vSeq :VALUE*) ≡
let tmi = mk-TIMERINST(Self.selfPid, tm, vSeq ) in
    if active(tmi) then
        DELETE(tmi, Self.inport)
    endif
endlet
```

F3.2.1.1.6 Exceptions

Exceptions are identified dynamic conditions. How the system behaves when an exception occurs, is not defined by SDL-2010. Each kind of exception has an identity that can be used in the implementation to report or to handle the exception. The *raise* function (see clause F3.3.1.1) is called for the dynamic conditions under which an exception occurs with the exception as a parameter. As the further behaviour is undefined when an exception occurs, it is preferable if the SDL-2010 is written to prevent the dynamic conditions arising (for example, checking on indexing bounds).

EXCEPTION =$_{\text{def}}$ *Exception-identifier*

NOTE – *Exception-identifier* is only defined as an abstract syntax rule here and is not used elsewhere. The predefined exception names are not part of the concrete grammar.

F3.2.1.2 SDL-2010 agents

In this clause, the domain *AGENT* is further refined to consist of three basically different types of agents, namely: link agent instances (modelled by the domain *LINK*, see clause F3.2.1.1.3), SDL-2010 agent instances, and SDL-2010 agent set instances (modelled by the derived domains *SDLAGENT* and *SDLAGENTSET*, respectively).

SDLAGENT =$_{\text{def}}$ *AGENT*

SDLAGENTSET =$_{\text{def}}$ *AGENT*-**set**

Initially, there is only a single agent *system* denoting a distinguished SDL-2010 agent set instance of the domain *SDLAGENTSET*.

```
static system: → SDLAGENTSET
    initially AGENT = { system }

controlled agentSetPids: SDLAGENTSET → PID*
```

The function *agentSetPids* contains the list of pid values corresponding to the SDL-2010 agent instances of the SDL-2010 agent set instances.

F3.2.1.2.1 State machine

The structure of the agent's state machine is directly modelled, and built up during the agent initialization. To represent the structure formally, several domains and functions are used. The state machine structure is exploited in the execution phase, when transitions are selected, and states entered and left.

```
controlled domain STATENODE
    initially STATENODE = ∅
```

The *STATENODE* domain is modified in clauses within clause F3.2.3.1 to contain entries for each basic node or composite state type in the system.

STATENODEKIND =def { *stateNode*, *statePartition*, *procedureNode*}

A *STATENODE* for which *stateNodeKind = procedureNode*, is the top-level node of a procedure graph, which is unfolded node by node subsequently. Such nodes are created dynamically, when a procedure call is made.

STATENODEREFINEMENTKIND =def { *compositeStateGraph*, *stateAggregationNode*}
STATEENTRYPOINT =def [*Name*] // State-entry-point-name or undefined
STATEEXITPOINT =def *Name* ∪ **DEFAULT** /. State-exit-point-name or DEFAULT
STATENODEWITHENTRYPOINT =def *STATENODE* × (*STATEENTRYPOINT* ∪ **HISTORY**)
STATENODEWITHEXITPOINT =def *STATENODE* × *STATEEXITPOINT*
STATENODEWITHCONNECTOR =def *STATENODE* × *Free-action*.**s-***Name* // Free-action Connector-name

The first group of declarations and definitions introduces a controlled domain *STATENODE*, and a number of derived domains.

controlled *stateNodeKind*: *STATENODE* → *STATENODEKIND*
controlled *stateNodeRefinement*: *STATENODE* → [*STATENODEREFINEMENTKIND*]
controlled *stateName*: *STATENODE* → *Name* // State-name
controlled *stateId*: *STATENODE* → *STATEID*
controlled *inheritedStateNode*: *STATENODE* → [*STATENODE*]
controlled *parentStateNode*: *STATENODE* → [*STATENODE*]
controlled *stateTransitions*: *STATENODE* → *SEMTRANSITION*-**set**
controlled *startTransitions*: *STATENODE* → *STARTTRANSITION*-**set**
controlled *freeActions*: *STATENODE* → *FREEACTION*-**set**
controlled *statePartitionSet*: *STATENODE* → *STATENODE*-**set**

The *stateNodeRefinement* of a *STATENODE* for a basic state is *undefined*.

The *parentStateNode* of a *STATENODE* is either *undefined* for a basic state, or the *STATENODE* for the composite state type of a composite state node, or *undefined* or the super type for a composite state type.

The *inheritedStateNode* of a *STATENODE* is either *undefined* for a basic state or an unspecialized composite state, or one of the specializations a composite state type.

The second group of declarations introduces controlled functions defined on the domain *STATENODE*, they can be understood as a state node control block and are used to model the state machine by a hierarchical inheritance state graph.

controlled *currentSubStates*: *STATENODE* → *STATENODE*-**set**
controlled *previousSubStates*: *STATENODE* → *STATENODE*-**set**

The *currentSubStates* function defines, for each state node, the *current* substates. If the state node is refined into a composite state graph, this is at most one substate. In case of a state aggregation node, this is a subset of the state partition set.

The *previousSubStates* function gives the set of state nodes to use when a composite state with **HISTORY** is re-entered.

collectCurrentSubStates(*sn*: *STATENODE*): *STATENODE*-**set** =def
{*sn*} ∪ **U** ({*collectCurrentSubStates*(*x*) | *x* ∈ *sn.currentSubStates* ∪ *sn.inheritedStateNodes*})

The *collectCurrentSubStates* function collects, for a given state node, all current substates.

controlled *currentExitPoints*: *STATENODE* → *STATEEXITPOINT*-**set**

The *currentExitPoints* function defines, for each state aggregation node, the *current* exit points: the exit points activated by exiting state partitions. The state aggregation is exited only if all state partitions have exited.

inheritsFrom(*sn1*: STATENODE, *sn2*: STATENODE): BOOLEAN =def
if *sn2.parentStateNode = undefined* **then** *false*
elseif *sn1.parentStateNode = undefined* **then** *false*
else
sn2.parentStateNode ∈ *sn1.parentStateNode.inheritedStateNodes* ∧
sn1.stateName ≠ *sn2.stateName*
endif

The *inheritsFrom* predicate determines whether the composite state type of one state node (*sn2*) inherits the composite state type of another state node (*sn1*).

directlyInheritsFrom(*sn1*: STATENODE, *sn2*: STATENODE): BOOLEAN =def
inheritsFrom(*sn1*, *sn2*) ∧
(¬ ∃*snx* ∈ STATENODE:
inheritsFrom(*sn1*, *snx*)∧ *inheritsFrom*(*snx*, *sn2*))

The *directlyInheritsFrom* predicate determines whether the composite state type of one state node (*sn2*) directly inherits (in one step) the composite state type of another state node (*sn1*).

directlyRefinedBy(*sn1*: STATENODE, *sn2*: STATENODE): BOOLEAN =def
sn2.parentStateNode = sn1

The *directlyRefinedBy* predicate determines whether a state node is refined by another state node by a single refinement step.

directlyInheritsFromOrRefinedBy(*sn1*: STATENODE, *sn2*: STATENODE): BOOLEAN =def
directlyRefinedBy(*sn1*, *sn2*) ∨ *directlyInheritsFrom*(*sn1*, *sn2*)

The *directlyInheritsFromOrRefinedBy* predicate determines whether two state nodes are related by a sequence of refinement or inheritance steps.

inheritsFromOrRefinedBy(*sn1*: STATENODE, *sn2*: STATENODE): BOOLEAN =def
directlyInheritsFromOrRefinedBy(*sn1*, *sn2*) ∨
(∃ *sn3* ∈ { *sn* ∈ STATENODE: *directlyInheritsFromOrRefinedBy* (*sn1*, *sn*) }:
(*inheritsFromOrRefinedBy*(*sn3*, *sn2*)))

The *inheritsFromOrRefinedBy* predicate determines whether *sn1* inherits from or is refined by *sn2*, taking transitivity of this relationship into account.

selectNextStateNode(*snSet*: STATENODE-**set**): [STATENODE] =def
let *sn* = *take*({*sn1* ∈ *snSet*: (¬ ∃*sn2* ∈ *snSet*: *inheritsFromOrRefinedBy*(*sn1*, *sn2*))}) **in**
if *sn* = *undefined* **then** *undefined*
elseif ∃*sn1* ∈ *snSet*: *directlyInheritsFrom*(*sn1*, *sn*) ∨ *sn* = *sn1.inheritedStateNode* **then**
selectNextStateNode(*snSet* \ {*sn*})
else *sn*
endif
endlet

The *selectNextStateNode* function returns a state node that may be checked next, provided *snSet* is a valid set of current state nodes reduced by state nodes that have already been selected with this function.

inheritedStateNodes(*sn*: STATENODE): STATENODE-**set** =def
if *sn.inheritedStateNode = undefined* **then** ∅
else {*sn.inheritedStateNode*} ∪ *sn.inheritedStateNode.inheritedStateNodes*
endif

The *inheritedStateNodes* function defines, for a given state node, the set of inherited state nodes.

parentStateNodes(*sn*: *STATENODE*): *STATENODE*-**set** =$_{\text{def}}$
if *sn.parentStateNode = undefined* **then** ∅
else {*sn.parentStateNode*} ∪ *sn.parentStateNode.parentStateNodes*
endif

The *parentStateNodes* function defines, for a given state node, the set of parent state nodes.

mostSpecialisedStateNode(*sn*:*STATENODE*): *STATENODE* =$_{\text{def}}$
let *sn1 = take*({*sn2* ∈ *STATENODE*: *inheritsFrom*(*sn2*, *sn*)}) **in**
if *sn1 = undefined* **then** *sn* **else** *sn1.mostSpecialisedStateNode* **endif**
endlet

The *mostSpecialisedStateNode* function returns, for a given state node, the most specialized state node applied during the selection of transitions in order to obtain the correct sequence of state node checks.

selectInheritedStateNode(*sn*: *STATENODE*, *snSet*: *STATENODE*-**set**): [*STATENODE*]=$_{\text{def}}$
take({*sn1* ∈ *snSet*: *directlyInheritsFrom*(*sn*,*sn1*)})

The *selectInheritedStateNode* function yields a state node that may be left next, provided *snSet* is a valid set of state nodes to be left.

getPreviousStatePartition(*sn*: *STATENODE*): *STATENODE* =$_{\text{def}}$
if *sn.stateNodeKind = statePartition* ∧
¬ ∃*sn1* ∈ *sn.parentStateNodes*: *sn1.stateNodeKind = procedureNode*
then *sn.mostSpecialisedStateNode*
else *getPreviousStatePartition*(*sn.parentStateNode*)
endif

The *getPreviousStatePartition* function determines, for a given state node, the innermost state partition not belonging to a procedure.

controlled *resultLabel*: *STATENODE* → *LABEL*

The *resultLabel* function refers to the location of the return value, if the state node is a procedure state node: that is, a state node owning the procedure graph.

controlled *resultRefVarId*: *STATENODE* → { *id* : *id* ∈ *Identifier* ∧ *id.refersto1* ∈ *VARDEF* }

The *resultRefVarId* function is set if the state node is a procedure state node (that is, a state node owning the procedure graph), the procedure has a *Result-aggregation* **REF** and the *Value-returning-call-node* that created the procedure graph is assigned to variable or parameter that has an *Aggregation-kind* **REF**. The *resultRefVarId* function is set to the identifier of the variable or parameter.

controlled *callingProcedureNode*: (*AGENT* ∪ *STATENODE*) → [*STATENODE*]

The *callingProcedureNode* function refers to the root node of the calling procedure, if any, and is associated with the state node owning the procedure graph. Thus, nested procedure calls are modelled.

controlled *entryConnection*: *STATEENTRYPOINT* × *STATENODE* → [*STATEENTRYPOINT*]

controlled *exitConnection*: *STATEEXITPOINT* × *STATENODE* → *STATEEXITPOINT*

Finally, the *entryConnection* and *exitConnection* functions model the entry and exit connections of state nodes.

F3.2.1.2.2 Agent modes

To model the dynamic semantics of agents, several activity phases are distinguished. These phases are modelled by a hierarchy of *agent modes*. At this point, the agent modes are formally introduced; their usage is explained in clause F3.2.3.

```
AGENTMODE =def {
    initialisation,                 // agent mode 1
    execution,                      // agent mode 1

    selectingTransition,            // agent mode 2
    firingTransition,               // agent mode 2
    stopping,                       // agent mode 2

    initialising1,                  // agent mode 2, 4
    initialising2,                  // agent mode 2
    initialisingStateMachine,       // agent mode 2
    initialisingProcedureGraph,     // agent mode 4
    initialisationFinished,         // agent mode 2, 4

    startSelection,                 // agent mode 3
    selectFreeAction,               // agent mode 3
    selectExitTransition,           // agent mode 3
    selectStartTransition,          // agent mode 3
    selectPriorityInput,            // agent mode 3
    selectInput,                    // agent mode 3
    selectContinuous,               // agent mode 3

    startPhase,                     // agent mode 2, 4
    selectionPhase,                 // agent mode 4, 5
    evaluationPhase,                // agent mode 4, 5
    selectSpontaneous,              // agent mode 4

    leavingStateNode,               // agent mode 3
    firingAction,                   // agent mode 3, 4
    enteringStateNode,              // agent mode 3
    exitingCompositeState,          // agent mode 3
    initialisingProcedure,          // agent mode 3

    enterPhase,                     // agent mode 4
    enteringFinished,               // agent mode 4
    leavePhase,                     // agent mode 4
    leavingFinished}                // agent mode 4
```

The agent modes are grouped according to their usage and the level of the agent mode hierarchy where they are relevant. In cases no conflict arises, agent modes may be applied on more than one level of this hierarchy.

F3.2.1.2.3 Agent control block

The state information of an SDL-2010 agent instance is collected in an *agent control block*. The agent control block is partially initialized when an SDL-2010 agent (set) instance is created, and completed/modified during its initialization and execution. Since part of the state information is valid only during certain activity phases, the agent control block is structured accordingly. Following is the state information needed in all phases. Further control blocks that form part of the agent control block, but are relevant during certain activity phases only, are defined subsequently.

controlled *owner*: AGENT ∪ STATENODE ∪ LINK → [AGENT]

Hierarchical system structure is modelled by means of a function *owner* defined on agents, on state nodes (see clause F3.2.1.2.1) and on agents that are links representing channels, expressing *structural relations* between them and their constituent components. More specifically, an agent set

instance is considered as *owner* of all those agent instances currently contained in the set; an agent instance is the *owner* of its substructure, consisting of agent set instances. Similarly, a composite state node is the *owner* of the state nodes or state partitions forming the refinement. Since the system agent is the root of the agent hierarchy, it has no owner (*system.owner = undefined*).

controlled *agentAS1*: *AGENT* → *Agent-definition*

controlled *channelAS1*: *LINK* → *Channel-definition*

controlled *gateAS1*: *GATE* → [*Gate-definition*]

controlled *stateAS1*: *STATENODE* → *State-node*

controlled *procedureAS1*: *STATENODE* → [*Procedure-definition*]

controlled *stateDefinitionAS1*: *STATENODE* → *Composite-state-type-definition*

controlled *partitionAS1*: *STATENODE* → [*State-partition*]

A series of unary functions (*agentAS1* to *partitionAS1*, see above, defined on agents, gates and state nodes) identify the corresponding AST definition. These definitions are needed during the initialization phase and also during dynamic creation of agents.

isAgentSet(*ag*: *AGENT*): *BOOLEAN* =$_{\text{def}}$ *ag.program* = AGENT-SET-PROGRAM

To distinguish SDL-2010 agent sets from other agents, the predicate *isAgentSet* is defined.

controlled *parent*: *SDLAGENT* → [*PID*]

controlled *offspring*: *SDLAGENT* → *PID*

controlled *selfPid*: *SDLAGENT* → *PID*

controlled *sender*: *SDLAGENT* → *PID*

The above functions model the corresponding Pid expressions introduced in [ITU-T Z.101].

signal: (*ag* : *SDLAGENT*) : *Identifier* =$_{\text{def}}$
  take({ *id* ∈ *Identifier* |
      id.refersto1 ∈ *ag.agentAS1*.**s-***Identifier.refersto1*.**s-***Variable-definition*-**set**
    ∧ *id*.**s-***Name*.**s-***TOKEN* = "#signal"
  })

The above function models the implicit anonymous choice variable capable of holding any of the values of signals in the valid input signal set of the agent as described in clause 12.3.4.8 of [ITU-T Z.104].

controlled *state*: *SDLAGENT* → *STATE*

The values of the variables of an agent are collected in a state associated with some agent, modelled by the function *state*. This function is changed dynamically whenever the variable values of an agent or a procedure change. The data semantics provides the initial value for this function via *initAgentState* and *initProcedureState*.

controlled *stateAgent*: *SDLAGENT* → *SDLAGENT*

The values of the variables of an SDL-2010 agent are normally associated with the agent. However, in case of nested process agents (that is, process agents contained within a process agent), they are associated with the outermost process agent. The function *stateAgent* yields, for a given SDL-2010 agent, the SDL-2010 agent to which the variable values are associated.

controlled *topStateId*: *SDLAGENT* → *STATEID*

The *topStateId* function associates the outermost scope with an agent. In case of nested process agents, it is only defined for the outermost process agent.

controlled *isActive*: *SDLAGENT* → [*SDLAGENT*]

Nested process agents are to be executed in an interleaving manner. To model the required synchronization, the function *isActive* of the outermost process agent is used.

monitored *spontaneous*: *AGENT* → *BOOLEAN*

The SDL-2010 concept of *spontaneous transition* is abstractly modelled by means of a monitored predicate *spontaneous* associated with a particular SDL-2010 agent instance, which serves for triggering spontaneous transition events. It is assumed that spontaneous transitions occur from time to time without being aware of any causal dependence on external conditions and events. This view reflects the indeterminate nature behind the concept of spontaneous transition.

controlled *inport*: *SDLAGENT* → *GATE*

Each SDL-2010 agent instance has its local *input port* at which arriving signals are stored until these signals either are actively received, or until they are discarded. Input ports are modelled as a gate, containing a finite sequence of signals.

controlled *currentSignalInst*: *SDLAGENT* → [*SIGNALINST*]

During the firing of input transitions, the signal instance removed from the input port is available through the function *currentSignalInst*.

controlled *topStateNode*: *SDLAGENT* → *STATENODE*

The state nodes of an agent are rooted at a top state node modelling the state machine of the agent instance.

controlled *currentStartNodes*: *SDLAGENT* → *STATENODEWITHENTRYPOINT*-**set**

Start transitions take precedence over regular transitions; they are identified by tuples consisting of a state node and an entry point.

controlled *currentExitStateNodes*: *SDLAGENT* → *STATENODEWITHEXITPOINT*-**set**

Exit transitions take precedence over regular transitions; they are identified by tuples consisting of a state node and an exit point.

controlled *currentConnector*: *SDLAGENT* → [*STATENODEWITHCONNECTOR*]

Free actions take precedence over regular transitions; they are identified by tuples consisting of a state node and a connector name.

controlled *scopeName*: *SDLAGENT* × *STATEID* → *Compound-node*.**s**-*Name* // Compound st./Connector name

controlled *scopeContinueLabel*: *SDLAGENT* × *STATEID* → *CONTINUELABEL* // next for compound statement

controlled *scopeStepLabel*: *SDLAGENT* × *STATEID* → *STEPLABEL* // label for next loop iteration

The functions *scopeName*, *scopeContinueLabel* and *scopeStepLabel* are used for *Compound-node* interpretation (see [ITU-T Z.102]).

INITSTATEMACHINE/INITPROCEDUREGRAPH control block

When the state machine of an agent is initialized, a hierarchical inheritance state graph is created. Because this normally takes several steps, the intermediate status of the creation is kept in an INITSTATEMACHINE/INITPROCEDUREGRAPH control block. Based on this information, it is, for instance, possible to control the order of node creation as far as necessary. This control block is used during the initialization of the agent instance, and also dynamically when a procedure call occurs.

controlled *stateNodesToBeCreated*: *SDLAGENT* → *State-node*-**set**

controlled *statePartitionsToBeCreated*: *SDLAGENT* → *State-partition*-**set**

controlled *stateNodesToBeRefined*: *SDLAGENT* → *STATENODE*-**set**

controlled *stateNodesToBeSpecialised*: *SDLAGENT* → *STATENODE*-**set**

In order to keep track of the state machine creation, a distinction is made between the state nodes and the state partitions to be created. Also, the refinement and specialization of state nodes is taken into account.

Selection control block

During the selection of a transition, additional information is needed to keep track of the selection status. For instance, when the selection starts, the input port is "frozen", meaning that its state at the beginning of the selection is the basis for this selection cycle. This does not prevent signal instances arriving while the selection is active, but these signals are not considered before the next selection cycle.

controlled *inputPortChecked*: *SDLAGENT* → *SIGNALINST**

controlled *stateNodesToBeChecked*: *SDLAGENT* → *STATENODE*-**set**

controlled *stateNodeChecked*: *SDLAGENT* → [*STATENODE*]

controlled *startNodeChecked*: *SDLAGENT* → *STATENODEWITHENTRYPOINT*

controlled *exitNodeChecked*: *SDLAGENT* → *STATENODEWITHEXITPOINT*

controlled *transitionsToBeChecked*: *SDLAGENT* → *SEMTRANSITION*-**set**

controlled *transitionChecked*: *SDLAGENT* → *SEMTRANSITION*

controlled *signalChecked*: *SDLAGENT* → *SIGNALINST*

controlled *SignalSaved*: *SDLAGENT* → *BOOLEAN*

controlled *continuousPriorities*: *SDLAGENT* → *NAT*-**set**

Enter/Leave/ExitStateNode control block

In general, to enter, leave or exit a state node requires a sequence of steps. In hierarchical state graphs, entering a state node means to enter contained states, and to execute start transitions and entry procedures. Likewise, leaving a state node means to leave the contained states and to execute exit procedures. Exiting a composite state in addition means to fire an exit transition. During these activity phases, the status information is maintained in the enter/leave/exitStateNode control block.

controlled *stateNodesToBeEntered*: *SDLAGENT* → *STATENODEWITHENTRYPOINT*-**set**

controlled *stateNodesToBeLeft*: *SDLAGENT* → *STATENODE*-**set**

controlled *stateNodeToBeExited*: *SDLAGENT* → [*STATENODEWITHEXITPOINT*]

Procedure control block

The procedure control block comprises the part of the agent control block that has to be stacked when a procedure call occurs. This includes the agent modes, the current action label, and the state identification. Once the procedure terminates, this state information has to be restored. The stacked information is associated with the state node containing the procedure graph. Such a state node is created dynamically for each procedure call.

During the execution of a procedure, other control blocks may be required, for instance, the INITSTATEMACHINE control block or the selection control block. However, the corresponding phases do not lead to the execution of further procedures, and are not interrupted by other phases. Therefore, it is not necessary to stack these parts of the agent control block.

controlled *agentMode1*: *AGENT* ∪ *STATENODE* → *AGENTMODE*

controlled *agentMode2*: *AGENT* ∪ *STATENODE* → *AGENTMODE*

controlled *agentMode3*: *AGENT* ∪ *STATENODE* → *AGENTMODE*

controlled *agentMode4*: *AGENT* ∪ *STATENODE* → *AGENTMODE*

controlled *agentMode5*: *AGENT* ∪ *STATENODE* → *AGENTMODE*

To control the execution of agents, a control hierarchy is formed, which consists of up to five levels, depending on the current execution phase. For each of these levels, a specific function *agentMode* is defined.

controlled *currentStateId*: *SDLAGENT* ∪ *STATENODE* → *STATEID*

In order to handle nested process agents and procedure calls, a state may contain substates. Every substate is given an identification at the time of its creation; for example, when a procedure is called or when a nested process agent is started. These identifications are taken from the domain *STATEID*. A *STATE* contains associations between a number of *STATEID* values, a number of variable identifiers, and their respective values.

controlled *currentLabel*: *SDLAGENT* ∪ *STATENODE* → [*LABEL*]

The *currentLabel* function, which identifies the action currently executed or to be executed next, controls the firing of transitions and the evaluation of expressions. When a sequence of steps is completed, *currentLabel* is set to *undefined*.

controlled *continueLabel*: *SDLAGENT* ∪ *STATENODE* → [*CONTINUELABEL*]

The *continueLabel* function is needed while a state node is left, which forms part of the firing of a transition and may lead to the execution of further action sequences. When the state node is left, firing of the transition is resumed. In particular, this value is needed when procedures are executed. Also, this function records the label where execution is continued after a procedure call.

controlled *currentParentStateNode*: *SDLAGENT* ∪ *STATENODE* → *STATENODE*

The *currentParentStateNode* function defines the correct ownership between state nodes, and identifies states to be left and to be entered.

controlled *previousStateNode*: *SDLAGENT* ∪ *STATENODE* → *STATENODE*

When a transition is fired, the *previousStateNode* function refers to the state node where the transition started.

controlled *currentProcedureStateNode*: *SDLAGENT* ∪ *STATENODE* → *STATENODE*

The *currentProcedureStateNode* function refers to the current procedure state node.

F3.2.1.2.4 Agent connections

SDL-2010 agents are organized in agent sets. All members of an agent set have the same sets of input gates and output gates as defined for the agent set.

gateUnconnected(*g*:*GATE*):*BOOLEAN* =def
let *myDef*: *Agent-type-definition* = *g.myAgent.agentAS1*.**s**-*Identifier.refersto1* **in**
∀*cd* ∈ *myDef*.**s**-*Channel-definition*-**set**: ∀*cp* ∈ *cd*.**s**-*Channel-path*-**set**:
(*g.gateAS1* ≠ *cp*.**s1**-*Identifier.refersto1* ∧ // Originating-gate of path = Gate-identifier = Identifier
g.gateAS1 ≠ *cp*.**s2**-*Identifier.refersto1*) // Destination-gate of path = Gate-identifier = Identifier
endlet

The *gateUnconnected* is true if the gate is not linked to an inner gate by a channel path:

ingates(*a*: *AGENT*): *GATE*-**set** =def
if *a.isAgentSet* **then** // is a an agent for an SDL-2010 agent
{ *g* ∈ *GATE*: *g.myAgent* = *a* ∧ *g.direction* = *inDir* ∧ *g.gateUnconnected*}
else // find ingates of owner of a
if *a.owner* ≠ *undefined* **then** *a.owner.ingates* **else** ∅ **endif**
endif

outgates(*a*:*AGENT*): *GATE*-**set** =def
if *a.isAgentSet* **then** // is a an agent for an SDL-2010 agent
{ *g* ∈ *GATE*: *g.myAgent* = *a* ∧ *g.direction* = *outDir* ∧ *g.gateUnconnected*}
else // find ingates of owner of a
if *a.owner* ≠ *undefined* **then** *a.owner.outgates* **else** ∅ **endif**
endif

The derived function *ingates* (*outgates*) collects all input gates (all output gates) of an agent. Input gates (output gates) are gates of an agent set or agent with direction *inDir* (*outDir*) that are not connected to inner gates by a channel path.

F3.2.1.2.5 Agent behaviour

For the transitions of agents, a tuple domain *SEMTRANSITION* is introduced, consisting of the signal type, an optional gate if the transition is selected only if the signal arrives via that gate, a start label for any firing condition (the optional provided expression of an ordinary input or spontaneous transition, or the enabling condition of a continuous signal), a optional input priority value, the start label of the transition actions and a optional state exit point. Depending on the kind of transition, some components of *SEMTRANSITION* are optional. For instance, in case of a non-priority input transition without an enabling condition, there is no priority and no firing transition.

SEMTRANSITION=def {(*s*,*g*,*lab1*,*n*,*lab2*,*exit*) ∈ *SIGNAL* × [*GATE*] × [*LABEL*] × [*NAT*] × *LABEL* × [*STATEEXITPOINT*]:
if *s* ≠ **NONE** ∧ *n* ≠ *undefined* **then** *g* = *undefined* ∧ *lab1* = *undefined* ∧ *exit* = *undefined* // priority input
elseif *s* ≠ **NONE** ∧ *n* = *undefined* **then** *exit* = *undefined* // ordinary input
elseif *s* = **NONE** ∧ *lab1* ≠ *undefined* ∧ *n* ≠ *undefined* **then** *g* = *undefined* ∧ *exit* = *undefined* // continuous
elseif *s* = **NONE** ∧ *n* = *undefined* **then** *g* = *undefined* ∧ *exit* = *undefined* // spontaneous
else *exit* ≠ *undefined* **then** *s* = **NONE** ∧ *g* = *undefined* ∧ *lab1* = *undefined* ∧ *n* = *undefined* // connect node
endif }

STARTTRANSITION =def *LABEL* × *STATEENTRYPOINT*

FREEACTION =def *Free-action*.**s**-*Name* × *LABEL* // the name is a connector name of a free action

Given a set of transitions, several derived functions are defined to select particular subsets:

priorityInputTransitions(*tSet*:*SEMTRANSITION*-**set**): *SEMTRANSITION*-**set** =$_{def}$
{ *t* ∈ *tSet*: *t*.**s**-*SIGNAL* ≠ **NONE** ∧ *t*.**s**-*NAT* ≠ *undefined* }

inputTransitions(*tSet*:*SEMTRANSITION*-**set**): *SEMTRANSITION*-**set** =$_{def}$
{ *t* ∈ *tSet*: *t*.**s**-*SIGNAL* ≠ **NONE** ∧ *t*.**s**-*NAT* = *undefined* }

continuousSignalTransitions(*tSet*:*SEMTRANSITION*-**set**): *SEMTRANSITION*-**set** =$_{def}$
{ *t* ∈ *tSet*: *t*.**s**-*SIGNAL* = **NONE** ∧ *t*.**s1**-*LABEL* ≠ *undefined* ∧ *t*.**s**-*NAT* ≠ *undefined* }

spontaneousTransitions(*tSet*:*SEMTRANSITION*-**set**): *SEMTRANSITION*-**set** =$_{def}$
{ *t* ∈ *tSet*: *t*.**s**-*SIGNAL* = **NONE** ∧ *t*.**s**-*NAT* = *undefined* }

exitTransitions(*tSet*:*SEMTRANSITION*-**set**): *SEMTRANSITION*-**set** =$_{def}$
{ *t* ∈ *tSet*: *t*.**s**-*STATEEXITPOINT* ≠ *undefined* }

F3.2.1.3 Interface to the data type part

The semantics of the data type part of SDL-2010 is handled separately from the concurrency related aspects of the language. To make this splitting possible, an interface for the semantics definition is defined.

F3.2.1.3.1 Functions provided by the data type part

The data interface is grouped around a derived domain *STATE*. This domain is abstract from the concurrency side, and concrete from the data type side. It represents the values of the variables of an agent, which are collected in the outermost process agent. This is achieved by a dynamic, controlled function *state* defined on process instances (see clause F3.2.1.2.3).

derived domain *STATE*

The function *state* is changed dynamically whenever the state of an agent or a procedure changes. It is solely used within the concurrency semantics part. The data type semantics part provides the initial value for the *state* function via the functions *initAgentState* and *initProcedureState*. In order to handle recursion, a state might contain substates. Every substate is given an identification at the time of its creation; for example, when a procedure is called or when a nested process agent is started. These identifications are in the domain *STATEID*. A *STATE* contains associations between a number of *STATEID* values, a number of variable identifiers, and their respective values.

The parameters of *initAgentState* are:

- State of the outermost process agent (undefined if the outermost process agent is being created)
- State ID of the new state
- State ID of the super state of the new state (undefined for the outermost agent)
- Declarations of the agent

The additional parameter for *initProcedureState* is

- List of formal parameters and list of actual parameters

controlled domain *STATEID*

initAgentState: [*STATE*] × *STATEID* × [*STATEID*] × *DECLARATION*-**set**→ *STATE*

initProcedureState:*STATE* × *STATEID* × *STATEID* × *DECLARATION*-**set** × *CALLPARAM** × *VALUE**
→ *STATEOREXCEPTION* // The last two parameters are the procedure formal parameters and actual parameters.

The domain *DECLARATION* is used to create variables for a state.

DECLARATION =$_{\text{def}}$ *PROCPARAM* ∪ *Variable-definition*

The domain *PROCPARAM* represents the union *In-parameter* ∪ *Inout-parameter* ∪ *Out-parameter*, and the selector **s-***PROCPARAM* can replace **s-**(*In-parameter* ∪ *Inout-parameter* ∪ *Out-parameter*).

PROCPARAM =$_{\text{def}}$ *In-parameter* ∪ *Inout-parameter* ∪ *Out-parameter*

There is also a domain for values, called *VALUE*.

VALUE =$_{\text{def}}$ *PID* ∪ *SDLARRAY* ∪ *SDLBAG* ∪ *SDLBIT* ∪ *SDLBITSTRING* ∪ *SDLBOOLEAN*
∪ *SDLCHARACTER* ∪ *SDLCHARSTRING* ∪ *SDLCHOICE* ∪ *SDLDURATION* ∪ *SDLINTEGER*
∪ *SDLLITERAL* ∪ *SDLOCTETSTRING* ∪ *SDLPOWERSET* ∪ *SDLREAL* ∪ *SDLSTRING*
∪ *SDLSTRUCTURE* ∪ *SDLTIME* ∪ *SDLVECTOR*

Some operations invoked in the data part may raise an exception. In SDL-2010 there is no definition of the handling of exceptions, so that if one occurs the further behaviour of the system is not defined. Therefore, the termination is not defined if an exception occurs in the operation, so the formal semantics is only given for the case of termination without an exception. The possibility of the operation raising an exception is shown by the return being in one of the following domains:

STATEOREXCEPTION =$_{\text{def}}$ *STATE* ∪ *EXCEPTION*
VALUEOREXCEPTION =$_{\text{def}}$ *VALUE* ∪ *EXCEPTION*

The data type part provides the function *assign* that models how assignments are performed.

assign: *Identifier* × *VALUE* × *STATE* × *STATEID* → *STATEOREXCEPTION*

The function *assign* associates a new value with a given variable. There is an ASSIGN rule macro using the *assign* function, which is doing the real assignment.

ASSIGN(*variableId*: *Identifier*, *value*: *VALUE*, *state*: *STATE*, *id*: *STATEID*) ≡
Self.stateAgent.state:= *assign*(*variableId*, *value*, *state*, *id*)

Assignments are the only way to change the state.

The data part provides the function *eval* that determines the value associated with a variable for a given state and state id in the case of aggregation kind **PART**. The function *eval* returns *undefined* if the variable is not set.

eval: *Identifier* × *STATE* × *STATEID* → *VALUE*

The semantics of these functions is given by the data semantics part clause F3.3.5, State access.

When there is an assignment to an item (variable, or procedure formal parameter, or procedure result) with aggregation kind **REF**, the assigned expression has to be a variable access or an operation application that accesses a field of a structure or choice. In these cases, the assignment of a variable is handled by the EVALVAR, and of an operation application is handled by EVALOPERATIONAPPLICATION.

In order to handle expressions, the concurrent semantics provides a domain for procedure bodies, which is also used for method and operator bodies. The data part, in return, provides a domain *OPLITSIGNATURE* for operation (and literal) signatures and a function *dispatch* for calling the procedure definition for an operation.

OPLITSIGNATURE =$_{\text{def}}$ *Static-operation-signature* ∪ *Dynamic-operation-signature* ∪ *Literal-signature*

For modelling the dynamic dispatch, a dispatch function is provided by the data part clause F3.3.7, Operators and methods.

dispatch: *OPLITSIGNATURE* × *VALUE** → *Identifier*

Finally, there are two functions to model the predefined functions that do not have a procedure body because they are part of the predefined data. There is one function to check if the procedure is *functional* (predefined), and one function to *compute* the result in this case. The semantics of these functions is in clause F3.3.1, Predefined data.

functional: *OPLITSIGNATURE* × *VALUE** → *BOOLEAN*
compute: *OPLITSIGNATURE* × *VALUE** → *VALUEORXCEPTION*

F3.2.1.3.2 Functions used by the data type part

The following special points are worth noting:

- If two processes have part of their state in common (which could be possible due to the reference nature of the data type part), there are no semantic problems in the concurrency part, as all state changes are automatically synchronized by the underlying ASM semantics.
- The values for the predefined variables of an agent such as **SENDER**, **PARENT**, **OFFSPRING**, **SELF**, as well as the value of **NOW** are provided by the concurrency part.

F3.2.1.4 Behaviour primitives

This clause describes the SAM behaviour primitives and how these primitives are evaluated. It describes how actions are evaluated, and gives for each primitive a short *explanation* of its intended meaning. Together with the domains, functions and macros that are used to define the behaviour of a primitive, an informal description of the intended meaning is provided as well. Additional *reference clauses* for further explanations complement the description of behaviour primitives.

behaviour: *BEHAVIOUR* =$_{def}$ *rootNodeAS1.compile*

The result of the compilation is accessible through the function *behaviour*. This function is static to reflect the fact that SAM code cannot be modified during execution.

STARTLABEL =$_{def}$ *LABEL*
BEHAVIOUR =$_{def}$ *PRIMITIVE*-**set**
PRIMITIVE =$_{def}$ *LABEL* × *ACTION*

The behaviour consists of a start label and label-action pairs. The label is used to uniquely identify the action and to represent the current state of the interpretation.

F3.2.1.4.1 Action evaluation

Explanation

Action evaluation is used within the execution phase of agents. Primitives are attached to labels. The function *currentLabel* determines for each agent an action to be evaluated next. Actions have different types. For example, there exists, beside others, a primitive for the evaluation of variables and one for procedure calls. The evaluation of an action first determines the type of an action and then, depending of this type, fires an appropriate rule.

Representation

The domain *ACTION* is defined as disjoint union of derived domains, which are explained in the subsequent clauses. For example, there exists a domain *VAR* that contains actions for the evaluation of variables.

ACTION =def *ANYVALUE* ∪ *AGENTINSTPID* ∪ *ASSIGNPARAMETER* ∪ *ASSIGNSIGNAL* ∪ *ASSIGNSIGNALPAR* ∪ *ASSIGNSIGNALVAL* ∪ *BREAK* ∪ *CALL* ∪ *CONTINUE* ∪ *CREATE* ∪ *DECISION* ∪ *DECISIONANYVALUE* ∪ *ENTERSTATENODE* ∪ *EQUALITY* ∪ *LEAVESTATENODE* ∪ *OPERATIONAPPLICATION* ∪ *OUTPUT* ∪ *OUTPUTX* ∪ *OUTPUTZ* ∪ *REFFIELDASSIGN* ∪ *RESET* ∪ *RETURN* ∪ *SCOPE* ∪ *SET* ∪ *SETRANGECHECKVALUE* ∪ *SKIP* ∪ *STOP* ∪ *SYSTEMVALUE* ∪ *TASK* ∪ *TIMERACTIVE* ∪ *TIMERREMAINING* ∪ *TYPECHECK* ∪ *TYPECOERCION* ∪ *VAR* ∪ *ZDECODING* ∪ *ZENCODING* ∪ *ZENCODINGPAR* ∪ *ZENCODINGVAL*

Domains

During the execution phase and the evaluation of actions, labels are used in two ways: as jumps (continue labels) for modelling the corresponding control flow and as stores (value labels) for intermediate results. For example, intermediate results arise during the evaluation of expressions. A domain *CONTINUELABEL* represents labels where an agent continues execution after completing an action. A domain *VALUELABEL* represents labels at which an agent can write or read values.

CONTINUELABEL =def *LABEL*

VALUELABEL =def *LABEL*

Functions

Values stored at value labels can be accessed by a dynamic controlled function *value* and the function *values*.

controlled *value*: *VALUELABEL* × *SDLAGENT* → *VALUE*

values(*lSeq*: *VALUELABEL**, *sa*: *SDLAGENT*): *VALUE** =def
if *lSeq* = *empty* **then** *empty*
else < *value*(*lSeq.head*,*sa*) > ^ *values*(*lSeq.tail*,*sa*)
endif

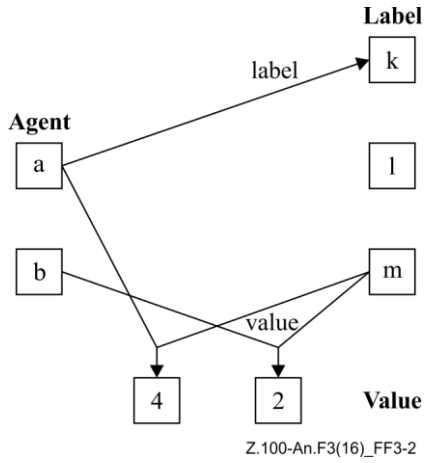

Figure F3-2 – Agents, labels and values

In Figure 3-2 there are two agents, *a* and *b*. The label of agent *a*, which determines the next action to be evaluated within the execution phase, is *k*. Agent *a* has stored value 4 at label *m*, whereas Agent *b* has a stored value 2 at the same label. In this way, different agents can write different values to the same label.

Behaviour

The evaluation of an action is defined by macro EVAL. Macro EVAL takes as argument an action and depending on the type of this action a specific macro is called. These macros are explained in the subsequent clauses. The subdomains of *ACTION* are pairwise disjoint.

EVAL(*a*:*ACTION*) ≡
if *a* ∈ *ANYVALUE* **then** EVALANYVALUE(*a*)
elseif *a* ∈ *AGENTINSTPID* **then** EVALAGENTINSTPID(*a*)
elseif *a* ∈ *ASSIGNPARAMETER* **then** EVALASSIGNPARAMETER(*a*)
elseif *a* ∈ *ASSIGNSIGNAL* **then** EVALASSIGNSIGNAL(*a*)
elseif *a* ∈ *ASSIGNSIGNALPAR* **then** EVALASSIGNSIGNALPAR(*a*)
elseif *a* ∈ *ASSIGNSIGNALVAL* **then** EVALASSIGNSIGNALVAL(*a*)
elseif *a* ∈ *BREAK* **then** EVALBREAK(*a*)
elseif *a* ∈ *CALL* **then** EVALCALL(*a*)
elseif *a* ∈ *CONTINUE* **then** EVALCONTINUE(*a*)
elseif *a* ∈ *CREATE* **then** EVALCREATE(*a*)
elseif *a* ∈ *DECISION* **then** EVALDECISION(*a*)
elseif *a* ∈ *DECISIONANYVALUE* **then** EVALDECISIONANYVALUE(*a*)
elseif *a* ∈ *ENTERSTATENODE* **then** EVALENTERSTATENODE(*a*)
elseif *a* ∈ *EQUALITY* **then** EVALEQUALITY(*a*)
elseif *a* ∈ *LEAVESTATENODE* **then** EVALLEAVESTATENODE(*a*)
elseif *a* ∈ *OPERATIONAPPLICATION* **then** EVALOPERATIONAPPLICATION(*a*)
elseif *a* ∈ *OUTPUT* **then** EVALOUTPUT(*a*)
elseif *a* ∈ *OUTPUTX* **then** EVALOUTPUTX(*a*)
elseif *a* ∈ *OUTPUTZ* **then** EVALOUTPUTZ(*a*)
elseif *a* ∈ *REFFIELDASSIGN* **then** EVALREFFIELDASSIGN(*a*)
elseif *a* ∈ *RESET* **then** EVALRESET(*a*)
elseif *a* ∈ *RETURN* **then** EVALRETURN(*a*)
elseif *a* ∈ *SCOPE* **then** EVALSCOPE(*a*)
elseif *a* ∈ *SET* **then** EVALSET(*a*)
elseif *a* ∈ *SETRANGECHECKVALUE* **then** EVALSETRANGECHECKVALUE(*a*)
elseif *a* ∈ *SKIP* **then** EVALSKIP(*a*)
elseif *a* ∈ *STOP* **then** EVALSTOP(*a*)
elseif *a* ∈ *SYSTEMVALUE* **then** EVALSYSTEMVALUE(*a*)
elseif *a* ∈ *TASK* **then** EVALTASK(*a*)
elseif *a* ∈ *TIMERACTIVE* **then** EVALTIMERACTIVE(*a*)
elseif *a* ∈ *TIMERREMAINING* **then** EVALTIMERREMAINING(*a*)
elseif *a* ∈ *TYPECHECK* **then** EVALTYPECHECK(*a*)
elseif *a* ∈ *TYPECOERCION* **then** EVALTYPECOERCION(*a*)
else EVALVAR(*a*) //*a* ∈ *VAR*
elseif *a* ∈ *ZDECODING* **then** EVALZDECODING(*a*)
elseif *a* ∈ *ZENCODING* **then** EVALZENCODING(*a*)
elseif *a* ∈ *ZENCODINGPAR* **then** EVALZENCODINGPAR(*a*)
elseif *a* ∈ *ZENCODINGVAL* **then** EVALZENCODINGVAL(*a*)
endif

F3.2.1.4.2 Primitive AnyValue

Explanation

The AnyValue primitive computes the any expression.

Representation

ANYVALUE =$_{def}$ { (*id*, *cl*) ∈ *Identifier* × *CONTINUELABEL* :
id.refersto1 ∈ *Interface-definition* ∪ *Value-data-type-definition* // id refers to a data type definition
∨ *id.refersto1* ∈ *Syntype-definition* } // or id refers to a syntype

Behaviour

EVALANYVALUE(*a*: *ANYVALUE*) ≡
value(*Self.currentLabel*, *Self*) := *selectAnyValue*(*a*.**s**-*Identifier*) // id refers to sort
Self.currentLabel := *a*.**s**-*CONTINUELABEL*

The *selectAnyValue* function returns the *nullPid* for a pid sort, a random value of the sort for other sorts and *undefined* if the sort has no values.

```
selectAnyValue(id: Identifier): VALUE =def // sort identifier
    if id.refersto1 ∈ Interface-definition then nullPid
    else // id should refer to a Value-data-type-definition
        take( {v |
                v ∈ VALUE
            ∧   if   id.refersto1 ∈ Value-data-type-definition
                then   v.sort = id // one of the values of the type
                else // id refers to Syntype-definition
                        v.sort = id.refersto1.s-Identifier // value of parent sort of syntype
                    ∧   rangeCheck(id.refersto1, v) // in the range for the syntype
                endif
        } ) // take
    endif
```

F3.2.1.4.3 Primitive AgentInstPid

Explanation

The *AGENTINSTPID* primitive models an agent instance Pid value expression that gives the Pid value for the identified agent instance.

Representation

The domain *AGENTINSTPID* is defined as a Cartesian product of an *Agent-instance-pid-value*, a *VALUELABEL* list (for the evaluated *Instance-number* expressions in order for each *Agent-instance* in the *Agent-instance* list of the *Agent-instance-pid-value*), and a *CONTINUELABEL*.

```
AGENTINSTPID =def { (aipd, nums, cl) ∈ Agent-instance-pid-value × VALUELABEL* × CONTINUELABEL
    :   | aipd.s-Agent-instance-seq | = | nums |  // same number of Agent-instance items as nums
    }
```

Behaviour

The action *AGENTINSTPID* returns the pid for the identified instance, in the value associated with *Self.currentLabel*.

```
EVALAGENTINSTPID(a(aipd, nums, cl) : AGENTINSTPID) ≡
    value(Self.currentLabel, Self) :=
        if aipd.s-Agent-instance-seq = empty
        then system.agentSetPids[1] // system Pid
        else
            getAipd(system.agentSetPids[1], aipd.s-Agent-instance-seq, values(nums,Self))
        endif // system or other Pid returned as the value of the currentLabel
    Self.currentLabel := cl
```

Auxiliary functions

Function *getAipd* finds the Pid for an agent instance identified by the Pid of the owning instance, the *Agent-instance* list (within the owning instance) and the corresponding list of evaluated values to select the Pid value of an agent within an agent set.

```
getAipd(ownerpid: PID, instlist: Agent-instance*, numlist: SDLINTEGER*): PID =def
    let owneragent= ownerpid.s-SDLAGENT in // owning agent instance
    let ownertype = owneragent.agentAS1.s-Identifier.refersto1 in // type of owning instance
    let agent = // NOT unique, but a member of a unique sdlagentset
        take({ sdlagent ∈ SDLAGENT
        :       sdlagent.owner = owneragent // owner is sdl agent with given Pid
            ∧   sdlagent.agentAS1 = agentdef // agent def in AS1
            ∧   agentdef ∈ ownertype.s-Agent-definition-set // agent def member of defs in owner type
            ∧   agentdef.s-Name = instlist[1].s-Name // agent def name = given name
        }) in
    let agentset = take({ sdlagentset ∈ SDLAGENTSET : agent ∈ sdlagentset }) in
```

let *agentpid* = *agentset.agentSetPids*[*numlist*[1]] **in**
if *instlist.length* = 1
then *agentpid*
else *getAipd*(*agentpid, instlist.tail, numlist.tail*)
endif
endlet // agentpid
endlet // agentset
endlet // agent
endlet // ownertype
endlet // owneragent

F3.2.1.4.4 Primitives AssignParameter and AssignSignal, AssignSignalPar, AssignSignalVal

Explanation

These primitives are used when a signal is received via an input.

The *ASSIGNPARAMETER* primitive is used for the assignment of a value conveyed by a signal to a receiving parameter variable. An action of type *ASSIGNPARAMETER* is defined as a tuple consisting of a variable identifier, a natural number, and a continue label.

The *ASSIGNSIGNAL* primitive is used for the assignment of the values conveyed by a signal to a choice variable to hold the conveyed signal value. An action of type *ASSIGNSIGNAL* is defined as a tuple consisting of a variable identifier, a value label, and a continue label. The *ASSIGNSIGNALPAR* primitive evaluates a structure value to be used as the choice value for the signal. The *ASSIGNSIGNALVAL* primitive evaluates the choice value to be assigned to the choice variable, based on the choice variable, the signal, and the signal value (as a structure as a value label from *ASSIGNSIGNALPAR*). The result is used as the value label for *ASSIGNSIGNAL*.

Representation

An action of type *ASSIGNPARAMETER* is defined as a tuple consisting of a variable identifier, a natural number selecting one of a sequence of parameters, and a continue label.

ASSIGNPARAMETER $=_{\text{def}}$ { (*id, n, cl*) ∈ *Identifier* × *NAT* × *CONTINUELABEL*: *id.refersto1* ∈ *Variable-definition*}

An action of type *ASSIGNSIGNAL* is defined as a tuple consisting of a variable identifier, a signal identifier, and a continue label.

ASSIGNSIGNAL $=_{\text{def}}$ { (*inch, vl, cl*) ∈ *In-choice* × *VALUELABEL* × *CONTINUELABEL*
: *inch*.**s**-*Identifier.refersto1* ∈ *Variable-definition*}

An action of type *ASSIGNSIGNALPAR* is defined as a tuple consisting of a signal identifier, a value label for the returned value and a continue label.

ASSIGNSIGNALPAR $=_{\text{def}}$ { (*sigid, rl, cl*) ∈ *Identifier* × *VALUELABEL* × *CONTINUELABEL*
: *sigid.refersto1* ∈ *Signal-definition*}

An action of type *ASSIGNSIGNALVAL* is defined as a tuple consisting of a variable identifier for the choice variable to hold the signal, a signal identifier, a value label for the structure value to modify the choice and a continue label.

ASSIGNSIGNALVAL $=_{\text{def}}$ { (*inch, sigid, vl, cl*)
∈ *In-choice* × *Identifier* × *VALUELABEL* × *CONTINUELABEL*
: *inch*.**s**-*Identifier.refersto1* ∈ *Variable-definition* ∧ *sigid.refersto1* ∈ *Signal-definition*}
∧ *sigid.refersto1* ∈ *Signal-definition*}

Behaviour

EVALASSIGNPARAMETER(*a*:*ASSIGNPARAMETER*) ≡
ASSIGN(*a*.**s**-*Identifier*, // variable id to assign to
Self.currentSignalInst.plainSignalValues[*a*.**s**-*NAT*], // nth parameter of signal

```
        Self.stateAgent.state,
        Self.currentStateId
    ) // Assign
    Self.currentLabel := a.s-CONTINUELABEL

EVALASSIGNSIGNAL(a:ASSIGNSIGNAL) ≡
    ASSIGN(a.s-In-choice.s-Identifier, value(a.s-VALUELABEL, Self), Self.stateAgent.state, Self.currentStateId)
    Self.currentLabel := a.s-CONTINUELABEL

EVALASSIGNSIGNALPAR(a(sigid, rl, cl):ASSIGNSIGNALPAR) ≡
    if sigid.refersto1.s-Signal-parameter-seq = empty // signal parameterless
    then
        value(rl, Self) := nullNULL // unique value null of the predefined NULL data type
        Self.currentLabel := cl
    else EVALOPERATIONAPPLICATION(mk-OPERATIONAPPLICATION(
            mk-OPLITSIGNATURE(take({os ∈ Operation-signature :
                os.s-Name = mk-Name("Make")
                ∧ os.s-Identifier-seq =
                    < sigid.refersto1.s-Signal-parameter-seq[i].s-Identifier // ith parameter sort
                    : i ∈ 1..sigid.refersto1.s-Signal-parameter-seq.length
                    > // list of sorts for Make from signal definition signal parameters
                ∧ os.s1-Identifier = sigid.asSignal1}),
            Self.currentSignalInst.plainSignalValues, // list of signal parameters – Value list
            rl, // value label for the returned value from OperationApplication used by AssignSignalVal
            cl // label for next action after EvalOperationApplication
        )) // EvalOperationApplication – leaves value in value(rl)
    endif // null or parameters

EVALASSIGNSIGNALVAL(a(inch, sigid, vl, cl):ASSIGNSIGNALVAL) ≡
    EVALOPERATIONAPPLICATION(mk-OPERATIONAPPLICATION(
        mk-OPLITSIGNATURE(take({os ∈ Operation-signature :
            os.s-Name= mk-Name(sigid.s-Name.s-TOKEN + "Modify" ) // modify choice for sig
            ∧ os.s-Identifier-seq = < sigid.asSignal1 > // list of one sort  for Modify- sort of signal
            ∧ os.s1-Identifier = inch.s-Identifier.refersto1.Identifier1} // sort of choice
        ), // mk-OpLitSignature
        < vl >, // list of one value label (for value from from EvalAssignSignalPar)
        undefined, // returned value from OperationApplication not needed
        cl // label for next action after EvalOperationApplication
    )) // EvalOperationApplication – leaves value in value(Self.currentLabel, Self)
```

Reference sections

The definition of macro ASSIGN can be found in clause F3.2.1.3.1.

F3.2.1.4.5 Primitive Break

Explanation

The Break primitive models the break operation, i.e., it leaves the current scope until the named scope is found.

Representation

```
BREAK =def { (n) ∈ Name: ∃ cn ∈ Compound-node: isAncestorAS1(cn, Break-node(n))=cn}
```

Behaviour

```
EVALBREAK(a:BREAK) ≡
        if scopeName(Self, Self.currentStateId) = a then
            Self.currentLabel := scopeContinueLabel(Self, Self.currentStateId)
        endif
        Self.currentStateId := caller(Self.stateAgent.state, Self.currentStateId)
```

F3.2.1.4.6 Primitive Call

Explanation

The call primitive models procedure calls, or method invocations. It is used within the evaluation of expressions and actions. The macro EVALCALL creates a new context (e.g., new local scope for variables, for names of its states and connectors) and saves the old context, which in turn is restored by the corresponding return.

Representation

An action of type *CALL* is defined as a tuple consisting of an identifier of the called procedure, a sequence of value labels and variable identifiers, a value label for the procedure return, and a continue label. In-parameters are represented by value labels, in/out-parameters by variable identifiers.

CALL =$_{\text{def}}$ { (*id*,*pl*,*v*,*cl*) ∈ *Identifier* × *CALLPARAM** × *VALUELABEL* × *CONTINUELABEL*:
id.refersto1 ∈ *Procedure-definition*}

CALLPARAM =$_{\text{def}}$ { (*par*) ∈ *VALUELABEL* ∪ *Identifier*: *par* ∈ *Identifier* ⇒ *par.refersto1* ∈ *Variable-definition*}

Behaviour

```
EVALCALL(a:CALL) ≡
    let pd = a.s-Identifier.refersto1 in // procedure definition
    CREATEPROCEDURE(
        pd, // procedure definition
        a.s-CALLPARAM-seq,
        a.s-VALUELABEL, // for returnLabel
        getResultRefVarId(a.s-Identifier.parentAS1, pd), // for resultRefValId
        a.s-CONTINUELABEL
    ) // CreateProcedure
    endlet // pd
```

A procedure call is evaluated with macro CREATEPROCEDURE, (see clause F3.2.3.1.4 State access) which basically performs a procedure initialization and additionally creates a procedure state node.

The parameter passing mechanism is realized by a call of function *assignParams* in the function *initProcedureState*, which is used by CREATEPROCEDUREVARIABLES used by CREATEPROCEDURE. The function *initProcedureState* returns a state, which contains *Self.state* as a substate. For all local variables and parameters *initProcedureState* "creates" new locations.

Auxiliary function

The function *getresultRefVarId* is called in EVALCALL with the surrounding context of the procedure call and the procedure definition. If the procedure has a **REF** result and if in the context surrounding the call the result is assigned to a **REF** item (a variable or parameter with *Aggregation-kind* **REF)**, the identifier for the **REF** item is returned. If the procedure has a **REF** result and the surrounding context is a *Value-return-node*, *getresultRefVarId* is called (recursively) to determine there is a **REF** item that receives the result. The result is undefined if the procedure is not recursive, or the result is not assigned to a **REF** item.

```
getresultRefVarId(parent: DEFINITIONAS1, pd: Procedure-definition): Identifier =def
    if pd.s-Result.s-Result-aggregation = REF
    then // procedure with REF result
        if      parent ∈ Assignment // is in assignment
            ∧   parent.s-Identifier.refersto1.s-Aggregation-kind = REF
        then parent.s-Identifier // assignment to REF var
        elseif
                parent ∈ Actual-parameters // parent is actual parameter list
            ∧   parent.parentAS1 ∈ (Call-node ∪ Value-returning-call-node) // in call node
```

```
        ∧  (∃! i ∈ 1..parent.length : // exactly one actual parameter parent[i] such that
              parent[i] = a.s-Identifier // parameter is the given id
           ∧  refersto1(parent.parentAS1.s-Identifier  //procedure id for act params refers to proc def
              ).s-PROCPARAM-seq[i].s-Parameter-aggregation.s-Aggregation-kind = REF
              // corresponding formal parameter has REF aggregation
           ) // exists meeting conditions
     then
     let i ∈ 1..parent.length : // exactly one actual parameter parent[i] such that
        parent[i] = a.s-Identifier // parameter is the given id
     ∧  refersto1(parent.parentAS1.s-Identifier  //procedure id for act params refers to proc def
        ).s-PROCPARAM-seq[i].s-Parameter-aggregation.s-Aggregation-kind = REF
        // corresponding formal parameter has REF aggregation
     in
        refersto1(parent.parentAS1.s-Identifier  //procedure id refers to proc definition
        ).s-PROCPARAM-seq[i].identifier1 // id of formal parameter with REF aggregation
     endlet
     elseif
           parent ∈ Value-return-node
        ∧  Self.callingProcedureNode.procedureAS1.s-Result.s-Result-aggregation = REF
     then // Call used in Value-return-node.
        getresultRefVarId(
           { callnode ∈ DEFINITIONAS1 :
              uniqueLabel(callnode, 1) = Self.callingProcedureNode.resultLabel
           }.take , // find the call node for the resultLabel
           Self.callingProcedureNode.procedureAS1
        )
     else undefined // none of the above cases
     endif
  else undefined
  endif // result Ref Var Id
```

F3.2.1.4.7 Primitive Continue

Explanation

The Continue primitive is used for modelling the loop continue operation.

Representation

$CONTINUE =_{def} \{ (n) \in Name:$
$\exists\ cn \in Compound\text{-}node: \exists\ c \in Continue\text{-}node(n): parentAS1ofKind(c, Compound\text{-}node) = cn\}$

Behaviour

```
EVALCONTINUE(a:CONTINUE) ≡
   if scopeName(Self, Self.currentStateId) = a then
      Self.currentLabel := scopeStepLabel(Self, Self.currentStateId)
   else
      Self.currentStateId := caller(Self.stateAgent.state, Self.currentStateId)
   endif
```

F3.2.1.4.8 Primitive Create

Explanation

The Create primitive specifies the creation of an SDL-2010 agent. An action of type *CREATE* is defined by a tuple consisting of an agent-definition, a sequence of value labels, and a continue label.

Representation

An action of type *CREATE* is defined as tuple consisting of an agent-definition, a sequence of value labels, and a continue label.

CREATE =$_{def}$ { (*id*, *vl*, *cl*) ∈ *Identifier* × *VALUELABEL** × *CONTINUELABEL*: *id.refersto1* ∈ *Agent-definition*}

Behaviour

```
EVALCREATE(a:CREATE) ≡
    let sas = take({as ∈ SDLAGENTSET: as.agentAS1 = a.s-Identifier.refersto1 }) in
        if sas.agentAS1.s-Number-of-instances.s2-NAT ≠ undefined // Maximum-number
        then
            let n = |{ sa ∈ SDLAGENT: sa.owner = sas }| in
            if n < sas.agentAS1.s-Number-of-instances.s2-NAT // Maximum-number
            then    CREATEAGENT(sas, Self, sas.agentAS1)
            else    Self.offspring := nullPid
            endif // n
            endlet
        else
            CREATEAGENT(sas, Self, sas.agentAS1)
        endif
    endlet
    Self.currentLabel := a.s-CONTINUELABEL
```

Reference sections

For the definition of the macro CREATEAGENT see clause F3.2.3.1.3.

F3.2.1.4.9 Primitive Decision

Explanation

The Decision primitive is used for the evaluation of decisions. A decision in *DECISION* consists of a value label and a set of answer. An answer in *ANSWER* is a tuple consisting of a value label and a continue label. The action itself chooses an answer such that the decision-value given by the corresponding value label coincides with the answer-value.

Representation

A *DECISION* consists of a value label and a set of answers. An answer in *ANSWER* is a tuple consisting of an optional value label and a continue label.

DECISION =$_{def}$ *VALUELABEL* × *ANSWER*-**set** × [*CONTINUELABEL*]

ANSWER =$_{def}$ [*VALUELABEL*] × *CONTINUELABEL*

A *DECISIONANYVALUE* consists of a value label, a set of transitions and a continue label, and is used for a random value in the range 1 to the number of transitions, so that a random transition is selected in an *Any-decision*.

DECISIONANYVALUE =$_{def}$ *VALUELABEL* × *Transition*-**set** × *CONTINUELABEL*

Behaviour

Macro EVALDECISION chooses an answer such that the decision-value given by the corresponding value label coincides with the answer-value. If the value label of an *ANSWER* is undefined, this represents an SDL false value.

```
EVALDECISION(d:DECISION) ≡
    if value(d.s-VALUELABEL, Self) ∈ { value(an.s-VALUELABEL, Self) | an ∈ d.s-ANSWER-set } then
        choose an: an ∈ d.s-ANSWER-set ∧
            value(d.s-VALUELABEL, Self) =
                if value(an.s-VALUELABEL, Self) ≠ undefined
                then value(an.s-VALUELABEL, Self)
                else mk-SDLBOOLEAN(false, BooleanType)
                endif //
```

```
        Self.currentLabel := an.s-CONTINUELABEL
      endchoose
    elseif d.s-CONTINUELABEL ≠ undefined then
      Self.currentLabel := d.s-CONTINUELABEL
    else raise(NoMatchingAnswer)
    endif
```

Macro EVALDECISIONANYVALUE chooses an answer such that a random transition of the *Any-decision* is chosen.

```
EVALDECISIONANYVALUE(d:DECISIONANYVALUE) ≡
  value(d.s-VALUELABEL, Self) := take({i : i ∈ 1..| d.s-Transition-set | } // random value
  Self.currentLabel := d.s-CONTINUELABEL
```

Reference sections

For the definition of function *value* refer to clause F3.2.1.4.1.

F3.2.1.4.10 Primitive EnterStateNode

Explanation

State nodes are entered when an SDL-2010 agent has been created, and at the end of each transition. Also, state nodes are entered when a procedure is invoked. The evaluation of the primitive starts the sequence of steps needed to enter a given state node, which may include the entering of composite states and the execution of start transitions and entry procedures.

Representation

ENTERSTATENODE $=_{\text{def}}$ (*Name* ∪ **HISTORY**) × *STATEENTRYPOINT* × *VALUELABEL**

Behaviour

```
EVALENTERSTATENODE(a:ENTERSTATENODE) ≡
  let enterName: (Name ∪ HISTORY) = a.s-implicit in
    if enterName = HISTORY then
      Self.stateNodesToBeEntered :=
        {mk-STATENODEWITHENTRYPOINT(Self.previousStateNode, HISTORY)}
    else
      choose sn: sn ∈ STATENODE ∧ sn.stateName = enterName ∧
        sn.stateNodeKind = stateNode ∧ sn.parentStateNode = Self.currentParentStateNode
        Self.stateNodesToBeEntered :=
          {mk-STATENODEWITHENTRYPOINT(sn, a.s-STATEENTRYPOINT)}
      endchoose
    endif
    let snwep ∈ Self.stateNodesToBeEntered in
      if snwep.stateAS1.s-State-timer ≠ undefined
      then
      let stm = snwep.stateAS1.s-State-timer in
      let tmi =
        mk-TIMERINST(Self.selfPid, stm.s-Identifier,
            < value(uniquelabel(stm.s-EXPRESSION[i], 1), Self) : ∈ 1.. stm.s-EXPRESSION-seq.length >
        ) // mk-TimerInst
      in
        if tmi.active = false then // timer not active
          if value(uniquelabel(stm.s-EXPRESSION, 1), Self) = undefined then
            // no time given, derive from timer def
            Self.inport.schedule :=
              insert(tmi, now + stm.s-Identifier.duration, delete(tmi, Self.inport.schedule))
            tmi.arrival := now + tm.duration
          else // time given
            Self.inport.schedule :=
              insert(tmi, value(uniquelabel(stm.s-EXPRESSION, 1), Self),
```

```
                        delete(tmi, Self.inport.schedule))
                    tmi.arrival := value(uniquelabel(stm.s-EXPRESSION, 1), Self)
                endif // no time given / time given
            endif // timer not active
        endlet // tmi
        endlet // stm
        endif // there is a state tmer
    endlet // snwep
    Self.agentMode3 := enteringStateNode
    Self.agentMode4 := startPhase
    Self.currentLabel := undefined
    Self.continueLabel := undefined
endlet // enterName
```

Given the *State-name* and the *currentParentStateNode*, the state node to be entered is determined. This has to be done at execution time, as the state node instance is not known during compilation. Agent modes are set such that the sequence of steps needed to enter the state node is performed.

Reference sections

See also clause F3.2.3.2.15.

F3.2.1.4.11 Primitive Equality

Explanation

The Equality primitive is used for the evaluation of equality tests. An action of type *EQUALITY* is defined as a tuple consisting of two value labels and a continue label. The values associated with these labels are compared. The result is stored at continue label.

Representation

$EQUALITY =_{def} VALUELABEL \times VALUELABEL \times CONTINUELABEL \times BOOLEAN$

The *BOOLEAN* is true for equality and false for negative equality.

Behaviour

```
EVALEQUALITY (a:EQUALITY) ≡
    if (    a.s-BOOLEAN ∧ (value(a.s-VALUELABEL, Self) = value(a.s2-VALUELABEL, Self))) ∨
       (¬ a.s-BOOLEAN ∧ ¬(value(a.s-VALUELABEL, Self) = value(a.s2-VALUELABEL, Self))) then
       value(a.s-CONTINUELABEL, Self) := mk-SDLBOOLEAN(true, BooleanType)
    else
       value(a.s-CONTINUELABEL, Self) := mk-SDLBOOLEAN(false, BooleanType)
    endif
    Self.currentLabel := a.s-CONTINUELABEL
```

Reference sections

No references.

F3.2.1.4.12 Primitive LeaveStateNode

Explanation

State nodes are left at the start of transitions.

Representation

$LEAVESTATENODE =_{def} Transition \times CONTINUELABEL$

A *LEAVESTATENODE* consists of the *Transition* that is to be fired and a continue label of the node or endnode to which control is passed. *LEAVESTATENODE* is only compiled for *t*:*Transition* if *t.parentAS1.parentAS1* ∈ *State-node*, therefore this does not need checking again here. The state

node name is derived from *Transition*. The *Transition* is used as a parameter rather than the state *Name*, to determine the special case of a *Transition* is that has no *Graph-node* items and ends in a *Terminator* that is a *Nextstate-node* leading back to the same state node. In this special case, an active *State-timer* of the state node remains active; otherwise an active *State-timer* of the state node is cancelled.

Behaviour

```
EVALLEAVESTATENODE(a:LEAVESTATENODE) ≡
    choose sn:
            sn ∈ STATENODE
        ∧ sn.stateName = a.s-Transition.parentAS1.parentAS1.s-Name // State-name
        ∧ sn.stateNodeKind = stateNode
        ∧ sn.parentStateNode = Self.currentParentStateNode
        // assertion: sn = Self.previousStateNode
        if sn.stateAS1.s-State-timer ≠ undefined
        then
        let stm = sn.stateAS1.s-State-timer in
        let tmi =
            mk-TIMERINST(Self.selfPid, stm.s-Identifier,
                < value(uniquelabel(stm.s-EXPRESSION[i], 1), Self) : ∈ 1.. stm.s-EXPRESSION-seq.length >
            ) // mk-TimerInst
        in
            if   tmi.active = true // timer active
            ∧ ¬(    a.s-Transition.s-Graph-node-seq = empty
                  ∧ a.s-Transition.s-implicit ∈ Terminator
                  ∧   (    ( a.s-Transition.s-implicit.s-Terminator ∈ Named-nextstate
                           ∧ a.s-Transition.s-implicit.s-Terminator.s-Name
                             =  a.s-Transition.parentAS1.parentAS1.s-Name
                           ) // nextstate same state
                       ∨   a.s-Transition.s-implicit.s-Terminator ∈ Dash-nextstate
                       ) // nextstate same state OR Dash nextstate
                ) // AND NOT (empty graph with (nextstate same state OR Dash nextstate))
            then    delete(tmi, Self.inport.schedule)
            endif // timer active and not special case of empty graph back to same state
        endlet // tmi
        endlet // stm
        endif // there is a state tmer
        Self.stateNodesToBeLeft := collectCurrentSubStates(sn)
    endchoose
    Self.agentMode3 := leavingStateNode
    Self.agentMode4 := leavePhase
    Self.currentLabel := undefined
    Self.continueLabel := a.s-CONTINUELABEL
```

Given the *State-name* and the *currentParentStateNode*, the state node to be left is determined. This has to be done at execution time, as the state node instance is not known during compilation. Agent modes are set such that the sequence of steps needed to leave the state node is performed.

Reference sections

See also clause F3.2.3.2.16 for information on how state nodes are left.

F3.2.1.4.13 Primitive OperationApplication

Explanation

The OperationApplication primitive models the application of operators. Procedures without procedure body are called functional or predefined procedures. In this sense, all built-in operators such as +, – on the set of integers are predefined procedures. A predefined (*functional* – see F3.2.1.3.1 Predefined data) operation is executed by function *compute* (see F3.2.1.3.1

Predefined data) and the result assigned to the controlled function *value* (see F3.2.1.4.1 Action evaluation). A non-functional operation is handled with function *dispatch* (see F3.3.7 Operators and methods) that determines (depending on the current values) the correct procedure identifier invoked though CREATEPROCEDURE (see F3.2.3.1.4 Procedure creation and initialization).

Normally the primitive has a *VALUELABEL* list as its second parameter, but is also called with a *VALUE* list (for example from EVALASSIGNSIGNALPAR with the *VALUE* list *plainSignalValues* of *Self.currentSignalInst*). If a *VALUELABEL* list is given it is changed to a *VALUE* list before invoking the procedure for the operation. There is an optional *VALUELABEL* for the returned value: if this is undefined, the value returned by the called procedure is in *Self.currentLabel*.

Representation

OPERATIONAPPLICATION $=_{\text{def}}$ *OPLITSIGNATURE* × (*VALUELABEL** ∪ *VALUE**) × [*VALUELABEL*] × *CONTINUELABEL*

Behaviour

```
EVALOPERATIONAPPLICATION(a(opsig, vseq, rl, cl) :OPERATIONAPPLICATION) ≡
    let valseq = if vseq ∈ VALUELABEL-seq then values(a.s-VALUELABEL-seq, Self) else vseq endif in
    if functional(opsig, valseq) then
        value(if rl ≠ undefined then rl else Self.currentLabel endif, Self):= compute(opsig, valseq)
        Self.currentLabel:= cl
    else
        CREATEPROCEDURE(
            dispatch(opsig, valseq).refersto1,
            valseq,
            if rl ≠ undefined then rl else Self.currentLabel endif,
            cl
        ) // CreateProcedure
    endif
    endlet // valseq
```

F3.2.1.4.14 Primitives Output, OutputX and OutputZ

Explanation

The *OUTPUT* primitive is used for expressing an output with a signal parameter. The *OUTPUTX* primitive is used for expressing an output with an expression parameter. The *OUTPUTZ* primitive is used for expressing an output with an encoded expression parameter.

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. See clause 11.13.4 of [ITU-T Z.104].

Representation

An action of type *OUTPUT* consists of: a signal type, a sequence of value labels for the parameters of the signal, a value label for the activation delay, a value label for the (signal) priority, an optional argument specifying a value label or agent identifier for the destination, an optional value label for the destination index number, an argument specifying a via path, and a continue label.

OUTPUT $=_{\text{def}}$ *SIGNAL* × *VALUELABEL** × *VALUELABEL* × *VALUELABEL*
× [*VALUELABEL* ∪ *Identifier* ∪ **THIS**] × [*VALUELABEL*] × *VIAARG* × *CONTINUELABEL*

An action of type *OUTPUTX* consists of a value label for the expression used in the output, a value label for the activation delay, a value label for the (signal) priority, an optional argument specifying a value label or agent identifier for the destination, an optional value label for the destination index number, an argument specifying a via path, and a continue label.

OUTPUTX =def *VALUELABEL* × *VALUELABEL* × *VALUELABEL*
× [*VALUELABEL* ∪ *Identifier* ∪ **THIS**] × [*VALUELABEL*] × *VIAARG* × *CONTINUELABEL*

An action of type *OUTPUTZ* consists of a value label for a choice value expression used in the output, a value label for the activation delay, a value label for the (signal) priority, an optional argument specifying a value label or agent identifier for the destination, an optional value label for the destination index number, an argument specifying a via path, and a continue label.

OUTPUTZ =def *VALUELABEL* × *VALUELABEL* × *VALUELABEL*
× [*VALUELABEL* ∪ *Identifier* ∪ **THIS**] × [*VALUELABEL*] × *VIAARG* × *CONTINUELABEL*

Behaviour

Macro EVALOUTPUT defines signal output by macro SIGNALOUTPUT, which takes the signal, a value sequence, the destination and the path as arguments.

```
EVALOUTPUT(a(sigid, sigparams, actdelay, sigpri, dest, destnum, via, cl):OUTPUT) ≡
        // this is invoked only from Eval in F3.2.1.4.1 Action evaluation
        SIGNALOUTPUT(a.s-SIGNAL, // sigid
            values(a.s-VALUELABEL-seq, Self), // sigparams
            mk-DURATION(
                value(actdelay, Self).s-REAL
            ), //activation delay - label value->SDLDuration, SDLDuration Real -> Duration Real
            if value(sigpri, Self).semvalueInt ≥ 0 // has to be a Nat value
            then value(sigpri, Self).semvalueInt
            else undefined
            endif, // signal priority –Nat value (or undefined)
            if dest = undefined
            then undefined
            else toPid(dest, destnum)
            endif,
            a.s-VIAARG
        )
        Self.currentLabel := a.s-CONTINUELABEL
where

toPid(dest: VALUELABEL ∪ Identifier ∪ THIS, destnum: [ VALUELABEL ] ): PID =def
    if dest ∈ VALUELABEL then value(dest, Self) // the expression should give a Pid
    elseif dest ∈ Identifier
    then // dest is Identifier
        let sas = take({as ∈ SDLAGENTSET : as.identifier1 = dest}) in
        if destnum ∈ VALUELABEL
        then sas.agentSetPids[value(destnum, Self)]
        else take(sas.agentSetPids.toSet)
        endif
        endlet // sas
    else // dest is THIS
        if destnum ∈ VALUELABEL
        then Self.agentSetPids[value(destnum, Self)]
        else take(Self.agentSetPids.toSet)
        endif
    endif
endwhere
```

A signal output operation causes the creation of a new signal instance. The agent instance initiating the output operation identifies itself as sender of the signal instance by setting a corresponding function *signalSender* defined on signals. In general, there may be none, one or more output gates of an agent to which a signal can be delivered depending on the specified constraints on:

- possible destinations,
- potential receivers and

- admissible paths,

as stated by the values of *ToArg* and *ViaArg*, which are obtained as parameters of an output operation and are assigned to a signal by setting corresponding functions defined on signals. Possible ambiguities are resolved by a non-deterministic choice for a gate that is connected to a path being *compatible* with *ToArg*, *ViaArg*. In the rule below, this choice is stated in abstract terms using the predicate *applicable* (see clause F3.2.1.1.4). If the constraints cannot be met, the signal instance is discarded.

When the output is interpreted, the *Activation-delay* is added to the current value of **now** to determine the availability Time: that is, the time after which the signal is made available in the input port of the destination. See clause 11.13.4 *Semantics* of [ITU-T Z.101].

```
SIGNALOUTPUT(s:SIGNAL, vSeq:VALUE*, actdelay:DURATION, sigpri:NAT,
    toArg:[TOARG], viaArg:VIAARG) ≡
    if s.refersto1.s-Abstract ≠ undefined // abstract signal definition
    then   raise(InvalidCall) // attempt to instantiate abstract signal definition
    elseif toArg ∈ PID ∧ s.refersto1 ∉ toArg.s-Interface-definition.s-Signal-definition-set
    then   raise(InvalidReference) // signal not receivable at given destination
    else
        choose g: g ∈ (Self.outgates ∪ Self.ingates) ∧ applicable(s, toArg, viaArg, g, undefined)
            extend PLAINSIGNALINST with si
                si.plainSignalType:= s
                si.plainSignalValues := vSeq
                si.availabilityTime := now + actdelay
                si.signalPriority := sigpri
                si.toArg := toArg
                si.viaArg := viaArg
                si.plainSignalSender := Self.selfPid
                INSERT(si, now, g)
            endextend
        endchoose
    endif
```

Macro EVALOUTPUTX defines signal output by macro SIGNALOUTPUT when an Output is used with an expression (that is an appropriate choice value) rather than a signal with a list of parameters. The signal and parameters for SIGNALOUTPUT are derived from the expression and the other parameters are handled in the same way as in the macro EVALOUTPUT (above).

```
EVALOUTPUTX(a(exlabel, actdelay, sigpri, dest, destnum, via, cl):OUTPUT) ≡
        // this is invoked only from Eval in F3.2.1.4.1 Action evaluation
        let ex = value(exlabel, Self) in
        let sigstructval =
            take({ex.s-FIELD-seq[i].s-VALUE // value of choice field that is defined
                :       i ∈ 1..ex.s-FIELD-seq.length
                    ∧   ex.s-FIELD-seq[i].s-VALUE ≠ undefined
            }) // set for sigstructval, take
        in
        SIGNALOUTPUT(
            take({sigid: sigid ∈ Identifier
                ∧ sigid.refersto1 ∈ Signal-definition // id for a signal definition
                ∧  (   (   sigid.s-Name = sigstructval.s-Name // name of choice field that is defined
                               ∧ | { sigdef ∈ Signal-definition : sigid.s-Name = sigdef.s-Name } | = 1
                           ) // 11.13.4 Z.104, (a) name unambiguously identifies a signal SEP
                           ∨
                           (   sigstructval.s-Name = sigid.asSignal1.s-Name
                           ) // 11.13.4 Z.104, (b) name is unique sort for <as signal> of a signal
                    ) // 11.13.4 Z.104, case (a) or (b)
                ∧  (∀i ∈ 1..sigstructval.s-FIELD-seq.length:
                        sigstructval.s-FIELD-seq[i].s-VALUE.s-Identifier = // sort structure field
                            sigid.refersto1.s-Signal-parameter-seq[i].s-Identifier // sort signal parameter
```

```
                ) // 11.13.4 Z.104, corresponding by position sorts - structure and signal
            }),  // sigid set, take
            < sigstructval.s-FIELD-seq[i].s-VALUE : i ∈ 1..sigstructval.s-FIELD-seq.length >, // sigparams
            mk-DURATION(
                value(actdelay, Self).s-REAL
            ), //activation delay - label value->SDLDuration, SDLDuration Real -> Duration Real
            if value(sigpri, Self).semvalueInt ≥ 0 // has to be a Nat value
            then value(sigpri, Self).semvalueInt
            else undefined
            endif, // signal priority –Nat value (or undefined)
            if dest = undefined
            then undefined
            else toPid(dest, destnum)
            endif,
            via
        )
        Self.currentLabel := cl
        endlet // sigstructval
        endlet // ex
where
    // toPid is defined in the where of EvalOutput
endwhere
```

Macro EVALOUTPUTZ is used when there is an output for an *Encoded-expression*, rather than a signal with a list of parameters or a choice value. The *Encoded-expression* is decoded, and a value label for the decoded expression passed to EVALOUTPUTZ, which is then a value label for a signal output passed to EVALOUTPUTX.

```
EVALOUTPUTZ(a(cvlabel, actdelay, sigpri, dest, destnum, via, cl):OUTPUTZ) ≡
        // this is invoked only from Eval in F3.2.1.4.1 Action evaluation
        EVALOUTPUTX(cvlabel, actdelay, sigpri, dest, destnum, via, cl)
```

Reference sections

Definitions of functions associated with signals can be found in clause F3.2.1.1.1.

F3.2.1.4.15 Primitive RefFieldAssign

Explanation

The RefFieldAssign primitive is used when there is an assignment of an operation application for a field access to a variable with **REF** aggregation.

Representation

An action of type reset is defined as tuple consisting of a timer, a sequence of value labels, and a continue label.

```
REFFIELDASSIGN =def
    { (id, op(opid, Actual-parameters(exprList)), cl) ∈ Identifier × Operation-application × CONTINUELABEL
    :       id.refersto1 ∈ VARDEF
        ∧   exprList.length = 1
        ∧   exprList[1] ∈ Variable-access
        ∧   (   exprList[1].s-Identifier.refersto1.s-Identifier.isStructSort
                ∨   exprList[1].s-Identifier.refersto1.s-Identifier.isChoiceSort
            )
        ∧   substring(opid.refersto1.s-Name.s-TOKEN, opid.refersto1.s-Name.s-TOKEN.length-6,7) = "Extract"
    }
```

Behaviour

Macro EVALREFFIELDASSIGN specifies the assignment.

```
EVALREFFIELDASSIGN(a(id, op(opid, Actual-parameters(exprList)), cl):REFFIELDASSIGN) ≡
    update(id,
        mk-BOUNDVALUE(
            mk-FIELDREF(
                exprList[1], // variable/parameter id
                // field name
                substring(opid.refersto1.s-Name.s-TOKEN, 1, opid.refersto1.s-Name.s-TOKEN.length-7)
            ) // FieldRef
        ), // BoundValue
        Self.stateAgent.state,
        Self.currentStateId
    ) // update binding – that is, do assignment
    Self.currentLabel := cl
```

F3.2.1.4.16 Primitive Reset

Explanation

The Reset primitive is used for expressing a timer reset. An action of type reset is defined as tuple consisting of a timer, a sequence of value labels, and a continue label. The primitive specifies a reset of a timer with macro RESETTIMER.

Representation

An action of type reset is defined as tuple consisting of a timer, a sequence of value labels, and a continue label.

$RESET =_{def} TIMER \times VALUELABEL^* \times CONTINUELABEL$

Behaviour

Macro EVALRESET specifies a reset of a timer with macro RESETTIMER.

```
EVALRESET(a:RESET) ≡
    RESETTIMER(a.s-TIMER, values( a.s-VALUELABEL-seq, Self))
    Self.currentLabel := a.s-CONTINUELABEL
```

Reference sections

The definition of macro RESETTIMER can be found in clause F3.2.1.1.5.

F3.2.1.4.17 Primitive Return

Explanation

The Return primitive is used to model a procedure, method or operator return, or the exit of a composite state. In case of a procedure, method or operator return, it basically restores the old context (e.g., local scope for names of its states and connectors) of the corresponding call. Since procedures can return values, an action of type *RETURN* is modelled by a value label. The return value of the procedure is stored at this label. In case of an exit, the state exit point name is given.

Representation

$RETURN =_{def} VALUELABEL \cup STATEEXITPOINT$

Behaviour

```
EVALRETURN(a: RETURN) ≡
    if a ∈ VALUELABEL then
```

EVALEXITPROCEDURE(*a*)
else
EVALEXITCOMPOSITESTATE(*a*)
endif

EVALEXITPROCEDURE(*vl*: *VALUELABEL*) ≡
value(*Self.callingProcedureNode.resultLabel, Self*) := *value*(*vl, Self*)
RESTOREPROCEDURECONTROLBLOCK(*Self.callingProcedureNode*)

EVALEXITCOMPOSITESTATE(*sep*: *STATEEXITPOINT*) ≡
Self.stateNodeToBeExited :=
mk-*STATENODEWITHEXITPOINT*(*Self.currentParentStateNode, sep*)
Self.agentMode3 := *exitingCompositeState*

RESTOREPROCEDURECONTROLBLOCK(*sn*:*STATENODE*) ≡
Self.agentMode1 := *sn.agentMode1*
Self.agentMode2 := *sn.agentMode2*
Self.agentMode3 := *sn.agentMode3*
Self.agentMode4 := *sn.agentMode4*
Self.agentMode5 := *sn.agentMode5*
Self.currentStateId := *sn.currentStateId*
Self.currentLabel := *sn.continueLabel*
Self.continueLabel := *sn.continueLabel*
Self.currentParentStateNode := *sn.currentParentStateNode*
Self.previousStateNode := *sn.previousStateNode*
Self.callingProcedureNode := *sn.callingProcedureNode*

Reference sections

Information on procedure control blocks is given in clause F3.2.1.2.3.

F3.2.1.4.18 Primitive Scope

Explanation

The Scope primitive creates a new scope for use in a compound node.

Representation

SCOPE =$_{\text{def}}$ *Compound-node*.**s-***Name* × *Variable-definition*-**set** × *STARTLABEL* × *STEPLABEL* × *CONTINUELABEL*

STEPLABEL =$_{\text{def}}$ *LABEL*

Behaviour

EVALSCOPE(*a*:*SCOPE*) ≡
CREATECOMPOUNDNODEVARIABLES(*Self, a*)
Self.currentLabel := *a*.**s-***STARTLABEL*

Reference sections

See also clause F3.2.3.1.8.

F3.2.1.4.19 Primitive Set

Explanation

The Set primitive is used for expressing a timer set. An action of type *SET* is defined as tuple consisting of a time label, a timer, a sequence of value labels, and a continue label. The action itself is mainly defined by macro SETTIMER.

Representation

An action of type *SET* is defined as tuple consisting of a time label, a timer, a sequence of value labels, and a continue label.

SET =$_{def}$ *TIMELABEL* × *TIMER* × *VALUELABEL** × *CONTINUELABEL*

Domains

TIMELABEL =$_{def}$ *VALUELABEL*

Behaviour

Macro EVALSET defines the setting of a timer by macro SETTIMER.

EVALSET(*a*:*SET*) ≡
    SETTIMER(*a*.**s**-*TIMER*, *values*(*a*.**s**-*VALUELABEL*-**seq**, *Self*), *semvalueReal*(*value*(*a*.**s**-*TIMELABEL*,*Self*)))
    Self.currentLabel := *a*.**s**-*CONTINUELABEL*

Reference sections

The definition of macro SETTIMER can be found in clause F3.2.1.1.5.

F3.2.1.4.20 Primitive SetRangeCheckLabel

Explanation

The SetRangeCheckValue primitive is used to set the value to be used in a range check.

Representation

SETRANGECHECKVALUE =$_{def}$ *VALUELABEL* × *CONTINUELABEL*

static *rangeCheckLabel*: → *LABEL*

The static function *rangeCheckLabel* denotes a special label, which is different from all other labels in the system. It is used to store the value to be used in the subsequent range check via the function *value*.

Behaviour

EVALSETRANGECHECKVALUE(*a*: *SETRANGECHECKVALUE*) ≡
    value(*rangeCheckLabel*, *Self*) := *value*(*a*.**s**-*VALUELABEL*, *Self*)
    Self.currentLabel := *a*.**s**-*CONTINUELABEL*

F3.2.1.4.21 Primitive Skip

Explanation

This is basically a no-op. It is used, for instance, to model joins.

Representation

SKIP =$_{def}$ *Join-node*.**s**-*Name* ∪ *CONTINUELABEL* // the name is a connector name

Behaviour

EVALSKIP(*a*:*SKIP*) ≡
  if *a* ∈ *Name* **then** // if it is a name, it is a connector name
    Self.stateNodeChecked := *Self.currentParentStateNode*
    Self.currentConnector := **mk**-*STATENODEWITHCONNECTOR*(*Self.currentParentStateNode*, *a*)
    Self.agentMode2 := *selectingTransition*
    Self.agentMode3 := *startSelection*
  else // if it is not a name, therefore it is a label
    Self.currentLabel := *a*
  endif

Reference sections

Clause F3.2.3.2.8.

F3.2.1.4.22 Primitive Stop

Explanation

If the number of instances in the agent instance set is not greater than the lower bound for that instance set, the predefined exception *OutOfRange* is raised.

Otherwise the Stop primitive initiates the stopping of an agent, which takes place in two phases. In the first phase, the state machine of the agent enters a stopping condition. The state machine of such an agent remains in the stopping condition until all contained agents have terminated, after which the agent terminates. While in the stopping condition, the agent will not accept any stimuli (other than the implicit set and get remote procedure calls, if any, introduced for each global variable. See clause 9 *Semantics* of [ITU-T Z.102].

The set and get remote procedure calls mentioned above, are given the names "#set_"+*varname* and "#get_"+*varname* in the transformation of the <agent type definition> for the block type definition containing the global variable *varname* (see F2.2.5.1.3). These remote procedure calls are transformed into an exchange of signals with the names *p*+"CALL" and *p*+"REPLY" (see F2.2.7.5 Remote procedures), where *p* is the result of a function (*implicitName*, see F2.2.7.5 Auxiliary functions) that is a unique name derived from a remote procedure definition. An agent in the stopping condition therefore has to handle signals with the names *p*+"CALL" for global variables. So that these signals can be distinquished from any other signals, if the remote procedure definition has the name "#set_"+*varname* or "#get_"+*varname* in these cases the function *implicitName* in F2.2.7.5 gives an arbitrary new name followed by "#"+*varname*+"#set" or "#"+*varname*+"#get" respectively. All signals not ending in "#setCALL" or "#getCALL" are discarded in the stopping condition. The *varname* is used in F3.2.3.2.18 (Stopping Agent Execution) to assign to the (global) variable, or obtain the value of the (global) variable.

The Stop primitive is used for expressing the evaluation of stop conditions.

Representation

STOP =$_{\text{def}}$ ()

Behaviour

Macro EVALSTOP specifies all actions to be taken when an agent performs a stop.

```
EVALSTOP(a:STOP) ≡
    if Self.agentAS1.s-Number-of-instances.s3-NAT // Lower-bound
        < |{ sa ∈ SDLAGENT: sa.agentAS1 = Self.agentAS1 }|
    then
        Self.agentMode2 := stopping
    else
        raise(OutOfRange)
    endif
```

Reference sections

Clause F3.2.3.2.18.

F3.2.1.4.23 Primitive SystemValue

Explanation

The SystemValue primitive computes the values of the predefined imperative operators.

Representation

SYSTEMVALUE =$_{\text{def}}$ *VALUEKIND* × *CONTINUELABEL*
VALUEKIND =$_{\text{def}}$ { *kNow, kParent, kOffspring, kSelf, kSender, kSignal, kSignallist, kState*}

Behaviour

```
EVALSYSTEMVALUE(a: SYSTEMVALUE) ≡
    value(Self.currentLabel, Self) :=
        case a.s-VALUEKIND of
        | kNow then mk-SDLTIME(now, TimeType)
        | kParent then Self.parent
        | kOffspring then Self.offspring
        | kSelf then Self.selfPid
        | kSender then Self.sender
        | kSignal then Self.signal
        | kSignallist then Self.inport.queue // SignalInst* for signals in input port where now > arrival time
        | kState then Self.stateAS1.s-Name.makeNameCharstring
        endcase
    Self.currentLabel := a.s-CONTINUELABEL
```

The *makeNameCharstring* function returns a character string value from the given name.

```
makeNameCharstring(n: Name): VALUE =def
    mk-SDLSTRING(< mk-SDLCHARACTER(n.s-TOKEN[i]) : i = 1..n.s-TOKEN.length > )
```

F3.2.1.4.24 Primitive Task

Explanation

The Task primitive is used for the evaluation of assignments. An action of type *TASK* is defined as a tuple consisting of a variable identifier, a value label and a continue label. The variable identifier becomes associated with a value within the state of the executing agent in the *NAMEDVALUE*-**set**.

Representation

An action of type *TASK* is defined as a tuple consisting of a variable name, a value label and a continue label.

$$TASK =_{def} \{ (id, v, b, cl) \in Identifier \times VALUELABEL \times CONTINUELABEL : id.refersto1 \in Variable\text{-}definition\}$$

Behaviour

The assignment is mainly realized by means of macro ASSIGN. Within the state of the executing agent the corresponding variable is set to the value, and this association is stored in the *NAMEDVALUE*-**set**. The macro ASSIGN is given the variable identifier, the value, the state, and current state identifier. ASSIGN uses the function *assign*, which uses the function *update* that uses *setValue* to modify the *NAMEDVALUE*-**set**.

If the sort of the variable is the *Sort* of a *Value-data-type-definition*, then if the *Aggregation-kind* of variable is **REF**, and the result of the *Expression* is not "undefined" (it does not access a variable that has no value associated), the value returned by the result of the *Expression* is associated with the identified variable. Otherwise, if the result of the *Expression* is "undefined", the variable is "undefined". See clause 12.3.3 Assignment, *Semantics* item (a) item (2) of [ITU-T Z.107].

If the *Variable-identifier* refers to a *Variable-definition* with *Aggregation-kind* **REF** and the *Expression* has an *Aggregation-kind* **PART**, the *Expression* shall be a *Variable-identifier* or an *Operation-application* that accesses a field of a data type defined in a <structure definition> or <choice definition>. See clause 12.3.3 Assignment, *Abstract grammar* of [ITU-T Z.107].

In the following an assignment where the (LHS) *Variable-identifier* refers to a *Variable-definition* with *Aggregation-kind* **PART** is called a value assignment, and an assignment where the (LHS) *Variable-identifier* refers to a *Variable-definition* with *Aggregation-kind* **REF** is called a reference assignment. If the *Expression* (RHS) of an assignment is a *Variable-access*, in the following the variable of the *Variable-access* is called the access variable, and is defined by a *Variable-definition* or a *Parameter*.

In a value assignment the expression (RHS) is evaluated, and the *NAMEDVALUE*-**set** item for the *Variable-identifier* has its *BOUNDVALUE* set to this value as a *VALUE*. If the RHS is an access variable, the evaluation takes place in the function *eval* (see clause F3.3.5 State access). If the expression (RHS) is an access variable with aggregation **PART**, the value is given by the *VALUE* of the *BOUNDVALUE* for access variable *NAMEDVALUE*-**set** item. If the expression (RHS) was an access variable with aggregation **REF**, the variable is dereferenced. An *Identifier* in the *BOUNDVALUE* of the *NAMEDVALUE*-**set** item for the access variable determines the item whose value is used. A *FIELDREF* in the *BOUNDVALUE* of the *NAMEDVALUE*-**set** item determines that the field given by the *Name* of the *FIELDREF* of the access variable given by the *Identifier* of the *FIELDREF* is used. If the *BOUNDVALUE* identifies an item with aggregation **REF**, dereferencing continues until an item with aggregation **PART** is found.

In a reference assignment the expression (RHS) has to be an access variable or an *Operation-application* that accesses a field of a data type defined in a <structure definition> or <choice definition>. In this case, reference assignment is recognised during the compilation of Assignment and EVALTASK is not compiled. Instead, EVALVAR, which is compiled to evaluate the variable access, detects the assignment context, includes code to update the LHS of the assignment of an access variable. If the expression (RHS) is an access variable, the *NAMEDVALUE*-**set** item for the *Variable-identifier* (LHS of the assignment) has its *BOUNDVALUE* set to a reference. If the expression (RHS) is an access variable with aggregation **PART**, the LHS *BOUNDVALUE* is set to the *Identifier* for the access variable. If the expression (RHS) is an access variable with aggregation **REF**, the *BOUNDVALUE* is set to the *BOUNDVALUE* (*Identifier* or *FIELDREF*) of the *NAMEDVALUE*-**set** item for the access variable. If the expression (RHS) is an operation application this is determined when the assignment is compiled and EVALREFFIELDASSIGN sets the LHS *BOUNDVALUE* to the *FIELDREF*) for the field.

```
EVALTASK(a:TASK) ≡
        ASSIGN(a.s-Identifier, value(a.s-VALUELABEL, Self), Self.stateAgent.state, Self.currentStateId)
        Self.currentLabel := a.s-CONTINUELABEL
```

Reference Sections

The definition of macro ASSIGN can be found in clause F3.2.1.3.1.

F3.2.1.4.25 Primitive TimerActive

Explanation

The TimerActive primitive is used for expressing a timer active expression. The primitive specifies the timer active check using the function *active*.

Representation

An action of type *TIMERACTIVE* is defined as tuple consisting of a timer, a sequence of value labels, and a continue label.

$$TIMERACTIVE =_{def} TIMER \times VALUELABEL^* \times CONTINUELABEL$$

Behaviour

Macro EVALTIMERACTIVE specifies the evaluation of a timer active expression.

```
EVALTIMERACTIVE(t:TIMERACTIVE) ≡
    let tmi = mk-TIMERINST(Self.selfPid, t.s-TIMER, values( t.s-VALUELABEL-seq, Self) ) in
        value(Self.currentLabel, Self) := mk-SDLBOOLEAN(active(tmi), BooleanType)
        Self.currentLabel := t.s-CONTINUELABEL
    endlet
```

Reference sections

The definition of function *active* can be found in clause F3.2.1.1.5.

F3.2.1.4.26 Primitive TimerRemaining

Explanation

The TimerRemaining primitive is used for expressing a timer remaining duration.

Representation

An action of type *TIMERREMAINING* is defined as a tuple consisting of a timer, a sequence of value labels, and a continue label.

TIMERREMAINING =def *TIMER* × *VALUELABEL** × *CONTINUELABEL*

Behaviour

Macro EVALTIMERREMAINING specifies the evaluation of a timer remaining duration.

```
EVALTIMERREMAINING(t:TIMERREMAINING) ≡
    let tmi = mk-TIMERINST(Self.selfPid, t.s-TIMER, values( t.s-VALUELABEL-seq, Self) ) in
        value(Self.currentLabel, Self) :=
            mk-SDLDURATION (if active(tmi) then tmi.arrival - now else 0.0 endif, DurationType)
        Self.currentLabel := t.s-CONTINUELABEL
    endlet
```

F3.2.1.4.27 Primitive TypeCheck

Explanation

The *TYPECHECK* primitive checks that the sort of a (dynamic) expression is compatible with a specified sort.

Representation

The domain *TYPECHECK* is defined as a Cartesian product of a *VALUELABEL* label (for the evaluated expression to be checked), an *Identifier* for the sort to check against and a *CONTINUELABEL*.

```
TYPECHECK =def { (vl, sort, cl) ∈ VALUELABEL × Identifier × CONTINUELABEL
    :   sort.refersto1 ∈ Value-data-type-definition }
```

Behaviour

The action *TYPECHECK* returns the result of the type check in the value associated with *Self.currentLabel*.

```
EVALTYPECHECK(a(vl, sort, cl) : TYPECHECK) ≡
    value(Self.currentLabel, Self) :=
        mk-SDLBOOLEAN(
            isSortCompatible(value(vl, Self).dynamicType, sort),
            BooleanType
        )
    Self.currentLabel := cl
```

F3.2.1.4.28 Primitive TypeCoercion

Explanation

The *TYPECOERCION* primitive changes the sort of a (dynamic) expression to the specified sort.

Representation

The domain *TYPECOERCION* is defined as a Cartesian product of a *VALUELABEL* label (for the evaluated expression), an *Identifier* for the sort and a *CONTINUELABEL*.

TYPECOERCION =$_{def}$ { (*vl*, *sort*, *cl*) ∈ *VALUELABEL* × *Identifier* × *CONTINUELABEL*
: *sort.refersto1* ∈ *Value-data-type-definition* }

Behaviour

The action *TYPECOERCION* returns either the value given as a value of the given sort or the invalidsort exception in the value associated with *Self.currentLabel*.

```
EVALTYPECOERCION(a(vl, sort, cl) : TYPECHECK) ≡
   value(Self.currentLabel, Self) := mk-VALUE(
      if isSuperSort(sort, value(vl, Self).dynamicType)
      then
         case value(vl, Self) of
         | PID(p, *)                 then mk-PID(p, sort)
         | SDLARRAY(p, q, r, *)      then mk-SDLARRAY(p, q, r, sort)
         | SDLBAG(x, *)              then mk-SDLBAG(x, sort)
         | SDLBIT(x, *)              then mk-SDLBIT(x, sort)
         | SDLBITSTRING(x, *)        then mk-SDLBITSTRING(x, sort)
         | SDLBOOLEAN(x,*)           then mk-SDLBOOLEAN(x, sort)
         | SDLCHARACTER(x, *)        then mk-SDLCHARACTER(x, sort)
         | SDLCHARSTRING(x, *)       then mk-SDLCHARSTRING(x, sort)
         | SDLCHOICE(x, *)           then mk-SDLCHOICE(x, sort)
         | SDLDURATION(x, *)         then mk-SDLDURATION(x, sort)
         | SDLINTEGER(x, *)          then mk-SDLINTEGER(x, sort)
         | SDLLITERAL(x,*)           then mk-SDLLITERAL(x, sort)
         | SDLOCTETSTRING(x, *)      then mk-SDLOCTETSTRING(x, sort)
         | SDLPOWERSET(x, *)         then mk-SDLPOWERSET(x, sort)
         | SDLREAL(n, d, *)          then mk-SDLREAL(n, d, sort)
         | SDLSTRING(x,*)            then mk-SDLSTRING(x, sort)
         | SDLSTRUCTURE(x,*)         then mk-SDLSTRUCTURE(x, sort)
         | SDLTIME(x, *)             then mk-SDLTIME(x, sort)
         | SDLVECTOR(x, *)           then mk-SDLVECTOR(x, sort)
         endcase
      else // not a supersort
         raise(InvalidSort)
      endif
   ) // mk-Value
   Self.currentLabel := cl
```

F3.2.1.4.29 Primitive Var

Explanation

The Var primitive models the evaluation of a variable access. It is used within the evaluation of expressions. An action of type *VAR* is a tuple consisting of a variable identifier and a continue label. The macro EVALVAR evaluates the given variable within the state of the executing agent and writes this value at the current label of this agent. In this way the result of the evaluation can be used in consecutive execution steps of this agent.

In some special contexts associated the use of **REF** (see **Behaviour** below), the *BOUNDVALUE* of the variable is needed, and this cannot be conveyed by the *VALUE* at the current label. For this reason, the action for these special contexts is applied in the macro EVALVAR and skipped in the compilation of the corresponding AS1 item.

Representation

The domain *VAR* is defined as a Cartesian product of *Identifier* where the identifier is constrained to refer to variable definitions, and the domain *CONTINUELABEL* of labels.

VAR =$_{\text{def}}$ { (*id*, *cl*) ∈ *Identifier* × *CONTINUELABEL*: *id.refersto1* ∈ *VARDEF* }

The domain *VARDEF* is the union of *Variable-definition* and *Parameter*, used where otherwise (*Variable-definition* ∪ *Parameter*) is needed.

VARDEF =$_{\text{def}}$ *Variable-definition* ∪ *Parameter*

Behaviour

The variable access receives special treatment if the context is:

- the right hand side of an *Assignment* where the left hand side is a variable with *Aggregation-kind* **REF**; or
- the actual parameter for a formal parameter with *Parameter-aggregation* that is *Aggregation-kind* **REF**; or
- one side of an equality expression where the accessed variable has *Aggregation-kind* **REF** and the other side of the equality expression also has *Aggregation-kind* **REF**; or
- the expression for a *Value-return-node* in a procedure graph where the procedure has a *Result-aggregation-kind* **REF** and the *Value-returning-call-node* of the calling procedure
  a) is assigned to a variable with *Aggregation-kind* **REF**; or
  b) is assigned to a formal parameter with *Parameter-aggregation* that is with *Aggregation-kind* **REF**; or
  c) is (recursively) the expression for a *Value-return-node* in a procedure graph (where the procedure has a *Result-aggregation-kind* **REF)** until the *Value-returning-call-node* is of kind (a) or (b) above.

In the contexts listed above, variable access determines these special cases and executes code that acts together with the handling of the context containing the variable access. For example, the compilation for an *Assignment* where the left hand side is a variable with *Aggregation-kind* **REF**, simply compiles the right hand side expression for assignment with no further action, and binding of the left hand side variable is handled in EVALVAR.

The controlled function *resultRefVarId* (see clause F3.2.1.2.1, State machine) is set to identifier of the variable or parameter to reference the accessed variable by the handling of *Value-returning-call-node* for a procedure that has a *Result-aggregation-kind* **REF** and satisfies case (a) or (b) above. Otherwise *resultRefVarId* is undefined.

In contexts other than the cases above, if the value of a variable in the current state of the executing agent is *undefined*, the *UndefinedVariable* exception is raised. Otherwise the value of a variable in the current state of the executing agent is determined by the function *eval* and is written at *Self.currentLabel* (where to avoid conflicts with other agents, the function *value* takes a further argument of type *AGENT*, which identifies the owner of the value).

The label which determines the next rule to be fired is set to the given continue label.

EVALVAR(*a*:*VAR*) ≡
  let *parent* = *a*.**s**-*Identifier*.*parentAS1* **in**
  let *namedValue* = *getValueSet*(*Self.stateAgent.state*.**s**-*NAMEDVALUE*-**set**, *Self.currentStateId*, *a*.**s**-*Identifier*)
  in // singleton set for namedValue binding of id in the current state
  if (*parent* ∈ *Assignment* // is in assignment
    ∧ *parent*.**s**-*Identifier*.*refersto1*.**s**-*Aggregation-kind* = **REF**) // and LHS of assign REF

```
then // assignment to variable with REF
    let rhsdef = a.s-Identifier.refersto1 in // Variable-definition or Parameter
    update(parent.s-Identifier,
        if      (rhsdef ∈ Variable-definition ∧ rhsdef.s-Aggregation-kind = PART)
            ∨   (rhsdef ∈ Parameter ∧ rhsdef.s-Parameter-aggregation.s-Aggregation-kind = PART)
        then mk-BOUNDVALUE(a.s-Identifier) //id of vari/param on RHS of assign –boundvalue of LHS
        elseif namedval ≠ ∅ then // there is a named value, but the bound value could be undefined
            namedval.take.s-BOUNDVALUE // should be one element in namedval. Result can be
            // undefined – named value, but no Boundvalue
            // Identifier – if reference to another variable
            // FieldRef – if reference to a field
            // null literal signature– if variable was set to null
        else // there is no named value in state for given id
            undefined // for boundvalue of LHS
        endif, // BoundValue
        Self.stateAgent.state,
        Self.currentStateId
    ) // update binding – that is, do assignment
    endlet // rhsdef
    Self.currentLabel := a.s-CONTINUELABEL
elseif // is an actual parameter for a formal parameter with REF
    (       parent ∈ Actual-parameters // parent is actual parameter list
        ∧   parent.parentAS1 ∈ (Call-node ∪ Value-returning-call-node) // in call node
        ∧   (∃! i ∈ 1..parent.length : // exactly one actual parameter parent[i] such that
                parent[i] = a.s-Identifier // parameter is the given id
            ∧   refersto1(parent.parentAS1.s-Identifier  //procedure id refers to proc definition
                ).s-PROCPARAM-seq[i].s-Parameter-aggregation.s-Aggregation-kind = REF
                // corresponding formal parameter has REF aggregation
            ) // exists meeting conditions
    )
then // assign actual parameter to a formal parameter with REF – similar to assign above
    let index = { i ∈ 1..parent.length : parent[i] = a.s-Identifier }.take in // index parameter list item
    let actpar = a.s-Identifier.refersto1 in // Variable-definition or Parameter as actual param
    update( // the first item is the identifier of the formal parameter of the procedure
        parent.parentAS1.s-Identifier.refersto1.s-PROCPARAM-seq[index].s-Parameter.identifier1,
        if      (actpar ∈ Variable-definition ∧ actpar.s-Aggregation-kind = PART)
            ∨   (actpar ∈ Parameter ∧ actpar.s-Parameter-aggregation.s-Aggregation-kind = PART)
        then mk-BOUNDVALUE(a.s-Identifier) // id of varparam for actual param – boundvalue of formal
        elseif namedval ≠ ∅ then namedval.take.s-BOUNDVALUE
        else undefined
        endif // BoundValue
        Self.stateAgent.state,
        Self.currentStateId
    ) // update binding – that is, do assignment
    endlet // actpar
    endlet // index
    Self.currentLabel := a.s-CONTINUELABEL
elseif // accessed variable has aggregation kind REF on one side of equality expr with other side REF
    let firstOpDef = parent.s1-EXPRESSION.refersto1 in
    let secondOpDef = parent.s2-EXPRESSION.refersto1 in
        parent ∈ (Positive-equality-expression ∪ Negative-equality-expression)
    ∧   (       ( firstOpDef ∈ Variable-definition ∧ firstOpDef.s-Aggregation-kind = REF)
            ∨   (firstOpDef ∈ Parameter ∧ firstOpDef.s-Parameter-aggregation.s-Aggregation-kind
                    = REF)
        ) // first operand REF
    ∧   (       ( secondOpDef ∈ Variable-definition ∧ secondOpDef.s-Aggregation-kind = REF)
            ∨   (secondOpDef ∈ Parameter ∧ secondOpDef.s-Parameter-aggregation.s-Aggregation-kind
                    = REF)
        ) // second operand REF
    endlet // secondOperandDef
    endlet // firstOperandDef
then // accessed variable has aggregation kind REF on one side of equality expr with other side REF
```

```
        let boundValue1 =
            getValueSet(Self.stateAgent.state.s-NAMEDVALUE-set,
                Self.currentStateId,
                parent.s1-EXPRESSION.refersto1
            ).take.s-BOUNDVALUE
        in
        let boundValue2 =
            getValueSet(Self.stateAgent.state.s-NAMEDVALUE-set,
                Self.currentStateId,
                parent.s2-EXPRESSION.refersto1
            ).take.s-BOUNDVALUE
        in
        value(Self.currentLabel, Self) :=
            mk-SDLBOOLEAN(
                if      (boundValue1 = undefined ∧ boundValue2 = undefined) // both undefined
                    ∨   (boundValue1.isNullValue ∧ boundValue2.isNullValue) // both null
                    ∨   (boundValue1 ∈ Identifier ∧ boundValue1 = boundValue2)
                    ∨   (boundValue1 ∈ FIELDREF ∧ boundValue1 = boundValue2)
                then if parent ∈ Positive-equality-expression then true else false endif
                else if parent ∈ Positive-equality-expression then false else true endif
                endif,
                BooleanType
            )
        endlet // boundValue2
        endlet // boundValue1
        Self.currentLabel := a.s-CONTINUELABEL
    elseif // value return for procedure with result aggregation REF and call assigns to REF item
        (   parent ∈ Value-return-node
        ∧   Self.currentProcedureStateNode.procedureAS1.s-Result.s-Result-aggregation = REF
        ∧   Self.callingProcedureNode.resultRefVarId ≠ undefined
        )
    then // assign value to the result with REF
        let valret = a.s-Identifier.refersto1 in // Variable-definition or Parameter as returned value
        update(Self.callingProcedureNode.resultRefVarId, // id of variable to be bound to result
        mk-BOUNDVALUE(
            if      (valret ∈ Variable-definition ∧ valret.s-Aggregation-kind = PART)
                ∨   (valret ∈ Parameter ∧ valret.s-Parameter-aggregation.s-Aggregation-kind = PART)
            then mk-BOUNDVALUE(a.s-Identifier) // id of var/param for actual param – for boundvalue of result
            elseif namedval ≠ ∅ then namedval.take.s-BOUNDVALUE
            else undefined
            endif, // BoundValue
            Self.stateAgent.state,
            Self.currentStateId
        ) // update binding – that is, do assignment
        endlet // valret
        Self.currentLabel := a.s-CONTINUELABEL
    elseif // none of the special cases apply – check variable not undefined
        eval(a.s-Identifier, Self.stateAgent.state, Self.currentStateId) = undefined
    then
        raise(UndefinedVariable)
    else // normal case – variable with defined value - evaluated
        value(Self.currentLabel, Self) := eval(a.s-Identifier, Self.stateAgent.state, Self.currentStateId)
        Self.currentLabel := a.s-CONTINUELABEL
    endif
    endlet // namedValue
    endlet // parent
```

Auxiliary functions

For the definition of function *value* refer to clause F3.2.1.4.1. The definition of function *eval* can be found in clause F3.2.1.3.1. Function *currentLabel* is defined in clause F3.2.1.2.3.

Function *isNullValue* is true if (and only if) the given *BOUNDVALUE* is a *Null-literal-signature*.

isNullValue(*bv*: *BOUNDVALUE*): *BOOLEAN* =$_{\text{def}}$
bv ∈ *Literal-signature* ∧ *bv*.**s**-*Name* = "null" ∧ *bv*.**s**-*Result*.**s**-*Result-aggregation* = **REF**

F3.2.1.4.30 Primitive ZDecoding and ZEncoding (including ZEncodingPar and ZEncodingVal)

Explanation

SDL-2010 allows values to be encoded according to rules for ASN.1 specified in [ITU-T X.690] or [ITU-T X.691] or [ITU-T X.693] or [ITU-T X.696] or according to user specified encode and decode procedures. For more details see clause 10.7 of [ITU-T Z.104]. The data type for the rules identifiers is part of the package Predefined that has to be modified if there are any user specified rules identifiers, therefore it is assumed that all encode and decode procedures are also defined in the package Predefined.

The *ZDECODING* primitive models the decoding of an encoded expression in a *Decoding-expression* to produce a value of the choice type for the agent that can hold any of the signals of the agent interface.

The *ZENCODING* primitive models the encoding of a signal in an *Encoding-expression* (given as a *Signal-identifier* and corresponding parameters as an *Expression* sequence), to a result value (*Charstring* ∪ *Bitstring* ∪ *Octetstring*) of the encode procedure identified by the *Rules-identifier* of the *Encoding-path*.

Representation

The domain *ZDECODING* is defined as a Cartesian product of a *VALUELABEL* for an *Expression* to be decoded, the *Encoding-path* for that identifies the decoding to be used, and a *CONTINUELABEL*.

ZDECODING =$_{\text{def}}$ { (*ex*, *ep*, *cl*) ∈ *VALUELABEL* × *Encoding-path* × *VALUELABEL* × *CONTINUELABEL*
: *value*(*ex*, *Self*) ∈ *encodeSort*(*ep*) } // data type for value of ex

The action *ZENCODING* is defined as a Cartesian product of a *VALUELABEL* for the choice value to be encoded constructed by *ZENCODINGPAR* and *ZENCODINGVAL* (from the signal *Identifier*, *Expression* list and *Encoding-path* that identifies the encoding to be used), the *Encoding-path* and a *CONTINUELABEL*.

ZENCODING =$_{\text{def}}$ *VALUELABEL* × *Encoding-path* × *CONTINUELABEL*

The action *ZENCODINGPAR* is defined as a Cartesian product of a signal *Identifier*, a *VALUELABEL* sequence for the expressions carried by the signal, a *VALUELABEL* for the returned value, and a *CONTINUELABEL*.

ZENCODINGPAR =$_{\text{def}}$ { (*sigid*, *exprs*, *rl*, *cl*) ∈
Identifier × *VALUELABEL** × *Encoding-path* × *VALUELABEL* × *CONTINUELABEL*
: *sigid.refersto1* ∈ (*Signal-definition* ∪ *Timer-definition*) }

The action *ZENCODINGVAL* is defined as a Cartesian product of a signal *Identifier*, a *VALUELABEL* sequence for the expressions carried by the signal, a *VALUELABEL* for the returned value, and a *CONTINUELABEL*.

ZENCODINGVAL =$_{\text{def}}$ { (*sigid*, *exprs*, *rl*, *cl*) ∈
Identifier × *VALUELABEL** × *Encoding-path* × *VALUELABEL* × *CONTINUELABEL*
: *sigid.refersto1* ∈ (*Signal-definition* ∪ *Timer-definition*) }

Behaviour

The action *ZDECODING* calls the decode procedure for the rules identifier with *ex* as a parameter.

```
EVALZDECODING(a(ex, ep, cl) : ZDECODING) ≡
    CREATEPROCEDURE(
        ep.encodingRules.s2-Identifier, // decode procedure Identifier
        < value(ex, Self)  >, // value expression to decode– as a sequence
        Self.currentlabel, // value label for the returned value
        cl // continue label
    )
```

The action *ZENCODING* calls the encode procedure identified by the *Encoding-path* (*ep*) for the rules identifier with *ex* as a parameter.

```
EVALZENCODING(a(ex, ep, cl) : ZDECODING) ≡
    CREATEPROCEDURE(
        ep.encodingRules.s1-Identifier, // encode procedure Identifier
        < value(ex, Self) >, // value expression to decode from ZEncodingVal – as a sequence
        Self.currentlabel, // value label for the returned value
        cl // continue label
    )
```

The action *ZENCODINGPAR* evaluates the structure value returned as a value label and provided as a parameter to *ZENCODINGVAL*.

```
EVALZENCODINGPAR(a(sigid, exprs, rl, cl): ZENCODINGPAR) ≡
    if sigid.refersto1.s-Signal-parameter-seq = empty // signal parameterless
    then
        value(rl, Self) := nullNULL // unique value null of the predefined NULL data type
        Self.currentLabel := cl
    else EVALOPERATIONAPPLICATION(mk-OPERATIONAPPLICATION(
            mk-OPLITSIGNATURE(take({os ∈ Operation-signature :
                os.s-Name = mk-Name("Make")
                ∧ os.s-Identifier-seq =
                    < sigid.refersto1.s-Signal-parameter-seq[i].s-Identifier // ith parameter sort
                    : i ∈ 1..sigid.refersto1.s-Signal-parameter–seq.length
                    > // list of sorts for Make from signal definition signal parameters
                ∧ os.s1-Identifier = sigid.asSignal1}),
            values(exprs, Self), // list of signal parameters – Value list
            rl, // value label for the returned structure value from OperationApplication
            cl // label for next action after EvalOperationApplication
        )) // EvalOperationApplication – leaves value in value(rl)
    endif // null or parameters
```

The action *ZENCODINGVAL* evaluates the choice value returned as a value label and provided as a parameter to *ZENCODING*.

```
EVALZENCODINGVAL(a(sigid, ex, rl, cl): ZENCODINGVAL) ≡
    EVALOPERATIONAPPLICATION(mk-OPERATIONAPPLICATION(
        mk-OPLITSIGNATURE(take({os ∈ Operation-signature :
            os.s-Name = sigid.s-Name
            ∧ os.s-Identifier-seq =
                < sigid.asSignal1 > // struct sort for signal from signal identifier
            ∧ os.s1-Identifier = sigid.asSignal1}),
        < value(ex,Self) >, // the structure value from ZEncodingVal
        rl, // value label for the returned choice value from OperationApplication
        cl // label for next action after EvalOperationApplication
    )) // EvalOperationApplication – leaves value in value(rl)
```

Auxiliary functions

Function *encodeSort* gives the result sort for the encode procedure corresponding to an *Encoding-path*.

```
encodeSort(ep: Encoding-path):Identifier =def
```

```
case riToken(ep) of // select result type based on Rules-identifier name
|   "text"      then CharstringType
|   "BER"       then OctetstringType
|   "CER"       then OctetstringType
|   "DER"       then OctetstringType
|   "APER"      then OctetstringType
|   "UPER"      then BitstringType
|   "CAPER"     then OctetstringType
|   "CUPER"     then BitstringType
|   "BXER"      then CharstringType
|   "CXER"      then CharstringType
|   "EXER"      then CharstringType
|   "OER"       then OctetstringType
otherwise
    ep.encodingRules.s1-Identifier.refersto1.s-Result.s-Identifier // Result of encode
endcase
```

Function *riToken* gives the *TOKEN* for the rules identifier corresponding to an *Encoding-path*.

```
riToken(ep: Encoding-path): TOKEN =def
    ep.encodingRules.s-Identifier.s-Name.s-TOKEN // Token for name of Rules-identifier of Encoding rules
```

Function *encodingRules* gives the *TOKEN* for the rules identifier corresponding to an *Encoding-path*.

```
encodingRules(ep: Encoding-path): Encoding-rules =def
    if ep.s-implicit ∈ Identifier // Encoding-path is Gate-identifier
    then ep.s-implicit.refersto1.s-Encoding-rules // Encoding-rules in Gate-definition
    else ep.s-implicit.s-Encoding-rules // Encoding-rules from Encoding path
    endif
```

F3.2.1.4.31 Primitive Raise (SDL-2000 feature)

Explanation

In SDL-2000 the Raise primitive is used for expressing the raising of exceptions. In SDL-2010, exceptions cannot be explicitly raised, so there is no need for the *RAISE* primitive, the EVALRAISE or RAISEEXCEPTION macros that were defined in the formal dynamic semantics for SDL-2000. Predefined exceptions still occur for certain well-defined runs as indicated by the use of the *raise* function with the exception identifier as a parameter. When this occurs the further behaviour of the system is not defined by SDL-2010.

Reference sections

The *EXCEPTION* domain is defined in clause F3.2.1.1.6. The *raise* function is defined in clause F3.3.1.1.

F3.2.1.5 Undefined behaviour

Undefined behaviour is represented by the following program:

```
UNDEFINEDBEHAVIOUR ≡
    Self.program := UNDEFINED-BEHAVIOUR-PROGRAM

UNDEFINED-BEHAVIOUR-PROGRAM:
// the contents of this program is not defined
```

The content of the program UNDEFINED-BEHAVIOUR-PROGRAM is not specified. Whenever the further behaviour of the system is undefined, the current agent is switched to this program.

This local undefinedness condition is in fact global as the program UNDEFINED-BEHAVIOUR-PROGRAM could involve setting *program* for all agents.

F3.2.2 Compilation function

The compilation is formalized in terms of the following two compilation functions, one for transition behaviour and one for expression behaviour, that form the interface between the compilation and the dynamic semantics. For all the behaviour parts that involve transitions, the corresponding runtime representation of the transitions is generated.

compile: *DEFINITIONAS1* → *BEHAVIOUR* // See F3.2.2.1 States and triggers

compileExpr: *DEFINITIONAS1* × *LABEL* → *BEHAVIOUR* // See F3.2.2.3 Actions

The computed value of an expression *e* is always stored at *value*(*uniqueLabel*(*e*, 1), *Self*), where *e* is an expression that is a *DEFINITIONAS1* item.

The two compilation functions are gradually introduced by defining a series of compilation patterns and the corresponding results in clauses F3.2.2.1 and F3.2.2.2 for *compile* and clause F3.2.2.3 for *compileExpr*; each individual pattern is uniquely associated with a certain type of node in the AST to be compiled. Afterwards, the function *startLabel* is defined also with a series of patterns in clause F3.2.2.4. The functions *compile* and *compileExpr* convert the AST of the SDL-2010 state machine description into an ASM representation.

A special *labelling* of graph nodes is used to model specific control-flow information. Intuitively, node labels relate individual operations of an SDL-2010 agent to transition rules in the resulting SAM model. The effect of state transitions of SDL-2010 agents is then modelled by firing the related transition rules in an analogous order.

Labels are abstractly represented by a static domain *LABEL*.

static domain *LABEL*

To start with the compilation, a function is needed to find unique labels for each syntactic construct. The second argument is introduced to allow for more than one such label within the same SDL-2010 pattern.

monitored *uniqueLabel*: *DEFINITIONAS1* × *NAT* → *LABEL*

For this function, it holds that

constraint ∀ *d*1, *d*2 ∈ *DEFINITIONAS1*: ∀ *i*1, *i*2 ∈ *NAT*:
uniqueLabel(*d*1, *i*1) = *uniqueLabel*(*d*2, *i*2) ⇔ (*d*1=*d*2 ∧ *i*1=*i*2)

Finally, to formalize the compilation, an auxiliary function is needed to generate a sequence out of a set. This function is used when the sequence of events has to be computed but does not really matter. See for instance *Decision-node* and *Range-condition*.

```
setToSeq(s: X-set): X* =def
    if s = ∅ then empty else
        let el = s.take in
            < el > ⌢ setToSeq(s \ { el })
        endlet
    endif
```

F3.2.2.1 States and triggers

The following parts in this clause and parts in clause F3.2.2.2 Terminators are considered to form the definition of the function *compile* if put together with the following header and followed by an **otherwise** and **endcase** (see the end of F3.2.2.2). The contents of the case expression are all the compilation cases as given below.

```
compile(a: DEFINITIONAS1): BEHAVIOUR =def
    case a of
```

All the contents of this function are given as patterns and what the result of the function is for these patterns. The default case when no pattern is matching is the collected set of all the results of all children nodes.

The handling of inheritance is done in the dynamic part. What you find below is the compilation of the plain behaviour descriptions.

The definition of the compilation function is done using a series of auxiliary derived functions.

```
| v=Variable-definition( *, *, *, init) then
    if init ≠ undefined then
        compileExpr(init, uniqueLabel(v,1))
        ∪ {mk-PRIMITIVE(uniqueLabel(v,1), mk-TASK(v.identifier1, uniqueLabel(init,1), undefined)) }
    else ∅
    endif

| State-transition-graph( start, states, freeActions) then
        compile(start)
        ∪ ⋃{ compile(s) | s ∈ states }
        ∪ ⋃{ compile(f) | f ∈ freeActions }

| Procedure-graph( start, states, freeActions) then
        compile(start)
        ∪ ⋃{ compile(s) | s ∈ states }
        ∪ ⋃{ compile(f) | f ∈ freeActions }

| State-start-node(transition) then compile(transition)

| Procedure-start-node(transition) then compile(transition)

| Named-start-node(*, trans) then compile(trans)

| State-node(*, *, inputs, exits, statetimer) then
            ⋃{ compile(i) | i ∈ inputs }
        ∪ if exits.Spontaneous-transition-set ≠ undefined ∨ exits.Continuous-signal-set ≠ undefined
            then ⋃{ compile(s) | s ∈ exits.s-Spontaneous-transition-set }
                ∪ ⋃{ compile(c) | c ∈ exits.s-Continuous-signal-set }
            else ⋃ { compile(e) | e ∈ exits.s-Connect-node-set }
            endif
        ∪ compile(statetimer)

| i = Input-node(*,sigid, *, provided, inchoice, vars, transition) then
            if provided = undefined then ∅ else compileExpr(provided, undefined) endif
            // The function getStateTransitions of clause F3.2.3.1.11 State node creation sets
            // the first Label of a SemTransition item to the start label of the provided expression,
            // evaluated at execution time by SelInputEvaluationPhase clause F3.2.3.2.10 Input selection.
        ∪ if inchoice = undefined then ∅
            else
                    { mk-PRIMITIVE(uniqueLabel(i,2), // evaluate signal parameters for in choice
                      mk-ASSIGNSIGNALPAR(sigid, uniqueLabel(i,2), uniqueLabel(i,3))) }
                ∪ { mk-PRIMITIVE(uniqueLabel(i,3), // value to assign to choice variable
                      mk-ASSIGNSIGNALVAL(inchoice, sigid, uniqueLabel(i,2), uniqueLabel(i,4))) }
                ∪ { mk-PRIMITIVE(uniqueLabel(i,4),
                      mk-ASSIGNSIGNAL(inchoice, uniqueLabel(i,3), uniqueLabel(i,1))) } assign to inchoice
            endif //
        ∪ if vars = undefined ∨ vars.length = 0
            then { mk-PRIMITIVE(uniqueLabel(i,1), mk-SKIP(transition.startLabel) )}
            else // assign to vars
                { mk-PRIMITIVE(uniqueLabel(i,idx),
                    if vars[idx] ≠ undefined
                    then mk-ASSIGNPARAMETER(vars[idx],
                        idx,
                        if idx < vars.length then uniqueLabel(i,idx+1) else transition.startLabel endif
                        ) //AssignParameter
```

```
                    else mk-SKIP( if idx < vars.length then uniqueLabel(i,idx+1) else transition.startLabel)
                    endif
                ) // Primitive
            | idx ∈ 1..vars.length  }
        endif  // vars undefined or vars length =0
    ∪ compile(transition)

| Spontaneous-transition(provided, transition) then
        if provided = undefined then ∅ else compileExpr(provided, undefined) endif
        // The function getStateTransitions of clause F3.2.3.1.11 State node creation sets
        // the first Label of a SemTransition item to the start label of the provided expression,
        // evaluated at execution time by SelInputEvaluationPhase clause F3.2.3.2.10 Input selection.
    ∪ compile(transition)

| Continuous-signal(ce, *, transition) then
        compileExpr(ce, undefined)
        // The function getStateTransitions of clause F3.2.3.1.11 State node creation sets
        // the first Label of a SemTransition item to the start label of the continuous expression (ce),
        // evaluated at execution time by SelInputEvaluationPhase clause F3.2.3.2.10 Input selection.
    ∪ compile(transition)

| Connect-node(*, transition) then compile(transition)

| s = State-timer(te, stimerId, exprList, transition) then
    if exprList = empty then ∅
    else ⋃{  compileExpr(exprList[i], (if i < exprList.length then exprList[i+1] else te endif).startLabel)
        | i ∈ 1..exprList.length }
    endif
    ∪ compileExpr(te, uniqueLabel(s,1))
    ∪ {  mk-PRIMITIVE(uniqueLabel(s,1),
            mk-TIMERACTIVE(stimerId, <uniqueLabel(e,1) | e in exprList >, uniqueLabel(s,2))
        ) // Primitive
      }
    ∪ {  mk-PRIMITIVE(uniqueLabel(s,2),
            mk-DECISION(
                uniqueLabel(s,1), // value label for timer active
                { mk-ANSWER( undefined, // undefined is treated as SDL false in the Decision action
                    uniqueLabel(s,3)
                ) } // mk-Answer, set of answers
            ) // mk-Decision
        ) // Primitive – if the timer active is false, goto the SET
      }
    ∪ {  mk-PRIMITIVE(uniqueLabel(s,3),
            mk-SET(uniqueLabel(te,1),
                stimerId,
                <uniqueLabel(e,1) | e in exprList >,
                undefined) // Set
        ) // Primitive
      }
    ∪ compile(transition) // selected by st.startlabel, where st ∈ State-timer

| Free-action(*, transition) then compile(transition)

| t=Transition(nodes, endnode) then
  if    t.parentAS1.parentAS1 ∈ State-node
    ∧ t.parentAS1.parentAS1.s-Name ≠ undefined // Name is State-name
  then // entering a transition from a state node
    {  mk-PRIMITIVE(uniqueLabel(t,1),
          mk-LEAVESTATENODE(t.parentAS1.parentAS1.s-Name, // Name is State-name
              startLabel(if nodes = empty then endnode else nodes.head endif) //startLabel
          ) // LeaveStateNode
      ) // Primitive
    }
  else ∅
  endif
```

```
    ∪ compileNodes(nodes, endnode)
    ∪ compile(endnode) // Terminator or Decision-node
where

    compileNodes (nodes: Graph-node*, endnode: Terminator ∪ Decision-node): BEHAVIOUR =def
        if nodes = empty then ∅
        else
                ⋃{ compileExpr(nodes[i], nodes[i+1].startLabel) | i ∈ 1..nodes.length - 1 }
            ∪ compileExpr(nodes.last, endnode.startLabel)
        endif
endwhere
```

F3.2.2.2 Terminators

```
| Decision-node(body) then compile(body) // body is Decision-body or Any-decision

| Terminator(terminator) then compile(terminator)

| n=Named-nextstate(stateName, undefined) then
        {mk-PRIMITIVE(uniqueLabel(n,1),
            mk-ENTERSTATENODE(stateName, undefined, empty)) }

| n=Named-nextstate(stateName, Nextstate-parameters(Actual-parameters(exprList), entry)) then
        if exprList = empty then ∅
        else    ⋃{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }
            ∪ compileExpr(exprList.last, uniqueLabel(n,1))
        endif
        ∪ {   mk-PRIMITIVE(uniqueLabel(n,1),
                  mk-ENTERSTATENODE(stateName, entry, <uniqueLabel(e,1) | e in exprList >)
              ) // Primitive
          }

| n=Dash-nextstate(HISTORY) then
        {mk-PRIMITIVE(uniqueLabel(n,1), mk-ENTERSTATENODE(HISTORY, undefined, empty)) }
```

NOTE: Only the **HISTORY** case is handled, because the state does not change otherwise.

```
| s=Stop-node() then
        {mk-PRIMITIVE(uniqueLabel(s,1), mk-STOP()) }

| a=Action-return-node() then
        {mk-PRIMITIVE(
            uniqueLabel(a,1),
            case parentAS1ofKind(a, Procedure-graph ∪ Composite-state-graph) of
            | Procedure-graph then
                mk-RETURN(uniqueLabel(undefined,1)) // return for a procedure call
            | Composite-state-graph then
                if a.owner ∈ AGENT // determine that Action-return-node in composite state of state machine
                then mk-STOP() // interpret a Stop-node
                else // not in composite state of state machine
                    mk-RETURN(DEFAULT) // State exit point for Connect-node without a name
                endif // state machine or other composite state
            endcase // procedure or composite state graph
        ) } // Primitive
```

Clause 11.12.2.4 of [ITU-T Z.101] *Abstract grammar* states that an *Action-return-node* shall only be contained in the *Procedure-graph* of a *Procedure-definition* without *Result* or a *Composite-state-graph*. From clause 11.12.2.4 of [ITU-T Z.101] *Semantics,* when an *Action-return-node* of a procedure is interpreted, the *Procedure-graph* is completed and the procedure instance ceases to exist, then interpretation of the calling context continues. Also from clause 11.12.2.4 of [ITU-T Z.101] *Semantics,* an *Action-return-node* in the composite state that is the state machine of an agent is interpreted as a *Stop-node*, otherwise an *Action-return-node* in a composite state results in activation of a *Connect-node* and interpretation continues at the *Connect-node* without a name (**DEFAULT**).

```
| v=Value-return-node(expr) then
        if      Self.currentProcedureStateNode.procedureAS1.s-Result.s-Result-aggregation = REF
            ∧   expr ∈ Variable-access
        then // special case of return for result with REF handled in variable access action as above
            compileExpr(expr, next) // expr is a Variable-access
        else // normal case
            compileExpr(expr, uniqueLabel(v,1))
            ∪
            {mk-PRIMITIVE(uniqueLabel(v,1), mk-RETURN(uniqueLabel(expr,1))) }
        endif

| n=Named-return-node(name) then
    {mk-PRIMITIVE(uniqueLabel(n,1), mk-RETURN(name)) }

| j= Join-node(connector) then
    {mk-PRIMITIVE(uniqueLabel(j,1), mk-SKIP(connector)) }

| b= Break-node(connector) then
    {mk-PRIMITIVE(uniqueLabel(b,1), mk-BREAK(connector)) }

| c= Continue-node(connector) then
    {mk-PRIMITIVE(uniqueLabel(c,1), mk-CONTINUE(connector)) }

| d=Decision-body(question, answerset, Else-answer(elseanswer)) then
     let aseq = answerset.setToSeq in
        compileExpr(question, aseq[1].startLabel)
    ∪  { compileExpr(aseq[idx].s-VALUELABEL,
            if idx=aseq.length then uniqueLabel(d, 1) else aseq[idx+1].startLabel endif)
        | idx ∈ toSet(1..aseq.length) }
    ∪  { mk-PRIMITIVE(uniqueLabel(d, 1),
            mk-DECISION(uniqueLabel(question, 1),
                { mk-ANSWER(uniqueLabel(ans.s-VALUELABEL, 1), ans.s-Transition.startLabel)
                | ans ∈ answerset },
                if elseanswer=undefined then undefined else elseanswer.s-Transition endif)) }
    ∪  ⋃{ compile(ans.s-Transition) | ans ∈ answerset }
    ∪  compile(elseanswer.s-Transition)
    endlet // aseq

| d=Any-decision(tset) then
        {mk-PRIMITIVE(uniqueLabel(d,1), mk-DECISIONANYVALUE(uniqueLabel(d,1), tset,
uniqueLabel(d,2)))}
    ∪  {mk-PRIMITIVE(
            uniqueLabel(d,2),
            mk-DECISION(
                uniqueLabel(d,1), // random value from DecisionAnyValue
                {   mk-Decision-answer(
                        mk-Range-condition({Closed-range(sel, sel)}), //sel (selector for answer) is i
                        tset.setToSeq[i]
                    ) // Decision-answer = i
                    :       i ∈ 1.. | tset |
                    ∧   sel =
                        take({mk-Literal(litid)
                        :   litid ∈ Identifier
                            ∧ litid.refersto1 ∈ IntegerType.refersto1.s-Literal-signature-set
                        :   ∧ litid.refersto1.s-NAT = (2 * i) //
                        }), Literal-set, take to get Literal – should only be one Literal in the set
                } // Decision-answer set
            ) // mk-Decision
        )} mk-Primitive, set for primitive
```

This concludes the definition of the *compile* function.

```
otherwise ∅ // empty set if no matching pattern
endcase // end of the compile function definition
```

F3.2.2.3 Actions

The following compilation parts define the function *compileExpr* with the following header and followed by an **otherwise** and **endcase** as given at the end of this clause.

compileExpr(*a*: *DEFINITIONAS1*, *next*: *LABEL*): *BEHAVIOUR* =$_{\text{def}}$
case *a* **of**

Most applications of *compileExpr* are concerned with the compilation of *Expression* elements from AS1, though *compileExpr* is also applied to other elements such as the action of a *Graph-node*. However, the AS1 non-terminal *Expression* is not a constructor (it is defined using "=" rather than "::"), and for this reason a domain *EXPRESSION* is defined, each member of which is an AS1 element that is an expression.

EXPRESSION =$_{\text{def}}$ *Active-agents-expression*
∪ *Agent-instance-pid-value*
∪ *Any-expression*
∪ *Conditional-expression*
∪ *Decoding-expression*
∪ *Encoding-expression*
∪ *Literal*
∪ *Negative-equality-expression*
∪ *Now-expression*
∪ *Offspring-expression*
∪ *Operation-application*
∪ *Parent-expression*
∪ *Positive-equality-expression*
∪ *Range-check-expression*
∪ *Self-expression*
∪ *Sender-expression*
∪ *Signal-expression*
∪ *Signallist-expression*
∪ *State-expression*
∪ *Timer-active-expression*
∪ *Timer-remaining-duration*
∪ *Type-check-expression*
∪ *Type-coercion*
∪ *Value-returning-call-node*
∪ *Variable-access*

The function *isConstant1* from F2 tests whether an expression is a constant expression.

All the contents of the *compileExpr* function are given as patterns and the result of the function for these patterns. The default result when no pattern is matching is the empty set. All the patterns given below may use the variable *next* referring to the next label to be handled.

The patterns are sorted in alphabetical order of the abstract syntax rule names. Where more than one pattern uses the same abstract rule, the order chosen depends on which alternatives are considered most common.

```
| a=Active-agents-expression(ag) then // ag is an agent Identifier or THIS
        {   mk-PRIMITIVE(uniqueLabel(a,1),
              mkSDLInteger(
                 |{ sa ∈ SDLAGENT: sa.agentAS1 =
                      if ag ∈ Identifier // agent Identifier (rather than THIS)
                      then ag.refersto1
                      else Self.agentAS1 // THIS (rather than agent Identifier)
                 }|, // number of instances matching the agent definition
                 IntegerType
              ) // SDLInteger
           ) //Primitive
```

```
        }
| aipd=Agent-instance-pid-value(agentInstanceList) then
    if agentInstanceList = empty then ∅
    else    // compile number expressions in agentInstanceList
        U { compileExpr(agentInstanceList[i].s-EXPRESSION,
                agentInstanceList[i+1].s-EXPRESSION.startLabel) // compileExpr
            | i ∈ 1..agentInstanceList.length - 1 }
        ∪ compileExpr(agentInstanceList.last.s-EXPRESSION, uniqueLabel(aipd, 1))
    endif
    ∪ { mk-PRIMITIVE(uniqueLabel(aipd,1),
            mk-AGENTINSTPID(agentInstanceList,
                < uniqueLabel(agentInstanceList[i].s-EXPRESSION, 1) : i ∈ 1..agentInstanceList.length >,
                // value labels for num expressions
                next // continue label
            ) // mk-AgentInstancePid
        ) // mk-Primitive
        }
| a=Any-expression(id) then
        {mk-PRIMITIVE(uniqueLabel(a,1), mk-ANYVALUE(id, next)) }
| a=Assignment(id, expr) then
        if id.refersto1.s- Aggregation-kind = PART //  value assignment
        then
            compileExpr(expr, uniqueLabel(a,1))
            ∪ {mk-PRIMITIVE(uniqueLabel(a,1), mk-TASK(id, uniqueLabel(expr,1), next) )}
        else //  reference assignment – RHS expr is Variable-access or Operation-application for field
            if // operation application for access of field of variable on RHS of assign to REF
                expr ∈ Operation-application
            then // operation application for access of field of variable on RHS of assign to REF
                { mk-PRIMITIVE(uniqueLabel(a,1), mk-REFFIELDASSIGN(id, expr, next) ) }
            else
                compileExpr(expr, next) // handle RHS of assign to REF in variable access
            endif
        endif
| c=Call-node(*, procedureId, Actual-parameters(exprList)) then
        if exprList = empty then ∅
        else    U{ compileExpr(exprList[i], exprList[i+1].startLabel) | i ∈ 1..exprList.length - 1 }
            ∪ compileExpr(exprList.last, uniqueLabel(c,1))
        endif
        ∪
        let paramDefs = procedureId.refersto1.s-PROCPARAM-seq in
            {mk-PRIMITIVE(uniqueLabel(c,1),
                mk-CALL(procedureId,
                    < if paramDefs[idx] ∈ In-parameter
                      then uniqueLabel(exprList[idx], 1) // label for value of In-parameter
                      else exprList[idx] // Inout-parameter or Out-parameter – variable Identifier
                      endif
                     | idx in (1..exprList.length )
                    >, // list param expressions – values and var ids
                    uniqueLabel(c,1), // label for the return value
                    next) // mk-Call
            )} // mk-Primitive
        endlet // paramDefs
| c=Closed-range(r1, r2) then
        compileExpr(r1, r2.startLabel)
        ∪ compileExpr(r2, uniqueLabel(c, 1))
        ∪ { mk-PRIMITIVE(uniqueLabel(c, 1),
                mk-OPERATIONAPPLICATION(sdlAnd, // sdlAnd only used here.
                    < uniqueLabel(r1, 1), uniqueLabel(r2, 1) >, undefined, next)
            ) // Primitive
```

```
        }
| c=Compound-node(name, variables, initNodes, whileNode(whilexprs, final), trans, stepNodes) then
    // Create local variables Z.102 11.14.1 Semantics (a)
    {  mk-PRIMITIVE(uniqueLabel(c,1),
         mk-SCOPE(name, variables,
           if initNodes = empty
           then if whilexprs = empty then trans.startLabel else whilexprs[1].startLabel endif
           else initNodes.head.startLabel
           endif, //StartLabel
           if stepNodes = empty
           then if whilexprs = empty then trans.startLabel else whilexprs[1].startLabel endif
           else stepNodes[1].startLabel
           endif, //StepLabel
           next // ContinueLabel
         ) // Scope
       ) // Primitive
    }
    ∪ // Init graph Z.102 11.14.1 Semantics (b)
    if initNodes ≠ empty
    then
      U{ compileExpr( initNodes[i], initNodes[i+1]. startLabel) | i ∈ 1..initNodes.length - 1 }
      ∪ compileExpr( initNodes.last,
          if whilexprs = empty then trans.startLabel else whilexprs.head.startLabel endif
        ) // compileExpr
    endif // initNodes empty
    ∪ // Init graph Z.102 11.14.1 Semantics (c) evaluate if any While expr false
    if whilexprs ≠ empty // Boolean expressions for a While-graph-node
    then
      U{ compileExpr(whilexprs[i], uniqueLabel(whilexprs[i], 2)) // eval while expr – goto decision prim
        ∪ { mk-PRIMITIVE(uniqueLabel(whilexprs[i], 2), // decision primitive for while
            mk-DECISION(uniqueLabel(whilexprs[i],1), // value of while exprs [i]
              { mk-ANSWER( undefined, // undefined is treated as SDL false in the Decision action
                  if final ≠ empty then uniqueLabel(final,1) else next endif) // mk-Answer
              }, // answer set for decision
              whilexprs[i+1].startLabel // if not false continue next while expr
            ) // mk-Decision
          ) } // mk-Primitive, singleton set for decision
      | i ∈ 1..whilexprs.length - 1 }  // iterate – set for distributed union ofeval+decision pairs
    ∪ compileExpr(whilexprs.last, uniqueLabel(whilexprs.last, 2))
    ∪ { mk-PRIMITIVE(uniqueLabel(whilexprs.last, 2), // decision primitive for last while
        mk-DECISION(uniqueLabel(whilexprs.last,1), // value of last while
          { mk-ANSWER( undefined, // undefined is treated as SDL false in the Decision action
              if final ≠ empty then uniqueLabel(final,1) else next endif) // mk-Answer
          }, // answer set for decision
          trans.startLabel // if not false goto transition
        ) // mk-Decision
      ) } // mk-Primitive, singleton set for decision of last while
    endif // Boolean expressions for a While-graph-node
    ∪ compile(trans) // Z.102 11.14.1 Semantics (d) evaluate transition
    // transition Terminator determines what happens next
    ∪ // continue matching loop Z.102 11.14.1 Semantics (e) – do step nodes – use scope StepLabel
    if stepNodes ≠ empty
    then
      U{ compileExpr( stepNodes[i], stepNodes[i+1].startLabel) | i ∈ 1..stepNodes.length - 1 }
      ∪ compileExpr( stepNodes.last,
          if whilexprs = empty then trans.startLabel else whilexprs.head.startLabel endif
        ) // compileExpr
    endif
    ∪ compile(final) // Z.102 11.14.1 Semantics (c) – while expr false – finalization
```

```
| c=Conditional-expression(boolExpr, consExpr, altExpr) then
        compileExpr(boolExpr, uniqueLabel(c, 2))
    ∪ compileExpr(consExpr, next)
    ∪ compileExpr(altExpr, next)
    ∪ {  mk-PRIMITIVE(uniqueLabel(c,2),
            mk-OPERATIONAPPLICATION(sdlTrue.refersto1, empty, undefined, uniqueLabel(c, 1))
         ) // Primitive
      }
    ∪ {  mk-PRIMITIVE(uniqueLabel(c, 1),
            mk-DECISION(uniqueLabel(boolExpr, 1),
               { mk-ANSWER(uniqueLabel(c, 2), consExpr.startLabel) },
               altExpr.startLabel) // Decision
         ) // Primitive
      }

| c=Create-request-node(agentId ∈ Identifier, Actual-parameters(exprList)) then
    if exprList = empty then ∅
    else  ⋃{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }
       ∪ compileExpr(exprList.last, uniqueLabel(c,1))
    endif
    ∪ {  mk-PRIMITIVE(uniqueLabel(c,1),
         mk-CREATE(agentId, <uniqueLabel(e,1) | e in exprList >, next))
      }

| c=Create-request-node(THIS, Actual-parameters(exprList)) then
    if exprList = empty then ∅
    else  ⋃{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }
       ∪ compileExpr(exprList.last, uniqueLabel(c,1))
    endif
    ∪ {  mk-PRIMITIVE(uniqueLabel(c,1),
         mk-CREATE(parentAS1ofKind(c,Agent-definition).s-Identifier,
            <uniqueLabel(e,1) | e in exprList >, next))
      }

| de=Decoding-expression(expr, Encoding-path(encpath)) then
// Result is value of choice type that can hold any signal for the encoding pth
        compileExpr(expr, uniqueLabel(de,1))
  ∪ {  mk-PRIMITIVE(uniqueLabel(de,1),
         mk-ZDECODING(uniqueLabel(expr,1), encpath, next) // mk-ZDecoding
       ) // mk-Primitive
    }

| ede=Encoded-expression(expr) then
    compileExpr(expr, next)

| ege=Encoding-expression(sigid, exprList, encpath) then
  if exprList = empty then ∅
  else  ⋃{ compileExpr(exprList[i], exprList[i+1].startLabel) | i ∈ 1..exprList.length - 1 }
     ∪ compileExpr(exprList.last, uniqueLabel(ege, 3))
  endif
  ∪ {  mk-PRIMITIVE(uniqueLabel(ege,3),
       mk-ZENCODINGPAR( // evaluate structure value to pass aschoice parameter
          sigid,
          < uniqueLabel(expr[i], 1) : i ∈ 1..exprList.length >, // value labels for expressions
          uniqueLabel(ege,3), // value label for returned structure value
          uniqueLabel(ege,2)) // continue label for ZEncodingVal, mk-ZEncodingPar
       ) // mk-Primitive
    }
  ∪ {  mk-PRIMITIVE(uniqueLabel(ege,2),
       mk-ZENCODINGVAL( // choice value
          sigid,
          uniqueLabel(ege,3), // value label for structure value returned from ZEncodingPar
          uniqueLabel(ege,2), // value label for returned choice value
          uniqueLabel(ege,1)) // continue label for ZEncoding, mk-ZEncodingPar
```

```
        ) // mk-Primitive
    }
∪ { mk-PRIMITIVE(uniqueLabel(ege,1),
        mk-ZENCODING(
            uniqueLabel(expr,2), // value label for choice value returned from ZEncodingVal
            encpath,
            next) // continue label, mk-ZEncoding
        ) // mk-Primitive
    }

| Graph-node(action) then compileExpr(action, next)

| l=Literal(id) then
    {mk-PRIMITIVE(uniqueLabel(l,1),
        mk-OPERATIONAPPLICATION(id.refersto1, empty, undefined, next)) }

| en=Negative-equality-expression(first, second) then
    // more study needed for call of a procedure with a REF result, or null literal signature
    if  (   (first ∈ Variable-definition ∧ first.s-Aggregation-kind = REF)
          ∨ (first ∈ Parameter ∧ first.s-Parameter-aggregation.s-Aggregation-kind = REF)
        ) // first operand REF variable or parameter
        ∧
        (   (second ∈ Variable-definition ∧ second.s-Aggregation-kind = REF)
          ∨ (second ∈ Parameter ∧ second.s-Parameter-aggregation.s-Aggregation-kind = REF)
        ) // second operand REF variable or parameter
    then // special case of both operands REF variable or parameter; handled in variable access
        compileExpr(first, next)
    else // normal case
        compileExpr(first, second.startLabel)
    ∪ compileExpr(second, uniqueLabel(en,1))
    ∪ { mk-PRIMITIVE(uniqueLabel(en,1),
            mk-EQUALITY(uniqueLabel(first,1), uniqueLabel(second,1), next, false) // Equality
        ) // Primitive
        }
    endif // special or normal case

| n=Now-expression() then
    {mk-PRIMITIVE(uniqueLabel(n,1), mk-SYSTEMVALUE(kNow, next)) }

| o=Offspring-expression() then
    {mk-PRIMITIVE(uniqueLabel(o,1), mk-SYSTEMVALUE(kOffspring, next)) }

| o=Open-range(id, expr) then
        compileExpr(expr, uniqueLabel(o, 1))
    ∪ { mk-PRIMITIVE(uniqueLabel(o, 1),
            mk-OPERATIONAPPLICATION(id.refersto1,
                < rangeCheckLabel, uniqueLabel(expr, 1) >, undefined, next) // OperationApplication
        ) // Primitive
        }

| o=Operation-application(id, Actual-parameters(exprList)) then
// Assignment of an operation-application for a field access to a REF variable handled by Assignment compile
    if exprList = empty then ∅
    else ⋃{ compileExpr(exprList[i], exprList[i+1].startLabel) | i ∈ 1..exprList.length - 1 }
        ∪ compileExpr(exprList.last, uniqueLabel(o,1))
    endif
    ∪ { mk-PRIMITIVE(uniqueLabel(o,1),
            mk-OPERATIONAPPLICATION(id.refersto1,
                < uniqueLabel(e, 1) | e in exprList >,
                undefined,
                next) // OperationApplication
        ) // Primitive
        }

| o=Output-node(sig ∈ Identifier, Actual-parameters(exprList), actdelay, sigpri, sigdest, via ) then
```

let *dest* =
if *sigdest*.**s-***Signal-destination* = *undefined*
then *undefined*
else *sigdest*.**s-***Signal-destination*.**s1-implicit** //{ Expression | Agent-identifier | THIS }of Signal-destination
endif
in
let *destnum* =
if *sigdest*.**s-***Signal-destination* = *undefined* ∨ *sigdest*.**s2-implicit** = *undefined*
then *undefined*
else *sigdest*.**s-***Signal-destination*.**s2-implicit** // [Destination-number] of Signal-destination of Output-node
endif
in
if *exprList* = *empty* **then** ∅
else
U{ *compileExpr*(*exprList*[*i*], *exprList*[*i*+1].*startLabel*) | *i* ∈ 1..*exprList.length* - 1 }
∪ *compileExpr*(*exprList.last*, *actdelay.startLabel*)
endif
∪ *compileExpr*(*actdelay*, *sigpri.startLabel*)
∪ *compileExpr*(*sigpri*,
if *dest* ∈ *EXPRESSION*
then *dest.startLabel* // signal destination is an expression – continue label is dest expression
elseif *destnum* ∈ *EXPRESSION* // signal destination Agent-id or THIS and non empty destnum
then *destnum.startLabel* // continue label is destnum expression
else *uniqueLabel*(*o*,1) // continue label for Output action
endif)
∪ **if** *dest* ∈ *EXPRESSION* **then** *compileExpr*(*dest*,
if *destnum* ∈ *EXPRESSION* // non empty destnum
then *destnum.startLabel* // continue label is destnum expression
else *uniqueLabel*(*o*,1) // continue label for Output action
endif)
else ∅
endif
∪ **if** *destnum* ∈ *EXPRESSION* **then** *compileExpr*(*destnum*, *uniqueLabel*(*o*,1)) **else** ∅ **endif**
∪ { **mk-***PRIMITIVE*(*uniqueLabel*(*o*,1),
mk-*OUTPUT*(*sig*,
< *uniqueLabel*(*e*,1) | *e* **in** *exprList* >,
uniqueLabel(*actdelay*,1),
uniqueLabel(*sigpri*,1),
if *dest* ∈ *EXPRESSION* **then** *uniqueLabel*(*dest*, 1) **else** *dest* **endif**, //undefined if sigdest undef
if *destnum* ∈ *EXPRESSION* **then** *uniqueLabel* (*destnum*, 1) **else** *undefined* **endif**,
via,
next) // Output
) //Primitive
}
endlet // destnum
endlet // dest

| *o=Output-node*(*ex* ∈ *EXPRESSION*, *actdelay*, *sigpri*, *sigdest*, *via*) **then**
// This is essentially the same as Output with a signal, except the expression ex is evaluated,
// and the action OutputX is called that has the value label for ex replacing the signal id and parameters.
let *dest* =
if *sigdest*.**s-***Signal-destination* = *undefined*
then *undefined*
else *sigdest*.**s-***Signal-destination*.**s1-implicit** //{ Expression | Agent-identifier | THIS }of Signal-destination
endif
in
let *destnum* =
if *sigdest*.**s-***Signal-destination* = *undefined* ∨ *sigdest*.**s2-implicit** = *undefined*

```
    then undefined
    else sigdest.s2-implicit // [ Destination-number ] of Signal-destination of Output-node
    endif
    in
        compileExpr(ex, actdelay.startLabel)
    ∪ compileExpr(actdelay, sigpri.startLabel )
    ∪ compileExpr(sigpri,
        if dest ∈ EXPRESSION
        then dest.startLabel // signal destination is an expression – continue label is dest expression
        elseif destnum ∈ EXPRESSION // signal destination Agent-id or THIS and non empty destnum
        then destnum.startLabel // continue label is destnum expression
        else uniqueLabel(o,1) // continue label for OutputX
        endif)
    ∪ if dest ∈ EXPRESSION then compileExpr(dest,
            if destnum ∈ EXPRESSION // non empty destnum
            then destnum.startLabel // continue label is destnum expression
            else uniqueLabel(o,1) // continue label for OutputX
            endif)
        else ∅
        endif
    ∪ if destnum ∈ EXPRESSION then compileExpr(destnum, uniqueLabel(o,1)) else ∅ endif
    ∪ {   mk-PRIMITIVE(uniqueLabel(o,1),
              mk-OUTPUTX(
                  uniqueLabel(ex,1),
                  uniqueLabel(actdelay,1),
                  uniqueLabel(sigpri,1),
                  if dest ∈ EXPRESSION then uniqueLabel(dest, 1) else dest endif, //undefined if sigdest undef
                  if destnum ∈ EXPRESSION then uniqueLabel (destnum, 1) else undefined endif,
                  via,
                  next) // OutputX
          ) //Primitive
      } // do Output
    endlet // destnum
    endlet // dest
```

As described in the *Abstract grammar* of clause 11.13.4 of [ITU-T Z.104]: 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 SEP

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; SEP

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.

```
| o=Output-node(ee ∈ Encoded-expression, actdelay, priority, sigdest, via ) then
    let dest =
    if sigdest.s-Signal-destination  = undefined
    then undefined
    else  sigdest.s-Signal-destination.s1-implicit  //{  Expression  |  Agent-identifier  |  THIS  }of  Signal-
destination
    endif
    in
    let destnum =
    if sigdest.s-Signal-destination  = undefined ∨ sigdest.s2-implicit = undefined
    then undefined
    else sigdest.s2-implicit // [ Destination-number ] of Signal-destination of Output-node
    endif
    in
        compileExpr(ee, actdelay.startLabel) // evaluate the Encoded-expression
```

```
∪ compileExpr(actdelay, sigpri.startLabel )
∪ compileExpr(sigpri,
  if dest ∈ EXPRESSION
  then dest.startLabel // signal destination is an expression – continue label is dest expression
  elseif destnum ∈ EXPRESSION // signal destination Agent-id or THIS and non empty destnum
  then destnum.startLabel // continue label is destnum expression
  else uniqueLabel(o,1) // continue label for Call decode
  endif)
∪ if dest ∈ EXPRESSION then compileExpr(dest,
      if destnum ∈ EXPRESSION // non empty destnum
      then destnum.startLabel // continue label is destnum expression
      else uniqueLabel(o,1) // continue label for Call decode
      endif)
  else ∅
  endif
∪ if destnum ∈ EXPRESSION then compileExpr(destnum, uniqueLabel(o,1)) else ∅ endif
∪ {  mk-PRIMITIVE(uniqueLabel(o,1),
        mk-CALL(
          via.s-Identifier-seq[1].refersto1.s-Encoding-rules.s2-Identifier, // decode proc id
          < ee >,
          uniqueLabel(o,1),
          uniqueLabel(o,2) // goto OutputZ
        ) // call the decode procedure for ee – result in value label (o,1)
  } // decode ee to choice value
∪ {  mk-PRIMITIVE(uniqueLabel(o,2),
        mk-OUTPUTZ(
          uniqueLabel(o,1), // value label for decoded value of ee for signal
          uniqueLabel(actdelay,1),
          uniqueLabel(sigpri,1),
          if dest ∈ EXPRESSION then uniqueLabel(dest, 1) else dest endif, //undefined if sigdest undef
          if destnum ∈ EXPRESSION then uniqueLabel (destnum, 1) else undefined endif,
          via,
          next) // OutputZ
      ) //Primitive
  } // do OutputZ
endlet // destnum
endlet // dest
```

As described in the *Abstract grammar* of clause 11.13.4 of [ITU-T Z.104]: 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*.

```
| p=Parent-expression() then
      {mk-PRIMITIVE(uniqueLabel(p,1), mk-SYSTEMVALUE(kParent, next)) }

| ep=Positive-equality-expression(first, second) then
      // more study needed for call of a procedure with a REF result, or null literal signature
      if  (     (first ∈ Variable-definition ∧ first.s-Aggregation-kind = REF)
            ∨  (first ∈ Parameter ∧ first.s-Parameter-aggregation.s-Aggregation-kind = REF)
          ) // first operand REF variable or parameter
          ∧
          (     (second ∈ Variable-definition ∧ second.s-Aggregation-kind = REF)
            ∨  (second ∈ Parameter ∧ second.s-Parameter-aggregation.s-Aggregation-kind = REF)
          ) // second operand REF variable or parameter
      then
          compileExpr(first, next) // special case of both operands REF; handled in compile var access
      else // normal case
          compileExpr(first, second.startLabel)
      ∪ compileExpr(second, uniqueLabel(ep,1))
      ∪ {  mk-PRIMITIVE(uniqueLabel(ep,1),
```

```
                mk-EQUALITY(uniqueLabel(first,1), uniqueLabel(second,1), next, true) // Equality
            ) // Primitive
        }
    endif // special or normal case

| r=Range-check-expression(expr, *, range) then
        compileExpr(expr, uniqueLabel(r,1))
    ∪ compileExpr(range, undefined)
    ∪ {  mk-PRIMITIVE(uniqueLabel(r,1),
            mk-SETRANGECHECKVALUE(uniqueLabel(expr,1),                uniqueLabel(r,2))
//SetRangeCheckValue
            ) // Primitive
        }
    ∪ {mk-PRIMITIVE(uniqueLabel(r,2), mk-SCOPE(undefined, ∅, range.startLabel, undefined, next)) }

| r=Reset-node(timerId, exprList) then
    if exprList = empty then ∅
    else compileExpr(exprList.last, uniqueLabel(r,1))
        ∪ ⋃{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }
    endif
    ∪ {mk-PRIMITIVE(uniqueLabel(r,1), mk-RESET(timerId, <uniqueLabel(e,1) | e in exprList >, next)) }

| r=Range-condition(items) then
  let iseq = items.setToSeq in
        {  mk-PRIMITIVE(uniqueLabel(r,1),
           mk-OPERATIONAPPLICATION(
              sdlFalse.refersto1,  empty, undefined, uniqueLabel(r, iseq.length+1))
        }
    ∪ {  compileExpr(iseq[idx], uniqueLabel(r, idx)) | idx ∈ toSet(1..iseq.length)
        }
    ∪ {  mk-PRIMITIVE(uniqueLabel(r, idx),
            mk-OPERATIONAPPLICATION(sdlOr, // sdlOr only used here
                < uniqueLabel(r, idx+1), uniqueLabel(iseq[idx],1) >,
                undefined,
                if idx=1 then next else iseq[idx-1].startLabel endif) // OperationApplication
            ) // Primitive
        | idx ∈ toSet(1..iseq.length) }
    ∪ { mk-PRIMITIVE(uniqueLabel(r, 0), mk-BREAK(undefined)) }
  endlet
```

The *Range-condition* above is computed as follows. First, a *false* value is evaluated. Then all the expressions for the range condition items (sequentialized from a set) are evaluated; the results are cumulated using OR. Afterwards, the enclosing scope is left using a break.

```
| s=Self-expression() then
    {mk-PRIMITIVE(uniqueLabel(s,1), mk-SYSTEMVALUE(kSelf, next)) }

| s=Sender-expression() then
    {mk-PRIMITIVE(uniqueLabel(s,1), mk-SYSTEMVALUE(kSender, next)) }

| s=Set-node(expr, timerId, exprList) then
    if exprList = empty then ∅
    else compileExpr(exprList.last, expr.startLabel)
        ∪ ⋃{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }
    endif
    ∪ compileExpr(expr, uniqueLabel(s,1))
    ∪ {  mk-PRIMITIVE(uniqueLabel(s,1),
            mk-SET(uniqueLabel(expr,1), timerId, <uniqueLabel(e,1) | e in exprList >, next)
            ) // Primitive
        }

| s=Signal-expression() then
    {mk-PRIMITIVE(uniqueLabel(s,1), mk-SYSTEMVALUE(kSignal, next)) }

| s=Signallist-expression() then
```

{**mk-***PRIMITIVE*(*uniqueLabel*(*s*,1), **mk-***SYSTEMVALUE*(*kSignallist*, *next*)) }

| *s=State-expression*() **then**
{**mk-***PRIMITIVE*(*uniqueLabel*(*s*,1), **mk-***SYSTEMVALUE*(*kState*, *next*)) }

| *t=Timer-active-expression*(*id*, *exprList*) **then**
if *exprList = empty* **then** ∅
else ⋃{ *compileExpr*(*exprList*[*i*], *exprList*[*i*+1]. *startLabel*) | *i* ∈ 1..*exprList.length* - 1 }
∪ *compileExpr*(*exprList.last*, *uniqueLabel*(*t*,1))
endif
∪ {**mk-***PRIMITIVE*(*uniqueLabel*(*t*,1),**mk-***TIMERACTIVE*(*id*,< *uniqueLabel*(*e*, 1) | *e* **in** *exprList* >, *next*))}

| *t=Timer-remaining-duration*(*id*, *exprList*) **then**
if *exprList = empty* **then** ∅
else ⋃{ *compileExpr*(*exprList*[*i*], *exprList*[*i*+1]. *startLabel*) | *i* ∈ 1..*exprList.length* - 1 }
∪ *compileExpr*(*exprList.last*, *uniqueLabel*(*t*,1))
endif
∪ {**mk-***PRIMITIVE*(*uniqueLabel*(*t*,1),**mk-***TIMERREMAINING*(*id*,<*uniqueLabel*(*e*,1)|*e* **in** *exprList*>,*next*))}

| *tce=Type-check-expression*(*expr*, *sort*) **then**
compileExpr(*expr*, *uniqueLabel*(*tce*, 1))
∪ { **mk-***PRIMITIVE*(*uniqueLabel*(*tce*,1),
mk-*TYPECHECK*(
uniqueLabel(*expr*, 1),
sort,
next // continue label
) // mk-TypeCheck
) // mk-Primitive
}

| *tc=Type-coercion*(*expr*, *sort*) **then**
compileExpr(*expr*, *uniqueLabel*(*tce*, 1))
∪ { **mk-***PRIMITIVE*(*uniqueLabel*(*tce*,1),
mk-*TYPECOERCION*(
uniqueLabel(*expr*, 1),
sort,
next // continue label
) // mk-*TYPECOERCION*
) // mk-Primitive
}

| *v=Value-returning-call-node*(*, *procedureId*, *Actual-parameters*(*exprList*)) **then**
if *exprList = empty* **then** ∅
else ⋃{ *compileExpr*(*exprList*[*i*], *exprList*[*i*+1].*startLabel*) | *i* ∈ 1..*exprList.length* - 1 }
∪ *compileExpr*(*exprList.last*, *uniqueLabel*(*v*,1))
endif
∪
let *paramDefs = procedureId.refersto1*.**s-***PROCPARAM*-**seq in**
{ **mk-***PRIMITIVE*(*uniqueLabel*(*v*,1),
mk-*CALL*(*procedureId*,
< **if** *paramDefs*[*idx*] ∈ *In-parameter*
then *uniqueLabel*(*exprList*[*idx*], 1) // label for value of In-parameter
else *exprList*[*idx*] // Inout-parameter or Out-parameter – variable Identifier
endif
| *idx* **in** (1..*exprList.length*)
>, // list param expressions
uniqueLabel(*v*,1), // label for return value
next
) // Call
) // Primitive
}
endlet // paramDefs

| *v=Variable-access*(*id*) **then**
{**mk-***PRIMITIVE*(*uniqueLabel*(*v*,1), **mk-***VAR*(*id, next*)) }

This concludes the definition of the expression compilation function.

otherwise ∅ // empty set if no matching pattern
endcase // end of the compileExpr function definition

F3.2.2.4 Start labels

This clause introduces the function *startLabel*, which defines the start labels of all behavioural syntax constructs.

startLabel(*x*: *DEFINITIONAS1*): *LABEL* =$_{\text{def}}$
case *x* **of**
| *a=Action-return-node*() **then** *uniqueLabel*(*a*,1)
| *a=Active-agents-expression*(*) **then** *uniqueLabel*(*a*,1)
| *a= Agent-instance-pid-value*(*aiList*) **then**
if *aiList = empty* **then** *uniqueLabel*(*a*,1) **else** *aiList.head*.**s-***EXPRESSION*.*startLabel* **endif**
| *d=Any-decision*(*) **then** *uniqueLabel*(*d*,1)
| *a=Any-expression*(*) **then** *uniqueLabel*(*a*,1)
| *Assignment*(*, *expr*) **then** *expr.startLabel*
| *b= Break-node*(*) **then** *uniqueLabel*(*b*,1)
| *c=Call-node*(*, *, *Actual-parameters*(*exprList*)) **then**
if *exprList = empty* **then** *uniqueLabel*(*c*,1) **else** *exprList.head.startLabel* **endif**
| *Closed-range*(*, *) **then** *uniqueLabel*(*c*,1)
| *c=Compound-node*(*, *, *, *, *, *, *) **then** *uniqueLabel*(*c*,1)
| *c=Conditional-expression*(*, *, *) **then** *uniqueLabel*(*c*,1)
| *Connect-node*(*, *trans*) **then** *trans.startLabel*
| *c= Continue-node*(*) **then** *uniqueLabel*(*c*,1)
| *Continuous-signal*(*, *, *, *trans*) **then** *trans.startLabel*
| *c=Create-request-node*(*, *Actual-parameters*(*exprList*)) **then**
if *exprList = empty* **then** *uniqueLabel*(*c*,1) **else** *exprList.head.startLabel* **endif**
| *n=Dash-nextstate*(*) **then** *uniqueLabel*(*n*,1)
| *Decision-answer*(*r, **) **then** *r.startLabel*
| *Decision-body*(*question, *, *, **) **then** *question.startLabel*
| *Decision-node*(*body*) **then** *body.startLabel*
| *Decoding-expression*(*e, **) **then** *e.startLabel*
| *Encoding-expression*(*, *el, **) **then** *el.head.startLabel*
| *Free-action*(*, *trans*) **then** *trans.startLabel*
| *Graph-node*(*action*) **then** *action.startLabel*
| *Input-node*(*, *, *, *, *, *, *trans*) **then** *trans.startLabel*
| *v=Identifier*(*, **) **then** *uniqueLabel*(*v*,1)
| *j= Join-node*(*) **then** *uniqueLabel*(*j*,1)
| *l=Literal*(*) **then** *uniqueLabel*(*l*,1)
| *n=Named-nextstate*(*, *Nextstate-parameters*(*Actual-parameters*(*exprList*), *)) **then**
if *exprList = empty* **then** *uniqueLabel*(*n*,1) **else** *exprList.head.startLabel* **endif**
| *n=Named-nextstate*(*, *undefined*) **then** *uniqueLabel*(*n*,1)
| *n=Named-return-node*(*) **then** *uniqueLabel*(*n*,1)
| *Named-start-node*(*, *trans*) **then** *trans.startLabel*
| *Negative-equality-expression*(*first, **) **then** *first.startLabel*
| *n=Now-expression*() **then** *uniqueLabel*(*n*,1)
| *o=Offspring-expression*() **then** *uniqueLabel*(*o*,1)
| *Open-range*(*, *expr*) **then** *expr.startLabel*
| *o= Operation-application*(*, *Actual-parameters*(*exprList*)) **then**
if *exprList = empty* **then** *uniqueLabel*(*o*,1) **else** *exprList.head.startLabel* **endif**
| *o= Output-node*(*sig* ∈ *Identifier, Actual-parameters*(*exprList*), *, *, *sigdest*, *) **then**
if *sigdest* ≠ *undefined* **then** *sigdest.startLabel*
elseif *exprList = empty* **then** *uniqueLabel*(*o*,1)
else *exprList.head.startLabel* **endif**
| *o=Output-node*(*sig* ∈ (*EXPRESSION* ∪ *Encoded-expression*), *, *, *sigdest*, *) **then**
if *sigdest* ≠ *undefined* **then** *sigdest.startLabel* **else** *uniqueLabel*(*o*,1) **endif**
| *p=Parent-expression*() **then** *uniqueLabel*(*p*,1)

```
| Positive-equality-expression(first, *) then first.startLabel
| Procedure-start-node(trans) then trans.startLabel
| r=Reset-node(*, exprList) then
    if exprList = empty then uniqueLabel(r,1) else exprList.head.startLabel endif
| Range-check-expression(*, expr) then expr.startLabel
| s=Self-expression() then uniqueLabel(s,1)
| Set-node(when, *, *) then when.startLabel
| s=Sender-expression() then uniqueLabel(s,1)
| s=Signal-destination(d, *) then
    if d = THIS ∨ d ∈ Identifier then uniqueLabel(d,1) else d.startLabel endif
| s=Signal-expression() then uniqueLabel(s,1)
| s=Signallist-expression() then uniqueLabel(s,1)
| Spontaneous-transition(*, trans) then trans.startLabel
| s=State-expression() then uniqueLabel(s,1)
| State-timer(*, *, *, *, *, *, trans) then trans.startLabel
| State-start-node(trans) then trans.startLabel
| s= Stop-node() then uniqueLabel(s,1)
| Terminator(terminator) then terminator.startLabel
| t=Timer-active-expression(*, exprList) then
    if exprList = empty then uniqueLabel(t,1) else exprList.head.startLabel endif
| t=Timer-remaining-duration(*, exprList) then
    if exprList = empty then uniqueLabel(t,1) else exprList.head.startLabel endif
| t=Transition(nodes, endnode) then
    if t.parentAS1.parentAS1 ∈ State-node then uniqueLabel(t,1) // insert the Leavestatenode
    elseif nodes = empty then endnode.startLabel
    else nodes.head.startLabel
    endif
| Type-check-expression(e, *) then e.startLabel
| Type-coercion(e, *) then e.startLabel
| v=Value-return-node(*) then uniqueLabel(v,1)
| v=Value-returning-call-node(*, *, Actual-parameters(exprList)) then
    if exprList = empty then uniqueLabel(v,1) else exprList.head.startLabel endif
| v=Variable-access(*) then uniqueLabel(v,1)
| Variable-definition(*, *, *, init) then
    if init = undefined then undefined else init.startLabel endif
endcase
```

F3.2.3 SDL-2010 abstract machine programs

For each SDL-2010 specification, the set of legal system runs are built using the SDL-2010 abstract machine and the compilation in clause F3.2.2.

F3.2.3.1 System initialization

Starting from any pre-initial state of *S*0, the initialization rules describe a recursive *unfolding* of the specified system instance according to its initial hierarchical structure. For each SDL-2010 agent instance, a corresponding ASM agent is created and initialized. Furthermore, ASM agents are created to model links and SDL-2010 agent sets.

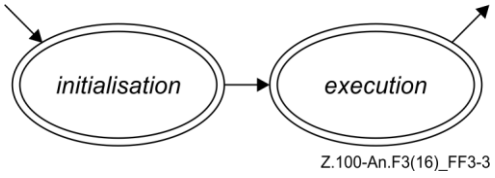

Figure F3-3 – Activity phases of SDL-2010 agents and agent sets (level 1)

During its lifetime, an agent first is in mode "initialisation", where its internal structure is built up. Then, it enters the mode "execution" and remains in this mode unless it is terminated.

F3.2.3.1.1 Pre-initial system state

This clause states some constraints on the set of initial states *S0* of the abstract state modelling a given SAM, i.e., the set of pre-initial states of the SAM. Further restrictions are defined in previous clauses, marked by the keyword **initially**. Usually, there is more than one pre-initial system state. It is only required that the system starts in one of these states.

```
initially
    if rootNodeAS1.s-Agent-definition ≠ undefined then
        system.agentAS1 = rootNodeAS1.s-Agent-definition ∧
        system.owner = undefined ∧
        system.agentMode1 = initialisation ∧
        system.program = AGENT-SET-PROGRAM
    else
        system.program = undefined
    endif
```

For a given SDL-2010 specification, the initial constraint distinguishes two cases. The first case applies when an agent definition is part of the SDL-2010 specification, i.e., when *rootNodeAS1*.**s**-*Agent-definition* ≠ *undefined*. Only then is the semantics defined to yield a dynamic behaviour. Since the system agent is the root of the agent hierarchy, it has no owner (*system.owner* = *undefined*). The SAM program of the agent *system* is the program applying to SDL-2010 agent sets in general. Further functions and domains are initialized when this program is executed, or are derived functions or derived domains. In the second case, no system agent is defined in the SDL-2010 specification; therefore, no behaviour is assigned via *program*.

F3.2.3.1.2 Agent set creation, initialization, and removal

ASM agents modelling SDL-2010 agent sets are created during system initialization and possibly dynamically, during system execution. They can be understood as containers that reflect certain structural aspects of SDL-2010 systems, in particular agent hierarchy and the connection structure. These structural aspects are crucial to the intelligibility of SDL-2010 specifications, and are therefore represented in the formal model, too.

```
CREATEALLAGENTSETS(ow:AGENT, atd:Agent-type-definition) ≡
    do forall ad: ad ∈ atd.collectAllAgentDefinitions
        CREATEAGENTSET(ow, ad)
    enddo

    where

        collectAllAgentDefinitions(atd: Agent-type-definition): Agent-definition-set =def
            if atd.s-Identifier = undefined then
                atd.s-Agent-definition-set
            else let typedef: Agent-type-definition = atd.s-Identifier.refersto1 in
                    atd.s-Agent-definition-set ∪ typedef.collectAllAgentDefinitions
                endlet
            endif
    endwhere
```

SDL-2010 agent sets are created when the surrounding SDL-2010 agent is initialized right after its creation. For each agent definition found via *collectAllAgentDefinitions*, an SDL-2010 agent set is created, taking inheritance into account.

```
CREATEAGENTSET(ow:SDLAGENT, ad:Agent-definition) ≡
    let typedef: Agent-type-definition = ad.s-Identifier.refersto1 in
    extend AGENT with sas
        sas.agentAS1 := ad
        sas.owner := ow
        CREATEALLGATES (sas, typedef)
```

```
        sas.program := AGENT-SET-PROGRAM
        sas.agentMode1 := initialisation
    endextend
    endlet
```

Creation of an SDL-2010 agent set is modelled by creating an ASM agent and initializing its control block. In particular, the node *Agent-definition* of the AST is assigned to the function *agentAS1*, the owner is determined, and the initial program is set. To complete the creation of the agent set, its interface as given by all its gates is created. Thus, these gates are ready to be connected by the owner of the agent set, an SDL-2010 agent instance. Further functions and domains are initialized when AGENT-SET-PROGRAM is executed, or are derived functions or derived domains. The initial agent instances of the considered SDL-2010 agent set are created when this program is executed. Apart from the creation of gates, there are strong similarities between this rule macro and the initial constraint, because *system* is an SDL-2010 agent set too.

The creation of SDL-2010 agent set instances relies on information of the abstract syntax tree. An element of domain *Agent-definition* defines the root from which this information can be accessed. In particular, there is an agent type identifier, which is a link to the agent type definition providing the internal structure of the agents, and their behaviour.

```
AGENT-SET-PROGRAM:
if Self.agentMode1 = initialisation then
    INITAGENTSET
endif
if Self.agentMode1 = execution then
    EXECAGENTSET
endif
```

Depending on the current agent mode, level 1, the activity phase is selected. After a single initialization step, the agent set is switched to the execution mode.

```
INITAGENTSET ≡
    let typedef: Agent-type-definition = Self.agentAS1.s-Identifier.refersto1 in
    if typedef.s-Agent-kind = SYSTEM then
        CREATEALLGATES(Self, typedef)
    endif
    CREATEALLAGENTS(Self, Self.agentAS1)
    Self.agentMode1:= execution
    endlet
```

The initialization of agent sets (and hence also of the agent *system*) is given by the rule macro INITAGENTSET, which is applied in the program AGENT-SET-PROGRAM. During initialization, the initial agent instances – in the case of *system* a single agent instance – are created. After this initialization, the ASM agent is switched to the execution mode.

In case of the SDL-2010 agent set *system*, the gates of the system instance are created. The reasons why this is done during initialization (and not at creation as for other agent sets) are technical.

```
REMOVEALLAGENTSETS(ow:SDLAGENT) ≡
    do forall sas: sas ∈ SDLAGENTSET ∧ sas.owner = ow
        REMOVEAGENTSET(sas)
    enddo

REMOVEAGENTSET(sas:SDLAGENTSET) ≡
    sas.owner := undefined
    sas.program := undefined
```

Removal of an agent set is modelled by resetting the program (and the owner) to *undefined*.

F3.2.3.1.3 Agent creation, initialization, and removal

The creation of SDL-2010 agent instances happens during system initialization, and possibly dynamically, during system execution. The creation as defined by the rule macro CREATEAGENT leaves an agent in what is called "pre-initial state". The agent's "initial state" is reached after agent initialization, which is defined subsequently.

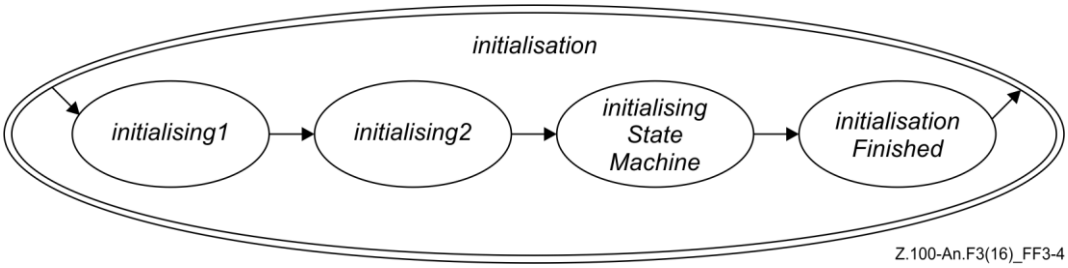

Figure F3-4 – Activity phases of SDL-2010 agents: initialization (level 2)

The initialization of an agent is decomposed into a sequence of phases, as shown in the state diagram above. In each of these phases, certain parts of the agent's structure are created. After agent initialization, the agent execution is started.

```
CREATEALLAGENTS(ow:SDLAGENTSET, ad:Agent-definition) ≡
    ow.agentSetPids := empty
    do forall i: i ∈ 1..ad.s-Number-of-instances.s1-NAT // Initial-number
        CREATEAGENT(ow, undefined, ad)
    enddo
```

The initial number of agent instances of an agent set is defined in its *Agent-definition*. The macro CREATEALLAGENTS is used during system initialization, and possibly during system execution, when agent instances containing agent sets themselves are created dynamically.

```
CREATEAGENT(ow:SDLAGENTSET, pa: [SDLAGENT], ad:Agent-type-definition) ≡
    extend AGENT with sa
        INITAGENTCONTROLBLOCK(sa, ow, pa, ad)
        CREATEINPUTPORT(sa)
        sa.agentMode1 := initialisation
        sa.agentMode2 := initialising1
        sa.program := AGENT-PROGRAM
    endextend

    where

        INITAGENTCONTROLBLOCK(sa: SDLAGENT, ow:SDLAGENTSET, pa: [SDLAGENT],
            ad:Agent-type-definition) ≡
            sa.agentAS1 := ad
            sa.owner := ow
            sa.isActive := undefined
            sa.currentStartNodes := ∅
            sa.currentExitStateNodes := ∅
            sa.currentConnector := undefined
            sa.callingProcedureNode := undefined
            sa.currentSignalInst := undefined
            sa.parent := if pa ≠ undefined then pa.selfPid else undefined endif
            sa.sender := nullPid
            sa.signal:= undefined
            sa.offspring := nullPid
            sa.selfPid := mk-PID(sa, undefined)
            if pa ≠ undefined then
                pa.offspring := mk-PID(sa, undefined)
```

```
            endif
            if ow.agentAS1.s-Identifier.refersto1.s-Agent-kind = PROCESS then
                sa.stateAgent := ow.owner.stateAgent
            else // SYSTEM or BLOCK or other
                sa.stateAgent := sa
            endif
            ow.agentSetPids := ow.agentSetPids^ <sa.selfPid>
    endwhere
```

To create an agent, the controlled domain *AGENT* is extended. The control block of this new agent is initialized. An input port for receiving signals from other agents is created and attached to the new agent. The setting of agent modes and assignment of a program completes the creation of the agent.

```
AGENT-PROGRAM:
if Self.agentMode1 = initialisation then
    INITAGENT
elseif Self.agentMode1 = execution then
    if Self.ExecRightPresent then
        EXECAGENT
    else
        GETEXECRIGHT
    endif
endif
```

Depending on the current agent mode level 1, the activity phase is selected. After initialization, the agent is switched to the execution mode. Additionally, the agent synchronizes in case it belongs to a set of nested agents, in order to obtain an interleaving execution amongst these agents.

```
INITAGENT ≡
    let myDefinition: Agent-type-definition = Self.agentAS1.s-Identifier.refersto1 in
    if myDefinition.s-Abstract ≠ undefined
    then    raise(InvalidCall) // attempt to instantiate abstract agent type definition
    elseif  Self.agentMode2 = initialising1
    then
        CREATEAGENTVARIABLES(Self, myDefinition )
        CREATEALLAGENTSETS(Self, myDefinition )
        CREATESTATEMACHINE(myDefinition.s-State-machine)
        Self.agentMode2 := initialising2
    elseif Self.agentMode2 = initialising2
    then
        CREATEALLCHANNELS(Self, myDefinition )
         // no implicit links (done by DeliverSignals)
        Self.agentMode2 := initialisingStateMachine
    elseif Self.agentMode2 = initialisingStateMachine
    then    INITSTATEMACHINE
    elseif Self.agentMode2 = initialisationFinished
    then
        Self.agentMode1 := execution
        Self.agentMode2 := startPhase
    endif
    endlet
```

The initialization of agent instances starts in the "pre-initial state" and consists of four phases, triggered by agent modes. In the first phase, the inner "structure" of the agent is built up. This structure consists of the agent's local variable instances, its agent sets, and its state machine. A state machine is created even if it is not defined in the SDL-2010 specification; in this case, no behaviour is associated with the state machine. The information about this structure is drawn from the abstract syntax tree, in particular, from the part of the tree representing the agent's type definition.

Once the structure of the agent has been created, channels and links are established. Next, the state machine is initialized, i.e., a "hierarchical inheritance state graph" modelling the agent's state

machine is unfolded in a sequence of steps. Finally, execution is triggered by setting the agent modes.

```
REMOVEAGENT(sa:SDLAGENT) ≡
    sa.owner.agentSetPids := removePid(sa.selfPid, sa.owner.agentSetPids)
    REMOVEALLLINKS(sa)
    sa.program := undefined
    sa.owner := undefined
    where

    removePid(p: PID, plist: PID*): PID* =def
        if plist = empty then empty
        elseif plist.head = p then plist.tail
        else < plist.head > ⌢ removePid(p, plist.tail)
        endif
    endwhere
```

Removal of an agent is modelled by removing its pid from the agent pid list of the owning agent set, by removing all owned link agents and by resetting the program (and the owner) to *undefined*. The function *removePid* is used to remove a *PID* from the agent set pid list.

F3.2.3.1.4 Procedure creation and initialization

The creation of SDL-2010 procedure instances happens dynamically, during system execution. The creation as defined by the rule macro CREATEPROCEDURE leaves a procedure in what is called "pre-initial" state.

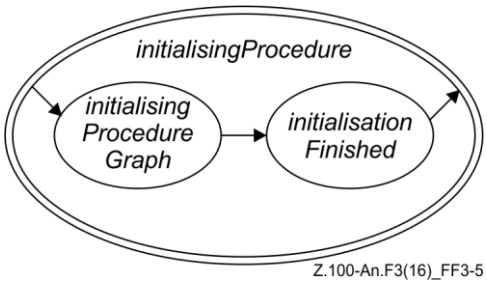

Figure F3-5 – Activity phases of SDL-2010 agents: firing of transitions (level 4)

The initialization of a procedure is decomposed into a sequence of phases, as shown in the state diagram above. In each of these phases, certain parts of the procedure's structure are created. After procedure initialization, the agent execution is continued.

```
CREATEPROCEDURE
    (pd:Procedure-definition, ap: CALLPARAM*, rl: [VALUELABEL], rRVI:[Identifier], cl:[CONTINUELABEL]) ≡
    if pd.s-Abstract ≠ undefined
    then   raise(InvalidCall) // attempt to instantiate abstract procedure definition
    else
        CREATEPROCEDUREGRAPH(pd, ap, rl, rRVI, cl)
        Self.agentMode3 := initialisingProcedure
        Self.agentMode4 := initialisingProcedureGraph
    endif // check for abstract procedure definition

INITPROCEDURE ≡
    if Self.agentMode4 = initialisingProcedureGraph then
        INITPROCEDUREGRAPH
    elseif Self.agentMode4 = initialisationFinished then
        Self.stateNodesToBeEntered :=
            {mk-STATENODEWITHENTRYPOINT (Self.currentProcedureStateNode, undefined)}
        Self.agentMode3 := enteringStateNode
```

```
        Self.agentMode4 := startPhase
        Self.currentLabel := undefined
    endif
```

The initialization of procedure instances starts in the "pre-initial state" and consists of two phases, triggered by agent modes. In the first phase, the inner "structure" of the procedure is built up. This structure consists of the procedure's local variable instances, and its state machine. The information about this structure is drawn from the abstract syntax tree, in particular, from the part of tree representing the procedure's type definition.

Once the structure of the procedure has been created, the state machine is initialized, i.e., a "hierarchical inheritance state graph" modelling the procedure's state machine is unfolded in a sequence of steps. Finally, execution is triggered by setting the agent modes, and by assigning the state node to be entered.

F3.2.3.1.5 Gate creation

Exchange of signals between SDL-2010 agents is modelled by means of *gates* from a controlled domain *GATE*. A gate forms an interface for *serial* and *unidirectional* communication between two or more agents.

```
CREATEALLGATES(ow:AGENT, atd: Agent-type-definition) ≡
    do forall gd: gd ∈ atd.collectAllGateDefinitions
        CREATEGATE(ow, gd)
    enddo

    where
        collectAllGateDefinitions(atd: Agent-type-definition): Gate-definition-set =def
            if atd.s-Identifier = undefined then
                atd.s-Gate-definition-set
            else
                let typedef: Agent-type-definition = atd.s-Identifier.refersto1 in
                atd.s-Gate-definition-set ∪
                typedef.collectAllGateDefinitions
                endlet
            endif
    endwhere
```

SDL-2010 agent sets are created when the surrounding SDL-2010 agent is initialized right after its creation. For each gate definition found via *collectAllGateDefinitions*, a gate is created, taking inheritance into account.

```
CREATEGATE(ow:AGENT, gd:Gate-definition) ≡
    if gd.s1-Identifier-set ≠ ∅ // In-signal-identifier-set
    then
        extend GATE with g
            g.myAgent := ow
            g.gateAS1 := gd
            g.schedule := empty
            g.direction := inDir
        endextend
    endif
    if gd.s2-Identifier-set ≠ ∅ // Out-signal-identifier-set
    then
        extend GATE with g
            g.myAgent := ow
            g.gateAS1 := gd
            g.schedule := empty
            g.direction := outDir
        endextend
    endif
```

For each SDL-2010 gate, one or two elements of the controlled domain *GATE* (also called "gates") are added, depending on whether the gate is uni-directional or bi-directional. The decision of which gates to create is based upon the signal identifier sets in the inward and outward direction, respectively. For each gate, the owning agent, the AST node representing the gate definition, and the direction are assigned to the corresponding functions. Furthermore, the schedule, i.e., the sequence of signals waiting to be forwarded, is initialized to be empty.

```
CREATEINPUTPORT(ow:AGENT) ≡
    extend GATE with g
        g.myAgent := ow
        g.gateAS1 := undefined
        g.schedule := empty
        g.direction := inDir
        ow.inport := g
    endextend
```

As it has turned out, input ports have strong similarities with elements of the domain *GATE* (called "gates"). Therefore, input ports are modelled as gates, and the same functions are defined and initialized. In addition, the created gate explicitly becomes the input port of the owning agent.

F3.2.3.1.6 Channel creation

Channels are modelled through unidirectional channel paths connecting a pair of gates.

```
CREATEALLCHANNELS(ow:AGENT, atd:Agent-type-definition) ≡
    do forall cd: cd ∈ atd.collectAllChannelDefinitions
        CREATECHANNEL(ow, cd)
    enddo

    where

        collectAllChannelDefinitions(atd: Agent-type-definition): Channel-definition-set =def
            if atd.s-Identifier = undefined then
                atd.s-Channel-definition-set
            else
                let typedef: Agent-type-definition = atd.s-Identifier.refersto1 in
                atd.s-Channel-definition-set ∪
                typedef.collectAllChannelDefinitions
                endlet
            endif
    endwhere
```

Channels are created by agents during the second phase of their initialization. For each element found via *collectAllChannelDefinitions*, a channel is created, taking inheritance into account.

```
CREATECHANNEL(ow:AGENT, cd:Channel-definition) ≡
    do forall cp: cp ∈ cd.s-Channel-path-set
        CREATECHANNELPATH(ow, cd.s-NODELAY, cp, cd)
```

Creating a channel amounts to creating the specified channel paths.

```
CREATECHANNELPATH(ow:AGENT, nd:[NODELAY], cp:Channel-path, cd:Channel-definition) ≡
    let origDef: Gate-definition = cp.s1-Identifier.refersto1 in // Originating-gate = Gate-identifier = Identifier
    let destDef: Gate-definition = cp.s2-Identifier.refersto1 in // Destination-gate = Gate-identifier = Identifier
    choose fromGate: fromGate ∈ GATE ∧ fromGate.gateAS1= origDef ∧
        (OuterGate(ow, fromGate, inDir) ∨ InnerGate(ow, fromGate, outDir) )
        choose toGate: toGate ∈ GATE ∧ toGate.gateAS1 = destDef ∧
                (OuterGate(ow, toGate, outDir) ∨ InnerGate(ow, toGate, inDir) )
            CREATELINK(ow,fromGate, toGate, nd, cp.s-Identifier-set, cd)
```

```
        endchoose
    endchoose

    where

        OuterGate(ow: AGENT, g: GATE, dir: DIRECTION): BOOLEAN =def
            g.myAgent = ow.owner ∧ g.direction = dir

        InnerGate(ow: AGENT, g: GATE, dir: DIRECTION): BOOLEAN =def
            g.myAgent.owner = ow ∧ g.direction = dir
    endwhere
```

A channel path is modelled as a link between two gates. The gates to be connected have already been created together with their agent sets. Originating and destination gates are distinguished, which defines the direction of the channel path. The correspondence between gate identifiers (referring to the AST) and gate instances is obtained by exploiting the functions *myAgent* and *direction* defined on gates.

F3.2.3.1.7 Link creation and removal

Agents of type *LINK* model the transport of signals. The behaviour of link agents is defined by the ASM program LINK-PROGRAM.

In addition to modelling explicit channel paths, links are used to model implicit channel paths that connect input gates (as defined by the derived function *ingates*) with the input port of an agent.

```
CREATELINK(ow:AGENT,
    fromGate:GATE,
    toGate:GATE,
    nd:[NODELAY],
    w:Identifier-set, // In-signal-identifier set
    cd:[Channel-definition]) ≡
extend LINK with l
    l.channelAS1 := cd
    l.owner := ow
    l.from := fromGate
    l.to := toGate
    l.noDelay := nd
    l.with := w
    l.program := LINK-PROGRAM
endextend

LINK-PROGRAM:
        if Self.from.queue ≠ empty then
            let si = Self.from.queue.head in
                if applicable(si.signalType,si.toArg,si.viaArg,Self.from,Self) then
                    DELETE(si,Self.from)
                    INSERT(si,now+Self.delay,Self.to) // insert in destination gate, adding the delay for this link
                    si.viaArg := si.viaArg \{Self.from.gateAS1.identifier1, Self.channelAS1.identifier1}
                endif
            endlet
        endif
```

A link agent models the connection between a pair of gates. Since links are finally combined into channel paths and channels, respectively, a delay characteristic is associated with them. Also, the signals that can be transported by the link are determined. LINK-PROGRAM defines the dynamic behaviour of link agents.

```
REMOVEALLLINKS(ow:AGENT) ≡
    do forall l: l ∈ LINK∧ l.owner = ow
        REMOVELINK(l)
    enddo
```

```
REMOVELINK(l:LINK) ≡
    l.program := undefined
    l.owner := undefined
```

Removal of a link agent is modelled by deleting the program and the owner.

F3.2.3.1.8 Variable creation

For each agent, composite state, procedure, and compound node instance, a set of local variables may be declared in an SDL-2010 specification. This leads to nested scopes, where a scope is associated with each refined state node.

```
CREATEAGENTVARIABLES(sa:SDLAGENT, atd:Agent-type-definition) ≡
    extend STATEID with sid
        sa.topStateId := sid
        if sa.stateAgent = sa // the agent where the variables are–if sa then stored locally in sa
        then    sa.state := initAgentState(undefined, sid, undefined, atd.collectAllVariableDefinitions)
        else // sa is a process agent within a process agent and variables are in agent sa.stateAgent
            sa.stateAgent.state := initAgentState(sa.stateAgent.state,
                sid, sa.owner.owner.topStateId, atd.collectAllVariableDefinitions)
        endif
    endextend

    where

        collectAllVariableDefinitions(atd: Agent-type-definition): Variable-definition-set =def
            if atd.s-Identifier = undefined
            then    ∅ // no inherited variables
            else
                atd.s-Identifier.refersto1.collectAllVariableDefinitions
            endif
            ∪ // add local variables to set
            atd.s-Variable-definition-set
    endwhere
```

The outermost scope is associated with the top-level state node of an agent. It is created together with that state node. In case of nested process agents, the scopes of contained agents are added to the scope of the outermost process agent.

```
CREATECOMPOSITESTATEVARIABLES(sa:SDLAGENT, sn:STATENODE,
    cstd:Composite-state-type-definition) ≡
    extend STATEID with sid
        sn.stateId := sid
        sa.stateAgent.state := initAgentState(sa.stateAgent.state,
            sid,
            if sn.parentStateNode ≠ undefined then sn.parentStateNode.stateId else undefined endif,
             cstd.collectAllVariableDefinitions1) // initAgentState
    endextend

    where

        collectAllVariableDefinitions1(cstd: Composite-state-type-definition): Variable-definition-set =def
            if cstd.s-Identifier = undefined  // no inheritance
            then    ∅ // no inherited variables
            else // inherited variables
                cstd.s-Identifier.refersto1.collectAllVariableDefinitions1
            endif
            ∪ // add local variables to set
            cstd.s-Variable-definition-set
    endwhere
```

With each composite state, a new scope is associated, which is located below the scope of the parent state node.

```
CREATEPROCEDUREVARIABLES
    (sa:SDLAGENT, sn:STATENODE, pd:Procedure-definition, ap:CALLPARAM*) ≡
  extend STATEID with sid
    sn.stateId := sid
    sa.stateAgent.state :=
      initProcedureState(sa.stateAgent.state, // State
        sid, // new state id
        sn.parentStateNode.stateId, // state id
        pd.collectAllVariableDefinitions2, // variables
        pd.collectAllProcedureFPars, // formal parameter list
        ap // list of actual parameter values to assign to parameters
      ) // initProcedureState
  endextend

  where

    collectAllVariableDefinitions2(pd: Procedure-definition): Variable-definition-set =def
      if pd.s-Identifier = undefined // no inheritance
      then ∅ // no inherited variables
      else pd.s-Identifier.refersto1.collectAllVariableDefinitions2 // inherited variables
      endif
      ∪ // add local variables to set
      pd.s-Variable-definition-set // local variables

    collectAllProcedureFPars(pd:Procedure-definition): PROCPARAM* =def
      if pd.s-Identifier = undefined // no inheritance
      then empty // no inherited formal parameters
      else pd.s-Identifier.refersto1.collectAllProcedureFPars // inherited formal parameters
      endif
      ⌢ // add local formal parameters
      pd.s-PROCPARAM-seq
  endwhere
```

With each procedure state, a new scope is associated, which is located below the scope of the parent state node.

```
CREATECOMPOUNDNODEVARIABLES(sa:SDLAGENT, scope: SCOPE) ≡
  extend STATEID with sid
    sa.currentStateId := sid
    scopeName(Self, sid) := scope.s-Name
    scopeContinueLabel(Self, sid) := scope.s-CONTINUELABEL
    scopeStepLabel(Self, sid) := scope.s-STEPLABEL
    sa.stateAgent.state := initAgentState(sa.stateAgent.state, sid,
      sa.currentStateId, scope.s-Variable-definition-set)
  endextend
```

With each compound node, a new scope is associated, which is located below the current scope.

F3.2.3.1.9 State machine creation and initialization

The behaviour of an SDL-2010 agent is given by a state machine, which may be omitted if the agent is passive. This state machine is modelled as a "hierarchical inheritance graph", which is unfolded recursively.

```
CREATESTATEMACHINE(smd:[State-machine]) ≡
  if smd.s-Identifier.refersto1.s-Abstract ≠ undefined
  then  raise(InvalidCall) // attempt to instantiate state machine with abstract composite state type
  elseif CREATETOPSTATEPARTITION(smd)
  endif
```

When an SDL-2010 agent is created, the macro CREATESTATEMACHINE is applied with the effect that the root node (*topStateNode*) of the "hierarchical inheritance state graph" is created. If the SDL-2010 agent has behaviour, the root node is refined (and possibly specialized) subsequently. If the agent is passive, no refinement is made. The unfolding of the graph is treated by the macro INITSTATEMACHINE.

If an SDL-2010 agent has behaviour, a "hierarchical inheritance state graph" modelling the agent's state machine is built, node-by-node. This graph forms the basis for entering and leaving states, and for selecting transitions. Inheritance is taken into account during execution, and is not handled by transformations. The unfolding of the graph is controlled by the following macro.

```
INITSTATEMACHINE ≡
    if Self.stateNodesToBeCreated ≠ ∅ then
        CREATESTATENODE
    elseif Self.statePartitionsToBeCreated ≠ ∅ then
        CREATESTATEPARTITION
    elseif Self.stateNodesToBeSpecialised ≠ ∅ then      // these are composite states!
        CREATEINHERITEDSTATE
    elseif Self.stateNodesToBeRefined ≠ ∅ then
        CREATESTATEREFINEMENT
    else
        Self.agentMode2 := initialisationFinished
    endif
```

Nodes to be created are kept in the agent's state components *stateNodesToBeCreated*, *statePartitionsToBeCreated*, *stateNodesToBeSpecialised*, and *stateNodesToBeRefined*, and are treated in that order. Unfolding of the graph updates these state components and ends with the graph being completed, i.e., no further nodes to be created.

F3.2.3.1.10 Procedure graph creation and initialization

The behaviour of a procedure is given by a procedure graph. This procedure graph is modelled as a "hierarchical inheritance graph", which is unfolded recursively.

```
CREATEPROCEDUREGRAPH
    (pd:Procedure-definition,ap:CALLPARAM*,rl:[VALUELABEL], rRVI:[Identifier], cl:CONTINUELABEL) ≡
    CREATEPROCEDURESTATENODE(pd, ap, rl, rRVI, cl)
```

When a procedure is called, the macro CREATEPROCEDUREGRAPH is applied with the effect that the root node of the "hierarchical inheritance state graph" modelling the procedure is created. The unfolding of the graph is treated by the macro INITPROCEDUREGRAPH.

```
INITPROCEDUREGRAPH ≡
    if Self.stateNodesToBeCreated ≠ ∅ then
        CREATESTATENODE
    elseif Self.statePartitionsToBeCreated ≠ ∅ then
        CREATESTATEPARTITION
    elseif Self.stateNodesToBeSpecialised ≠ ∅ then      // these are composite states!
        CREATEINHERITEDSTATE
    elseif Self.stateNodesToBeRefined ≠ ∅ then
        CREATESTATEREFINEMENT
    else
        Self.agentMode4 := initialisationFinished
    endif
```

Nodes to be created are kept in the agent's state components *stateNodesToBeCreated*, *statePartitionsToBeCreated*, *stateNodesToBeSpecialised* and *stateNodesToBeRefined*, and are treated in that order. Unfolding of the graph updates these state components and ends with the graph being completed, i.e., no further nodes to be created.

F3.2.3.1.11 State node creation

The creation of state nodes is modelled by extending the controlled domain *STATENODE*. A macro is defined to handle the creation of state nodes. State partitions are also modelled as elements of the domain *STATENODE*, in clause F.3.2.3.1.12.

```
CREATESTATENODE ≡
choose snd: snd ∈ Self.stateNodesToBeCreated
    Self.stateNodesToBeCreated := Self.stateNodesToBeCreated \ {snd}
    extend STATENODE with sn
        sn.stateAS1 := snd // used, e.g., as argument for startLabel
        sn.owner := Self
        sn.parentStateNode := Self.currentParentStateNode
        sn.stateNodeKind := stateNode
        sn.stateName := snd.s-Name // State-name of State-node to be created
        sn.stateTransitions := snd.getStateTransitions
        sn.startTransitions := ∅ // updated if the state node is refined
        if snd.s-Identifier ≠ undefined then // inheritance
            Self.stateNodesToBeRefined := Self.stateNodesToBeRefined ∪ {sn}
            Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised ∪ {sn}
            sn.stateDefinitionAS1 := snd.s-Identifier.refersto1 // Composite-state-type-definition parent
        endif
    endextend
endchoose

where

getStateTransitions(s: State-node): SEMTRANSITION-set =def
    // transitions for input node set
{ mk-SEMTRANSITION(sigid, // Signal-identifier
    gateid, // optional Gate-identifier for via gate
    if provided = undefined // Provided-expression undefined
    then   undefined
    else
        provided.startLabel // Provided-expression
    endif,
    pri, // Priority-name
    transition.startLabel,
    undefined // no state exit point
    ) // SemTransition
|   i(pri, sigid, gateid, provided, *, *, transition)  ∈ s.s-Input-node-set
}
∪  // transitions for spontaneous transitions
{ mk-SEMTRANSITION(NONE,
    undefined, // not relevant  - Gate-identifier for via gate for Input-node
    if provided = undefined // Provided-expression undefined
    then   undefined
    else
        provided.startLabel // Provided-expression
    endif,
    undefined,
    transition.startLabel,
    undefined // no state exit point
    ) // SemTransition
|   sp(provided, transition) ∈ s.s-Spontaneous-transition-set
}
∪  // transitions for continuous signals
{ mk-SEMTRANSITION(NONE, // no signal
    undefined, // not relevant  - Gate-identifier for via gate for Input-node
    ce.startLabel, // Continuous-expression
    pri, // Priority-name
    transition.startLabel,
```

```
    undefined // no state exit point
    ) // SemTransition
| c(ce, pri, transition) ∈ s.s-Continuous-signal-set }
∪  // transitions for connect nodes
{ mk-SEMTRANSITION(NONE, // no signal
    undefined, // not relevant  - Gate-identifier for via gate for Input-node
    undefined, // no provided expression
    undefined, // no priority
    transition.startLabel, // transition for state exit
    if exitname = undefined then DEFAULT else exitname endif // State-exit-point-name
    ) // SemTransition
| c(exitname, transition) ∈ s.s-Connect-node-set
}
∪  // transition for state timer
if      s.s-State-timer ≠ undefined
    ∧  (∃ tmi(pid, tid, elist) ∈ TIMERINST):
            (       pid = Self.selfPid
                ∧   tid = s.s-State-timer.s-Identifier // timer Identifier
                ∧   elist = s.s-State-timer.s-EXPRESSION-seq
            ) // current pid, same id and same expression list
    ∧  active(tmi)
then
    { mk-SEMTRANSITION(s.s-State-timer.s-Identifier, // timer signal
        undefined, // not relevant  - Gate-identifier for via gate for Input-node
        undefined, // no provided expression for state timer
        undefined, // no priority state timer
        s.s-State-timer.s-Transition.startLabel, // transition for state timer
        undefined // no state exit point
        ) // SemTransition
    }
else ∅
endif
endwhere
```

State nodes are created as part of a state transition graph, which is unfolded node by node. The nodes to be created are kept in the agent's state component *stateNodesToBeCreated*. If that set is not empty, this means that the unfolding of a state transition graph is currently in progress, and some element of the set is chosen. When a state node is created, its bookkeeping information is initialized. Since being a regular state node, the created state node may have a substructure; it is included in the set of state nodes to be refined.

The local function *getStateTransitions* gives the set of *SEMTRANSITION* items for the state node. For input nodes, spontaneous inputs and continuous signals, the first *LABEL* of a *SEMTRANSITION* item is the start label for the provided/continuous expression, which is evaluated at execution time through SELINPUTEVALUATIONPHASE in clause F3.2.3.2.10 Input selection.

```
CREATEPROCEDURESTATENODE
    (pd:Procedure-definition, ap:CALLPARAM*, rl:[VALUELABEL], rRVI:[Identifier], cl:CONTINUELABEL) ≡
    extend STATENODE with sn
        sn.procedureAS1 := pd
        sn.owner := Self
        sn.parentStateNode := Self.currentParentStateNode
        sn.stateNodeKind := procedureNode
        sn.stateName := undefined
        sn.stateTransitions := ∅
        sn.startTransitions := ∅          // updated if the state node is refined
        sn.resultLabel := rl
        sn.resultRefVarId := rRVI
        Self.stateNodesToBeRefined := {sn}
        Self.stateNodesToBeCreated := ∅
        Self.statePartitionsToBeCreated := ∅
```

```
        Self.stateNodesToBeSpecialised := {sn}
        Self.currentProcedureStateNode := sn
        Self.callingProcedureNode := sn
        CREATEPROCEDUREVARIABLES(Self,sn,pd,ap)
        SAVEPROCEDURECONTROLBLOCK(sn,cl)
    endextend
```

Procedure state nodes are the top-level nodes of a procedure graph, which is unfolded node by node subsequently. These nodes are created dynamically, when a procedure call is made. Thus, recursive procedure calls can be handled in a uniform way.

```
SAVEPROCEDURECONTROLBLOCK(sn:STATENODE, cl:CONTINUELABEL) ≡
        sn.agentMode1 := Self.agentMode1
        sn.agentMode2 := Self.agentMode2
        sn.agentMode3 := Self.agentMode3
        sn.agentMode4 := Self.agentMode4
        sn.agentMode5 := Self.agentMode5
        sn.currentStateId := Self.currentStateId
        sn.currentLabel := Self.currentLabel
        sn.continueLabel := cl
        sn.currentParentStateNode := Self.currentParentStateNode
        sn.previousStateNode := Self.previousStateNode
        sn.callingProcedureNode := Self.callingProcedureNode
```

Information on procedure control blocks is given in clause F3.2.1.2.3.

F3.2.3.1.12 State partition creation

The creation of state partitions is modelled by extending the controlled domain *STATENODE*. Several macros are defined to handle the creation of various kinds of state partitions, namely the top state partition, (regular) state partitions, and state partitions introduced to model inheritance.

```
CREATETOPSTATEPARTITION(smd:[State-machine]) ≡
    extend STATENODE with sn
        sn.owner := Self
        Self.topStateNode := sn
        sn.parentStateNode := undefined
        sn.stateNodeKind := statePartition
        sn.stateTransitions := ∅
        sn.startTransitions := ∅ // updated if the state partition is refined
        if smd ≠ undefined then
            sn.stateDefinitionAS1 := smd.s-Identifier.refersto1
            sn.stateName := smd.s-Name // State-name of state machine
            Self.stateNodesToBeRefined := {sn}
            Self.stateNodesToBeSpecialised := {sn}
        else
            sn.stateName := undefined
            Self.stateNodesToBeRefined := ∅
            Self.stateNodesToBeSpecialised := ∅
        endif
        Self.stateNodesToBeCreated := ∅
        Self.statePartitionsToBeCreated := ∅
    endextend
```

The unfolding of the "hierarchical inheritance state graph" modelling an agent's state machine starts with the creation of the root node, as defined by the macro CREATETOPSTATEPARTITION. When a root node is created, its bookkeeping information is initialized. In particular, the root node is classified as a state partition. If the agent has behaviour, the root node has a substructure, and is therefore included in the set of state nodes to be refined. Further state components of the agent are reset before starting the unfolding of the graph.

```
CREATESTATEPARTITION ≡
    choose spd: spd ∈ Self.statePartitionsToBeCreated
        Self.statePartitionsToBeCreated := Self.statePartitionsToBeCreated \ {spd}
        extend STATENODE with sn
            sn.partitionAS1 := spd // used, e.g., as argument for startLabel
            sn.owner := Self
            sn.parentStateNode := Self.currentParentStateNode
            sn.stateNodeKind := statePartition
            sn.stateName := spd.s-Name
            sn.stateTransitions := ∅
            sn.startTransitions := ∅                    // updated if the state partition is refined
            ∀ (cd ∈ spd.s-(Entry-connection-definition ∪ Exit-connection-definition)-set):
                (if cd ∈ Entry-connection-definition
                then entryConnection(cd.s-Outer-entry-point.adaptEntryPoint, sn) :=
                    adaptEntryPoint(cd.s-Inner-entry-point)
                else // cd is Exit-connection-definition
                    exitConnection(cd.s-Inner-exit-point, sn) := cd.s-Outer-exit-point
                endif)
            Self.currentParentStateNode.statePartitionSet :=
            Self.currentParentStateNode.statePartitionSet ∪ {sn}
            Self.stateNodesToBeRefined := Self.stateNodesToBeRefined ∪ {sn}
            Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised ∪ {sn}
        endextend
    endchoose

    where

        adaptEntryPoint(entry: Name ∪ DEFAULT): STATEENTRYPOINT =def
            if entry = DEFAULT then undefined else entry endif
    endwhere
```

(Regular) state partitions are created as part of a state aggregation node, which is unfolded node by node. The partitions to be created are kept in the agent's state component *statePartitionsToBeCreated*. If that set is not empty, this means that the unfolding of a state aggregation node is currently in progress, and some element of the set is chosen. When a state partition is created, its bookkeeping information is initialized. Modelling a state partition, the created state node may have a substructure, and is therefore included in the set of state nodes to be refined.

```
CREATEINHERITEDSTATE ≡
    choose sns: sns ∈ Self.stateNodesToBeSpecialised
        Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised \ {sns}
        let cstd = sns.stateDefinitionAS1 in
            if cstd.s-Identifier ≠ undefined // the composite state type inherits from another composite state
type
            then
                let parent = cstd.s-Identifier.refersto1 in
                if cstd.s-Identifier.refersto1.s-Abstract = undefined // cannot instantiate – it is abstract
                then   raise(InvalidCall) // attempt to instantiate abstract composite state type
                else
                    extend STATENODE with sn
                        sn.stateDefinitionAS1 := parent
                        sn.owner := Self
                        sn.parentStateNode := sns.parentStateNode
                        sn.stateNodeKind := sns.stateNodeKind
                        sn.stateName := sns.stateName
                        sn.stateTransitions := ∅
                        sn.startTransitions := ∅ // updated if the state node is refined
                        sns.inheritedStateNode := sn
                        Self.stateNodesToBeRefined := Self.stateNodesToBeRefined ∪ {sn}
                        Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised ∪ {sn}
```

```
            endextend
        endif // check if inherited composite state type abstract
        endlet // parent
    else
        sns.inheritedStateNode := undefined
    endif
  endlet
endchoose
```

Specialization of composite state types is modelled by adding another dimension to the hierarchical state graph, yielding a "hierarchical *inheritance* state graph". Formally, specialization is a relation between composite state *types*. In the state graph, it is modelled by an inheritance relation among state node *instances*. More specifically, if a state node is refined, and the refinement is defined using specialization, then a root node that is inherited by the refined state node, and has the composite state type being specialized, is created. By adding the root node to the set of state nodes to be refined, a "hierarchical inheritance state graph" modelling the specialization is subsequently attached to this root node.

F3.2.3.1.13 Composite state creation

All (regular) state nodes, state partitions, and procedure nodes are candidates for refinement and, if refined, for specialization. Refinements are defined by a composite state type, which includes another composite state type in case of specialization. In this clause, several macros treating these aspects are introduced.

```
CREATESTATEREFINEMENT ≡
  choose snr: snr ∈ Self.stateNodesToBeRefined
    Self.stateNodesToBeRefined := Self.stateNodesToBeRefined \ {snr}
    Self.currentParentStateNode := snr
    if snr.stateNodeKind = procedureNode
    then
      CREATEPROCEDUREVARIABLES(Self, snr, snr.procedureAS1, empty) // use parent node params
      CREATEPROCEDUREGRAPHNODES(snr, snr.procedureAS1.s-Procedure-graph)
    else
      let parent: Composite-state-type-definition = snr.stateDefinitionAS1 in
      CREATECOMPOSITESTATEVARIABLES(Self, snr, parent)
      CREATECOMPOSITESTATE(snr, parent)
      endlet
    endif
  endchoose
```

When a state node, state partition, or procedure node is created, it is added to a set of state nodes to be refined. In the macro CREATESTATEREFINEMENT, an arbitrary element of this set is selected, and it is checked whether a refinement applies. Refinements are then treated by the macro CREATECOMPOSITESTATE.

```
CREATECOMPOSITESTATE(sn:STATENODE, cstd:Composite-state-type-definition) ≡
  if cstd.s-Abstract ≠ undefined
  then  raise(InvalidCall) // attempt to instantiate abstract composite state type
  elseif
    case sr = cstd.s-implicit of
    | Composite-state-graph then CREATECOMPOSITESTATEGRAPH(sn,sr)
    | State-aggregation-node then CREATESTATEAGGREGATIONNODE(sn,sr)
    endcase
  endif // check for abstract Composite-state-type-definition
```

If a state is structured, it is refined into either a composite state graph or a state aggregation node. Based on this distinction, further rule macros are applied.

```
CREATECOMPOSITESTATEGRAPH(psn:STATENODE, csgd:Composite-state-graph) ≡
  psn.stateNodeRefinement := compositeStateGraph
```

psn.startTransitions := *getStartTransitions*({*csgd*.**s**-*State-transition-graph*.**s**-*State-start-node*})
∪ *getStartTransitions*(*csgd*.**s**-*Named-start-node*-**set**)
psn.freeActions := *getFreeActions*(*csgd*.**s**-*State-transition-graph*.**s**-*Free-action*-**set**)
CREATESTATETRANSITIONGRAPH(*psn*,*csgd*.**s**-*State-transition-graph*.**s**-*State-node*-**set**)

Creating a composite state graph means creating its state transition graph.

CREATESTATETRANSITIONGRAPH(*psn*:*STATENODE*, *nodes*: *State-node*-**set**) ≡
Self.stateNodesToBeCreated := *nodes*
Self.currentParentStateNode := *psn*

Creating a state transition graph means creating its state nodes. Creation of state nodes is performed in a series of subsequent ASM steps. These steps are triggered by assigning the state node definitions to the agent's state component *stateNodesToBeCreated*.

CREATEPROCEDUREGRAPHNODES(*psn*:*STATENODE*, *pg*:*Procedure-graph*) ≡
psn.stateNodeRefinement := *compositeStateGraph*
psn.startTransitions := *getStartTransitions*({*pg*.**s**-*Procedure-start-node*})
psn.freeActions := *getFreeActions*(*pg*.**s**-*Free-action*-**set**)
CREATESTATETRANSITIONGRAPH(*psn*, *pg*.**s**-*State-node*-**set**)
Self.stateNodesToBeCreated := *pg*.**s**-*State-node*-**set**
Self.currentParentStateNode := *psn*

Creating a procedure graph means creating its state nodes.

CREATESTATEAGGREGATIONNODE(*psn*:*STATENODE*, *sand*:*State-aggregation-node*) ≡
psn.stateNodeRefinement := *stateAggregationNode*
Self.statePartitionsToBeCreated := *sand*.**s**-*State-partition*-**seq**.*toSet*
Self.currentParentStateNode := *psn*
psn.statePartitionSet := ∅

Creating a state aggregation node means creating its state partitions, which is performed in a series of subsequent ASM steps. These steps are triggered by assigning the state partition definitions to the agent's state component *statePartitionsToBeCreated*.

The function *getStartTransitions* determines the start transitions for a start node or set of state nodes.

getStartTransitions(*s*: (*State-start-node* ∪ *Named-start-node* ∪ *Procedure-start-node*)-**set**):
STARTTRANSITION-**set** =$_{\text{def}}$
{ **if** *sn* ∈ *State-start-node*
then **mk**-*STARTTRANSITION*(*sn*.**s**-*Transition.startLabel*, *undefined*)
elseif *sn* ∈*Named-start-node*
then **mk**-*STARTTRANSITION*(*nsn*.**s**-*Transition.startLabel*, *nsn*.**s**-*Name*)
else // sn is a Procedure-start-node
mk-*STARTTRANSITION*(*sn*.**s**-*Transition.startLabel*, *undefined*)
endif | *sn* ∈ *s* }

The function *getFreeActions* for a set of *Free-action* items delivers a *FREEACTION* set, where for each *Free-action* items, there is *FREEACTION* (name and start label) item.

getFreeActions(*actions*: *Free-action*-**set**): *FREEACTION*-**set** =$_{\text{def}}$
{ **mk**-*FREEACTION*(*f*.**s**-*Name*, *f*.**s**-*Transition.startLabel*) | *f* ∈ *actions* }

F3.2.3.2 System execution

After initialization, SDL-2010 agents start their execution. The execution of the system is modelled by the concurrent execution of all its agents.

F3.2.3.2.1 Agent set execution

EXECAGENTSET ≡
let *child* = *take*({*ag* ∈ *SDLAGENT*: *ag.owner* = *Self* ∧ *ag.agentMode1* = *initialisation*}) in

```
        if child = undefined then // wait until all child agents have left initialisation mode
            DELIVERSIGNALS // then deliver signals
        endif
    endlet
```

As defined in clause 9 *Semantics* of [ITU-T Z.101]: The set of retained signals is ordered in the queue for delivery according to their availability time, which for each signal that does not convey an availability time is the same as its arrival time (that is, the signal is available as soon as it arrives).

Only signals that have arrived (that is, in the a queue for an in gate) are considered, and if the arrival time is later than or equal to the availability time, this is used for ordering the signals transferred to the input port. When a signal is in the input port of an agent instance, it is available for consumption only if the availability time is less than or equal to the current time.

```
DELIVERSIGNALS ≡
  choose g: g ∈ Self.ingates ∧ g.queue ≠ empty // g is the GATE at the end of the communication path
    let si = g.queue.head in // take the earliest signal instance for which now >= si.arrival
    let sortOrderTime = if si.arrival < si.availabilityTime then si.availabilityTime else si.arrival endif
    DELETE(si,g) // remove the signal instance from g, the arrival gate
    si.arrivalGate := g // the arrival gate for the signal instance – needed for via gate inputs
    if si.toArg ≠ undefined ∧ si.toArg ∈ PID then // deliver to agent with this Pid
      choose sa: sa ∈ SDLAGENT ∧ sa.owner = Self ∧ sa.selfPid = si.toArg
        INSERT(si, sortOrderTime, sa.inport)
      endchoose
    else // deliver to any agent instance of Self
      choose sa: sa ∈ SDLAGENT ∧ sa.owner = Self
        INSERT(si, sortOrderTime, sa.inport)
      endchoose
    endif
    endlet // sortOrderTime
    endlet // si
  endchoose
```

F3.2.3.2.2 Agent execution

The execution of SDL-2010 agents is modelled by a start phase followed by alternating phases, namely transition selection and transition firing. To distinguish between these phases, corresponding agent modes are defined. When in agent mode *selectingTransition* (*agentMode2*), the agent attempts to select a transition, obeying a number of constraints. In agent mode *firingTransition*, a previously selected transition is fired.

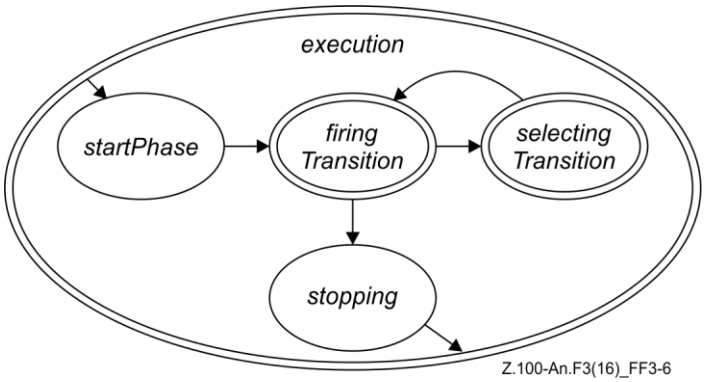

Figure F3-6 – Activity phases of SDL-2010 agents: execution (level 2)

An agent reaches the execution phase after it has completed its initialization. The execution phase consists of three sub-phases as shown in the state diagram. Two of these sub-phases will in turn be refined, which is indicated by the double line.

```
EXECAGENT ≡
    if Self.agentMode2 = startPhase then
        EXECUTIONSTARTPHASE
    elseif Self.agentMode2 = firingTransition then
        FIRETRANSITION
    elseif Self.agentMode2 = selectingTransition then
        SELECTTRANSITION
    elseif Self.agentMode2 = stopping then
        STOPPHASE
    endif
```

The execution of agents is given by the rule macro EXECAGENT. Depending on the current agent mode, the corresponding execution phases are selected.

```
GETEXECRIGHT ≡
    if Self.stateAgent.isActive = undefined then
        Self.stateAgent.isActive := Self
    endif

RETURNEXECRIGHT ≡
    Self.stateAgent.isActive := undefined

ExecRightPresent(sa:SDLAGENT): BOOLEAN =def
    let myDef: Agent-type-definition = sa.owner.agentAS1.s-Identifier.refersto1 in
    sa.stateAgent.isActive = sa ∨ myDef.s-Agent-kind ∈ {BLOCK, SYSTEM}
    endlet
```

F3.2.3.2.3 Starting agent execution

When the execution phase starts, several initializations are made: the set of state nodes to be entered is initialized to consist of the top state node; furthermore, the execution is switched to entering state nodes.

```
EXECUTIONSTARTPHASE ≡
    Self.isActive := undefined
    Self.stateNodesToBeEntered :=
        {mk-STATENODEWITHENTRYPOINT (Self.topStateNode,undefined)}
    Self.agentMode2 := firingTransition
    Self.agentMode3 := enteringStateNode
    Self.agentMode4 := startPhase
    Self.currentLabel := undefined
```

F3.2.3.2.4 Transition selection

In agent mode *selectingTransition* (*agentMode2*), an SDL-2010 agent searches for a fireable transition. SDL-2010 imposes certain rules on the search order. For instance, priority input signals have to be checked before ordinary input signals, and these have in turn to be checked before continuous signals can be consumed. Furthermore, a transition emanating from a substate has higher priority than a conflicting transition emanating from any of the containing states. Finally, redefined transitions take precedence over conflicting inherited transitions. These and some more constraints have to be observed when formalizing the transition selection.

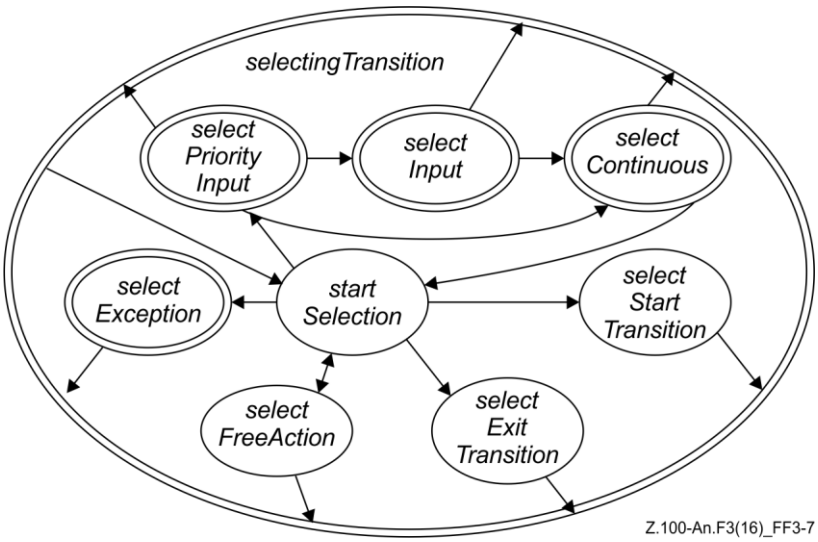

Figure F3-7 – Activity phases of SDL-2010 agents: selecting transition (level 3)

In order to structure the transition selection, several agent mode levels are defined. The uppermost level is shown in the diagram, where the agent mode *selectingTransition* is refined into four sub-modes (*agentMode3*). Some of these sub-modes will in turn be refined later.

```
SELECTTRANSITION ≡
    if Self.agentMode3 = startSelection then
        SELECTTRANSITIONSTARTPHASE
    elseif Self.agentMode3 = selectStartTransition then
        SELECTSTARTTRANSITION
    elseif Self.agentMode3 = selectExitTransition then
        SELECTEXITTRANSITION
    elseif Self.agentMode3 = selectFreeAction then
        SELECTFREEACTION
    elseif Self.agentMode3 = selectPriorityInput then
        SELECTPRIORITYINPUT
    elseif Self.agentMode3 = selectInput then
        SELECTINPUT
    elseif Self.agentMode3 = selectContinuous then
        SELECTCONTINUOUS
    endif
```

Transition selection starts with an attempt to select a start transition, free action, priority input, an ordinary input, and finally, a continuous signal (in that order). If no transition has been selected, the selection is repeated/aborted. The evaluation of provided expressions and continuous expressions may alter the local state of the agent, which may lead to different results depending on the evaluation order.

```
TRANSITIONFOUND(t:SEMTRANSITION) ≡
    Self.currentParentStateNode := Self.stateNodeChecked.parentStateNode
    Self.previousStateNode := Self.stateNodeChecked
    Self.currentStateId := Self.stateNodeChecked.parentStateNode.stateId
    Self.currentLabel := t.s2-LABEL // second label
    Self.agentMode2 := firingTransition
    Self.agentMode3 := firingAction
    RETURNEXECRIGHT
```

As soon as a selectable transition is found, the start label of the transition is assigned, and the agent modes are set to *firingTransition* and *firingAction*, respectively. Also, the current parent state node is set, which determines the current state name scope. This scope information is used when an *ENTERSTATENODE*-primitive is evaluated.

STARTTRANSITIONFOUND(*t*:*STARTTRANSITION*, *psn*:*STATENODE*) ≡
Self.currentParentStateNode := *psn*
Self.currentStateId := *psn.stateId*
Self.currentLabel := *t*.**s**-*LABEL*
Self.agentMode2 := *firingTransition*
Self.agentMode3 := *firingAction*
RETURNEXECRIGHT

As soon as a selectable start transition is found, the start label of the transition is assigned, and the agent modes are set to *firingTransition* and *firingAction*, respectively. Also, the current parent state node is set, which determines the current state name scope. This scope information is used when an *ENTERSTATENODE*-primitive is evaluated.

EXITTRANSITIONFOUND(*et*:*SEMTRANSITION*, *psn*:*STATENODE*) ≡
Self.currentParentStateNode := *psn*
Self.currentStateId := *psn.stateId*
Self.currentLabel := *et*.**s2**-*LABEL*
Self.agentMode2 := *firingTransition*
Self.agentMode3 := *firingAction*
RETURNEXECRIGHT

As soon as a selectable exit transition is found, the start label of the transition is assigned, and the agent modes are set to *firingTransition* and *firingAction*, respectively. Also, the current parent state node is set, which determines the current state name scope. This scope information is used when a *LEAVESTATENODE*-primitive is evaluated.

FREEACTIONFOUND(*fa*:*FREEACTION*, *psn*:*STATENODE*) ≡
Self.currentParentStateNode := *psn*
Self.currentStateId := *psn.stateId*
Self.currentLabel := *fa*.**s**-*LABEL*
Self.agentMode2 := *firingTransition*
Self.agentMode3 := *firingAction*
RETURNEXECRIGHT

As soon as a free action is found, the start label of the transition is assigned, and the agent modes are set to *firingTransition* and *firingAction*, respectively. Also, the current parent state node is set, which determines the current state name scope.

F3.2.3.2.5 Starting selection of transitions

When the selection of a transition starts, several initializations are made: the input port is "frozen", meaning that its state at the beginning of the selection is the basis for this selection cycle. This does not prevent signal instances to arrive while the selection is active; however, these signals will not be considered before the next selection cycle. Furthermore, the selection is switched to checking priority signals.

SELECTTRANSITIONSTARTPHASE ≡
if *Self.currentStartNodes* ≠ ∅ **then**
Self.stateNodeChecked := *undefined*
Self.agentMode3 := *selectStartTransition*
elseif *Self.currentExitStateNodes* ≠ ∅ **then**
Self.stateNodeChecked := *undefined*
Self.agentMode3 := *selectExitTransition*
elseif *Self.currentConnector* ≠ *undefined* **then**
Self.agentMode3 := *selectFreeAction*
else
Self.inputPortChecked := *Self.inport.queue*
Self.agentMode3 := *selectPriorityInput*
Self.agentMode4 := *startPhase*
endif

F3.2.3.2.6 Start transition selection

Selection of a start transition is performed by checking, for all current start nodes, whether a start transition can be selected.

```
SELECTSTARTTRANSITION ≡
    if Self.stateNodeChecked = undefined then
        let snwen = take(Self.currentStartNodes) in
            if snwen ≠ undefined then
                Self.currentStartNodes := Self.currentStartNodes \ {snwen}
                Self.startNodeChecked := snwen
                Self.stateNodeChecked := snwen.s-STATENODE
            endif
        endlet
    else
        let t = take({tr ∈ Self.stateNodeChecked.startTransitions:
            tr.s-STATEENTRYPOINT = Self.startNodeChecked.s-implicit}) in
            if t ≠ undefined then
                STARTTRANSITIONFOUND(t, Self.startNodeChecked.s-STATENODE)
            else
                Self.stateNodeChecked :=
                    take({sn1 ∈ Self.stateNodesToBeChecked:
                            directlyInheritsFrom(Self.stateNodeChecked,sn1)})
            endif
        endlet
    endif
```

Start transitions are associated directly with the refined node, and are distinguished by their state entry point.

F3.2.3.2.7 Exit transition selection

```
SELECTEXITTRANSITION ≡
    let snwex = take(Self.currentExitStateNodes) in
    if Self.stateNodeChecked = undefined then
            if snwex ≠ undefined then
                Self.currentExitStateNodes := Self.currentExitStateNodes \ {snwex}
                Self.exitNodeChecked := snwex
                Self.stateNodeChecked := snwex.s-STATENODE
            endif
    else
        let t = take({tr ∈ Self.stateNodeChecked.stateTransitions.exitTransitions:
            tr.s-STATEEXITPOINT = Self.exitNodeChecked.s-STATEEXITPOINT}) in
            if t ≠ undefined then
                EXITTRANSITIONFOUND(t,snwex.s-STATENODE)
            else
                Self.stateNodeChecked :=
                    take({sn1 ∈ Self.stateNodesToBeChecked:
                            directlyInheritsFrom(Self.stateNodeChecked,sn1)})
            endif
        endlet
    endif
    endlet
```

Exit transitions are associated with the containing node, and are distinguished by their state exit point.

F3.2.3.2.8 Free action selection

```
SELECTFREEACTION ≡
        let fa = take({elem ∈ Self.stateNodeChecked.freeActions:
```

```
        elem.s-Name = Self.currentConnector.s-Name}) in
        if fa ≠ undefined then
            Self.currentConnector := undefined
            FREEACTIONFOUND(fa, Self.currentParentStateNode)
        else
            Self.stateNodeChecked :=
                take({sn1 ∈ Self.stateNodesToBeChecked:
                        directlyInheritsFrom(Self.stateNodeChecked,sn1)})
        endif
    endlet
```

Free actions are associated directly with the refined node, and are distinguished by their connector name.

F3.2.3.2.9 Priority input selection

Selection of a priority input is performed by checking, for each signal instance of the agent's input port, all current state nodes. Inheritance is taken into account by checking, for each state node, the inherited state nodes.

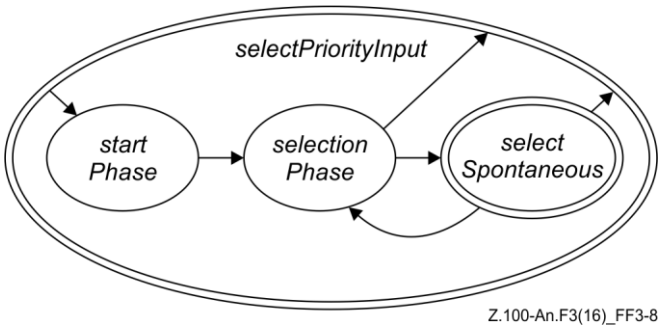

Figure F3-8 – Activity phases of SDL-2010 agents: selecting priority inputs (level 4)

The selection of a priority input consists of the sub-phases (*agentMode4*) shown in the diagram. At any time during the selection phase, an attempt to select a spontaneous signal may be made, depending on the value of the monitored predicate *Self.spontaneous*.

```
SELECTPRIORITYINPUT ≡
    if Self.agentMode4 = startPhase then
        SELPRIORITYINPUTSTARTPHASE
    elseif Self.agentMode4 = selectionPhase then
        SELPRIORITYINPUTSELECTIONPHASE
    elseif Self.agentMode4 = selectSpontaneous then
        SELECTSPONTANEOUS
    endif
```

This ASM macro defines the upper level control structure of the priority input selection. Depending on the agent mode *agentMode4*, further action is defined in the corresponding ASM macro. This control structure is part of the previous state diagram.

```
SELPRIORITYINPUTSTARTPHASE ≡
    if Self.inputPortChecked ≠ empty then
        Self.signalChecked := Self.inputPortChecked.head
        Self.SignalSaved := false
        Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)
        Self.stateNodeChecked := undefined
        Self.agentMode4 := selectionPhase
    else
        Self.agentMode3 := selectContinuous
        Self.agentMode4 := startPhase
```

```
        RETURNEXECRIGHT
    endif
```

When the selection starts, it is checked whether the input port carries signals. If so, several initializations are made: the first signal instance to be checked is determined, the state nodes to be checked are set, and the selection is activated. If the input port is empty, the selection of continuous signals is triggered.

```
SELPRIORITYINPUTSELECTIONPHASE ≡
    if Self.stateNodeChecked = undefined then
        NEXTSTATENODETOBECHECKED
    elseif Self.spontaneous then
        Self.agentMode4 := selectSpontaneous
        Self.agentMode5 := selectionPhase
    else
        let t = take({tr ∈ Self.stateNodeChecked.stateTransitions.priorityInputTransitions:
            tr.s-SIGNAL = Self.signalChecked.signalType}) in
            if t ≠ undefined then
                Self.currentSignalInst := Self.signalChecked
                Self.sender := Self.signalChecked.signalSender
                DELETE(Self.signalChecked, Self.inport)
                TRANSITIONFOUND(t)
            else
                Self.stateNodeChecked := undefined
            endif
        endlet
    endif

    where

        NEXTSTATENODETOBECHECKED ≡
            if Self.stateNodesToBeChecked ≠ ∅ ∧ ¬ Self.SignalSaved then
                SELECTNEXTSTATENODE
            else
                NEXTSIGNALTOBECHECKED
                Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)
                Self.stateNodeChecked := undefined
            endif

        SELECTNEXTSTATENODE ≡
            let sn = Self.stateNodesToBeChecked.selectNextStateNode in
                if sn = undefined then
                    UNDEFINEDBEHAVIOUR
                elseif sn.stateNodeKind = procedureNode then
                    Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \
                        collectCurrentSubStates(sn.getPreviousStatePartition)
                    // only state partitions of the state machine to be considered here
                elseif sn.stateNodeKind = statePartition then
                    Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}
                elseif sn.stateNodeKind = stateNode then
                    let curSigId: Identifier = Self.signalChecked.signalType in
                        Self.stateNodeChecked := sn
                        Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}
                        Self.transitionsToBeChecked :=
                            {t ∈ sn.stateTransitions.inputTransitions: t.s-SIGNAL = curSigId}
                        if curSigId ∈ // gate is ignored here, because via gate not valid for priority input
                            {   sig ∈ Identifier : sig.parentAS1 = sn.stateAS1.s-Save-signalset // Save-
signalset
                                ∧sig.refersto1 ∈ Signal-definition // signal rather than gate
                            } // is signal saved
                        then
                            Self.SignalSaved := true
```

```
                endif
            endlet // curSigId
        endif
    endlet // sn

NEXTSIGNALTOBECHECKED ≡
    let si = nextSignal(Self.signalChecked, Self.inputPortChecked) in
        if si ≠ undefined then
            Self.signalChecked := si
            Self.SignalSaved := false
        else
            Self.agentMode3 := selectInput
            Self.agentMode4 := startPhase
            RETURNEXECRIGHT
        endif
    endlet
endwhere
```

For a given signal instance in the input port, all current state nodes of the agent are checked in an arbitrary order, beginning, for each state partition, with the innermost state node. The latter reflects the priority among conflicting transitions. Furthermore, when a particular state node is being checked, the inherited state nodes are checked next, i.e., inheritance is taken into account at execution time and not handled by transformations. As a redefinition takes precedence over the redefined transition, the inherited nodes are to be checked only if the current signal instance is neither saved nor consumed in the current state.

If the given signal instance is not a priority input in the current states of the agent, the next signal instance of the input port is checked. This is repeated until either all signals have been checked, or a priority input has been found. In the former case, the selection of an input transition is triggered.

F3.2.3.2.10 Input selection

Selection of an input is performed by checking, for each signal instance of the agent's input port, all current state nodes until a signal instance satisfying certain conditions is found. If no such signal instance is found, the selection of a continuous signal is triggered.

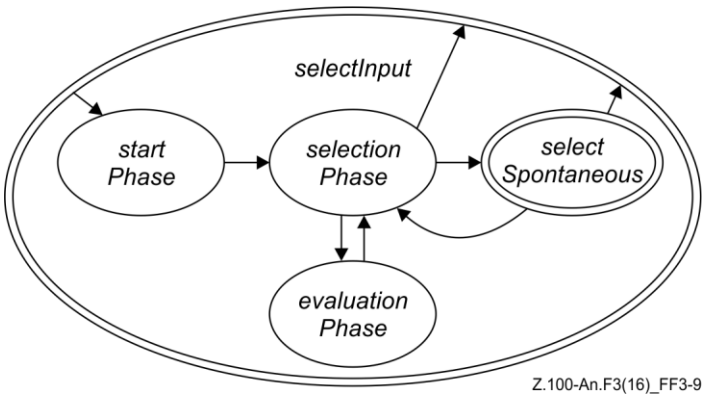

Figure F3-9 – Activity phases of SDL-2010 agents: selecting inputs (level 4)

The selection of an ordinary input consists of the sub-phases shown in the state diagram. In comparison to the selection of a priority input, an evaluation phase is added. This phase is entered when a provided expression has to be evaluated. At any time during the selection phase, an attempt to select a spontaneous signal may be made, depending on the value of the monitored predicate *Self.spontaneous*.

```
SELECTINPUT ≡
    if Self.agentMode4 = startPhase then
```

```
        SELINPUTSTARTPHASE
    elseif Self.agentMode4 = selectionPhase then
        SELINPUTSELECTIONPHASE
    elseif Self.agentMode4 = evaluationPhase then
        SELINPUTEVALUATIONPHASE
    elseif Self.agentMode4 = selectSpontaneous then
        SELECTSPONTANEOUS
    endif
```

The ASM macro SELECTINPUT defines the upper level control structure of the input selection. Depending on the agent mode *agentMode3*, further action is defined in the corresponding ASM macro. This control structure is part of the previous state diagram.

```
SELINPUTSTARTPHASE ≡
    if Self.inputPortChecked ≠ empty then
        Self.signalChecked := Self.inputPortChecked.head
        Self.SignalSaved := false
        Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)
        Self.stateNodeChecked := undefined
        Self.transitionsToBeChecked := ∅
        Self.agentMode4 := selectionPhase
    else
        Self.agentMode3 := selectContinuous
        Self.agentMode4 := startPhase
        RETURNEXECRIGHT
    endif
```

When the selection starts in SELINPUTSTARTPHASE, it is checked whether the input port contains signals. If so, several initializations are made: the first signal instance to be checked is determined, the state nodes to be checked are set, the transitions to be checked are reset, and the selection is activated. If the input port is empty, the selection of a continuous signal is triggered.

```
SELINPUTSELECTIONPHASE ≡
    if Self.stateNodeChecked = undefined then
        NEXTSTATENODETOBECHECKED1
    elseif Self.spontaneous then
        Self.agentMode4 := selectSpontaneous
        Self.agentMode5 := selectionPhase
    elseif Self.transitionsToBeChecked ≠ ∅ then
        choose t: t ∈ Self.transitionsToBeChecked
            Self.transitionsToBeChecked := Self.transitionsToBeChecked \ {t}
            if t.s1-LABEL ≠ undefined then
                EVALUATEENABLINGCONDITION(t)
            else
                Self.currentSignalInst := Self.signalChecked
                Self.sender := Self.signalChecked.signalSender
                DELETE(Self.signalChecked,Self.inport)
                TRANSITIONFOUND(t)
            endif
        endchoose
    else
        Self.stateNodeChecked := undefined
    endif

where

    EVALUATEENABLINGCONDITION(t:SEMTRANSITION) ≡
        Self.transitionChecked := t
        Self.currentStateId := Self.stateNodeChecked.parentStateNode.stateId
        Self.currentLabel := t.s1-LABEL
        Self.agentMode4 := evaluationPhase
```

```
NEXTSTATENODETOBECHECKED1 ≡
    if Self.stateNodesToBeChecked ≠ ∅ ∧ ¬ Self.SignalSaved then
        SELECTNEXTSTATENODE1
    else
        if ¬ Self.SignalSaved then   // implicit transition; different cf  NextStateNodeToBeChecked
            DELETE(Self.signalChecked,Self.inport)
        endif // different cf  NextStateNodeToBeChecked
        NEXTSIGNALTOBECHECKED1 // different cf  NextStateNodeToBeChecked
        Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)
        Self.stateNodeChecked := undefined
    endif

SELECTNEXTSTATENODE1 ≡
    let sn = Self.stateNodesToBeChecked.selectNextStateNode in
        if sn = undefined then
            UNDEFINEDBEHAVIOUR
        elseif sn.stateNodeKind = procedureNode then
            Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \
                collectCurrentSubStates(sn.getPreviousStatePartition)
            // less the current substates of the innermost state partition not belonging to a procedure
        elseif sn.stateNodeKind = statePartition then
            Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}
        elseif sn.stateNodeKind = stateNode then
            let curSigId: Identifier = Self.signalChecked.signalType in
                Self.stateNodeChecked := sn
                Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}
                Self.transitionsToBeChecked :=
                    {t ∈ sn.stateTransitions.inputTransitions: t.s-SIGNAL = curSigId
                    ∧ (t.s-GATE = undefined ∨ t.s-GATE = Self.signalChecked.arrivalGate)
                    }
                if  curSigId ∈
                    {       sig ∈ Identifier : sig.parentAS1 = sn.stateAS1.s-Save-signalset  // Save-signalset
                        ∧   sig.refersto1 ∈ Signal-definition // is signal in Save-signalset rather than gate
                        ∧   ( (∀ sv ∈ sig.parentAS1 : sv.s2-Identifier = undefined ) // no gate Save-item
                            ∨ Self.signalChecked.arrivalGate.gateAS1.identifier1 ∈ // sig+gate Save-item
                              { gid : sv ∈ sig.parentAS1 ∧ sig = sv.s1-Identifier ∧ gid = sv.s2-Identifier }
                            ) // either all Save-items without gate, or  sig+gate in same Save-item
                    } // signals saved in state
                then // is signal saved
                    Self.SignalSaved := true
                endif
            endlet // curSigId
        endif
    endlet

NEXTSIGNALTOBECHECKED1 ≡
    let si = nextSignal(Self.signalChecked,Self.inputPortChecked) in
        if si ≠ undefined then
            Self.signalChecked := si
            Self.SignalSaved := false
        else
            Self.agentMode3 := selectContinuous // Different cf NextSignalToBeChecked
            Self.agentMode4 := startPhase
            RETURNEXECRIGHT
        endif
    endlet
endwhere
```

For a given signal instance in the input port, all current state nodes of the agent are checked in an arbitrary order, beginning, for each state partition, with the innermost state node. The latter reflects the priority among conflicting transitions. Furthermore, when a particular state node is being checked, the inherited state nodes are checked next, i.e., inheritance is taken into account at

execution time and not handled by transformations. As a redefinition takes precedence over the redefined transition, the inherited nodes are to be checked only if the current signal instance is neither saved nor consumed in the current state.

If an input transition has a provided expression, this expression has to be evaluated before continuing with the selection.

If the given signal instance is saved in the current states of the agent, the next signal instance of the input port is checked. This is repeated until either all signals have been checked, or an input has been selected. In the former case, the selection of a continuous signal is triggered.

```
SELINPUTEVALUATIONPHASE ≡
    if Self.currentLabel ≠ undefined then
        choose b: b ∈ behaviour s ∧ b.s-LABEL = Self.currentLabel
            EVAL(b.s-ACTION)
        endchoose
    elseif semvalueBool(value(Self.transitionChecked.s1-LABEL,Self)) then // enabled
            Self.currentSignalInst := Self.signalChecked
            Self.sender := Self.signalChecked.signalSender
            DELETE(Self.signalChecked,Self.inport)
            TRANSITIONFOUND(Self.transitionChecked)
    else
        Self.agentMode4 := selectionPhase
    endif
```

As this evaluation consists of several actions in general, another agent mode, *evaluationPhase*, is entered. After completion of the evaluation, either the considered input signal is consumed, or the selection continues.

F3.2.3.2.11 Continuous signal selection

Selection of an input is performed by checking, for each signal instance of the agent's input port, all current state nodes until a signal instance satisfying certain conditions is found. If no such signal instance is found, this cycle of transition selection ends, and another cycle is started.

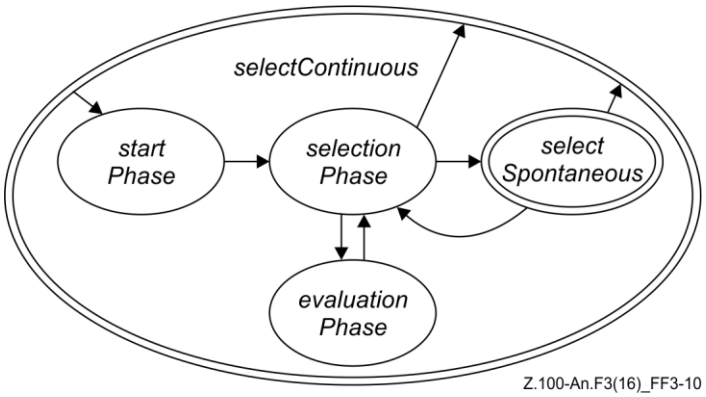

Figure F3-10 – Activity phases of SDL-2010 agents: selecting continuous signals (level 4)

The selection of a continuous signal consists of the sub-phases shown in the state diagram. The control is identical to the selection of an ordinary input.

```
SELECTCONTINUOUS ≡
    if Self.agentMode4 = startPhase then
        SELCONTINUOUSSTARTPHASE
    elseif Self.agentMode4 = selectionPhase then
        SELCONTINUOUSSELECTIONPHASE
    elseif Self.agentMode4 = evaluationPhase then
        SELCONTINUOUSEVALUATIONPHASE
```

```
elseif Self.agentMode4 = selectSpontaneous then
    SELECTSPONTANEOUS
endif
```

This ASM macro defines the upper level control structure of the continuous signal selection. Depending on the agent mode *agentMode4*, further action is defined in the corresponding ASM macro. This control structure is part of the previous state diagram.

```
SELCONTINUOUSSTARTPHASE ≡
    Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)
    Self.stateNodeChecked := undefined
    Self.transitionsToBeChecked := ∅
    Self.agentMode4 := selectionPhase
```

When the selection starts, several initializations are made: the state nodes to be checked are set, the transitions to be checked are reset, and the selection is activated.

```
SELCONTINUOUSSELECTIONPHASE ≡
    if Self.stateNodeChecked = undefined then
        NEXTSTATENODETOBECHECKED2
    elseif Self.spontaneous then
        Self.agentMode4 := selectSpontaneous
        Self.agentMode5 := selectionPhase
    else
        let t = selectContinuousSignal(Self.transitionsToBeChecked, Self.continuousPriorities) in
            if t ≠ undefined then
                Self.transitionsToBeChecked := Self.transitionsToBeChecked \ {t}
                if t.s1-LABEL ≠ undefined then
                    EVALUATEENABLINGCONDITION(t)
                else
                    TRANSITIONFOUND(t)
                endif
            else
                NEXTSTATENODETOBECHECKED2
            endif
        endlet
    endif

where
// EvaluateEnablingCondition defined in F3.2.3.2.10 Input selection (in "where" for SelInputSelectionPhase)

    NEXTSTATENODETOBECHECKED2 ≡
        if Self.stateNodesToBeChecked ≠ ∅ then
            if Self.stateNodeChecked = undefined then
                SELECTNEXTSTATENODE2
            else
                CHECKFORINHERITEDSTATENODES
            endif
        else
            Self.agentMode3 := startSelection
            RETURNEXECRIGHT
        endif

    SELECTNEXTSTATENODE2 ≡
        let sn = Self.stateNodesToBeChecked.selectNextStateNode in
            if sn = undefined then
                UNDEFINEDBEHAVIOUR
            elseif sn.stateNodeKind = procedureNode then
                Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \
                    collectCurrentSubStates(sn.getPreviousStatePartition)
                // only state partitions of the state machine to be considered here
            elseif sn.stateNodeKind = statePartition then
```

```
            Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}
        elseif sn.stateNodeKind = stateNode then
            Self.stateNodeChecked := sn
            Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}
            Self.transitionsToBeChecked := sn.stateTransitions.continuousSignalTransitions
            Self.continuousPriorities := ∅
        endif
    endlet

CHECKFORINHERITEDSTATENODES ≡
    let sn = Self.stateNodeChecked in
        let sn1 = selectInheritedStateNode(sn, Self.stateNodesToBeChecked) in
            if sn1 ≠ undefined then
                Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn1}
                Self.stateNodeChecked := sn1
                Self.transitionsToBeChecked :=
                    sn1.stateTransitions.continuousSignalTransitions
                Self.continuousPriorities := Self.continuousPriorities ∪
                    { t.s-NAT | t ∈ sn.stateTransitions.continuousSignalTransitions}
            else
                Self.stateNodeChecked := undefined
            endif
        endlet
    endlet
endwhere
```

All current state nodes of the agent are checked in an arbitrary order, beginning, for each state partition, with the innermost state node. The latter reflects the priority among conflicting transitions. Furthermore, when a particular state node is being checked, the inherited state nodes are checked. Finally, redefined transitions take precedence over conflicting inherited transitions also in case of continuous signals. If no continuous signal is found, another cycle of the transition selection is started.

```
SELCONTINUOUSEVALUATIONPHASE ≡
    if Self.currentLabel ≠ undefined then
        choose b: b ∈ behaviour ∧ b.s-LABEL = Self.currentLabel
            EVAL(b.s-ACTION)
        endchoose
    elseif semvalueBool(value(Self.transitionChecked.s-LABEL,Self)) then
        TRANSITIONFOUND(Self.transitionChecked)
    else
        Self.agentMode4 := selectionPhase
    endif
```

For each continuous signal, the continuous expression has to be evaluated. As this evaluation consists of several actions in general, another agent mode, *evaluationPhase*, is entered. After completion of the evaluation, either the considered continuous signal is consumed, or the selection continues.

F3.2.3.2.12 Spontaneous transition selection

Selection of a spontaneous transition is performed by enabling, at any time during selection, a single spontaneous transition.

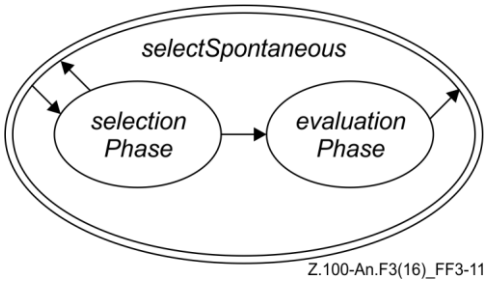

Figure F3-11 – Activity phases of SDL-2010 agents: selecting spontaneous transitions (level 5)

Since any time the agent mode *selectSpontaneous* is entered, only one spontaneous transition is checked, there are only two sub-modes (*agentMode5*), as shown in the diagram.

```
SELECTSPONTANEOUS ≡
    if Self.agentMode5 = selectionPhase then
        SELSPONTANEOUSSELECTIONPHASE
    elseif Self.agentMode5 = evaluationPhase then
        SELSPONTANEOUSEVALUATIONPHASE
    endif
```

This ASM macro defines the upper level control structure of the spontaneous transition selection. Depending on the agent mode*agentMode5*, further action is defined in the corresponding ASM macro. This control structure is part of the previous state diagram.

```
SELSPONTANEOUSSELECTIONPHASE ≡
    if Self.stateNodeChecked.stateTransitions.spontaneousTransitions ≠ ∅ then
        choose t: t ∈ Self.stateNodeChecked.stateTransitions.spontaneousTransitions
            if t.s-LABEL ≠ undefined then
                EVALUATEENABLINGCONDITION(t)
            else
                Self.sender := Self.selfPid
                TRANSITIONFOUND(t)
            endif
        endchoose
    else
        Self.agentMode4 := selectionPhase
    endif

where
// EvaluateEnablingCondition defined in F3.2.3.2.10 Input selection (in "where" for SelInputSelectionPhase)
endwhere
```

For a given state node, an arbitrary spontaneous transition is selected, and it is checked whether this transition is fireable.

```
SELSPONTANEOUSEVALUATIONPHASE ≡
    if Self.currentLabel ≠ undefined then
        choose b: b ∈ behaviour ∧ b.s-LABEL = Self.currentLabel
            EVAL(b.s-ACTION)
        endchoose
    elseif semvalueBool(value(Self.transitionChecked.s-LABEL,Self)) then
        Self.sender := Self.selfPid
        TRANSITIONFOUND(Self.transitionChecked)
    else
            Self.agentMode4 := selectionPhase
    endif
```

If a spontaneous transition has a provided expression, this expression has to be evaluated before continuing with the selection. As this evaluation consists of several actions in general, another agent

mode, *evaluationPhase*, is entered. After completion of the evaluation, either the considered spontaneous transition is selected, or the selection of priority input, input or continuous signals is resumed.

F3.2.3.2.13 Transition firing

The firing of a transition is decomposed into the firing of individual actions, which may in turn consist of a sequence of steps. At the beginning of a transition, the current state node is left; at the end, either a state node is entered, or a termination takes place.

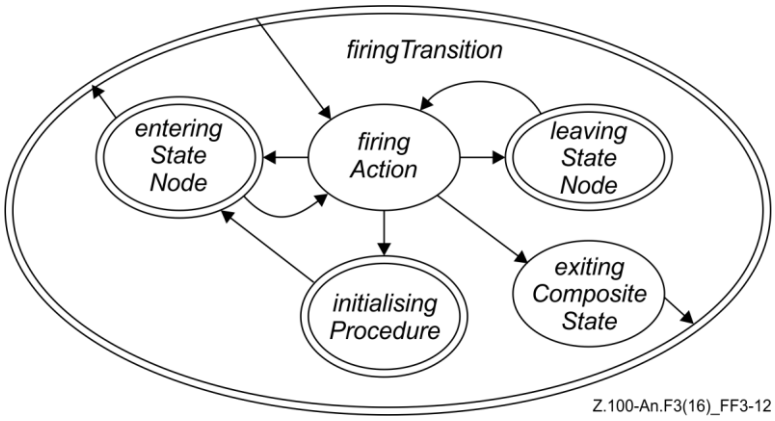

Figure F3-12 – Activity phases of SDL-2010 agents: firing transitions (level 3)

```
FIRETRANSITION ≡
    if Self.agentMode3 = firingAction then
        FIREACTION
    elseif Self.agentMode3 = leavingStateNode then
        LEAVESTATENODES
    elseif Self.agentMode3 = enteringStateNode then
        ENTERSTATENODES
    elseif Self.agentMode3 = exitingCompositeState then
        EXITCOMPOSITESTATE
    elseif Self.agentMode3 = initialisingProcedure then
        INITPROCEDURE
    endif
```

Firing of a transition consists of firing a sequence of actions. Once started, transitions are completely executed.

F3.2.3.2.14 Firing of actions

```
FIREACTION ≡
    if Self.currentLabel ≠ undefined then
        choose b: b ∈ behaviour ∧ b.s-LABEL = Self.currentLabel
            EVAL(b.s-ACTION)
        endchoose
    else
        Self.agentMode2 := selectingTransition
        Self.agentMode3 := startSelection
        RETURNEXECRIGHT
    endif
```

Firing of actions is defined by the selection and evaluation of the corresponding SAM primitives. Once started, the firing of actions continues until either a transition is completed (i.e., the current label has the value *undefined*) or until the agent mode is changed during the evaluation of a primitive. This is, for instance, the case when a state node is entered. The function *currentLabel* uniquely identifies a behaviour primitive.

F3.2.3.2.15 Entering of state nodes

```
ENTERSTATENODES ≡
    if Self.agentMode4 = startPhase then
        ENTERSTATENODESSTARTPHASE
    elseif Self.agentMode4 = enterPhase then
        ENTERSTATENODESENTERPHASE
    elseif Self.agentMode4 = enteringFinished then
        ENTERSTATENODESENTERINGFINISHED
    endif
```

State nodes are entered when the execution of an agent starts, and possibly when a next state action is executed. When this phase is started, a single state node with an entry point has already been selected. Depending on the structure of the hierarchical graph, further state nodes to be entered may be encountered when this single state node is entered.

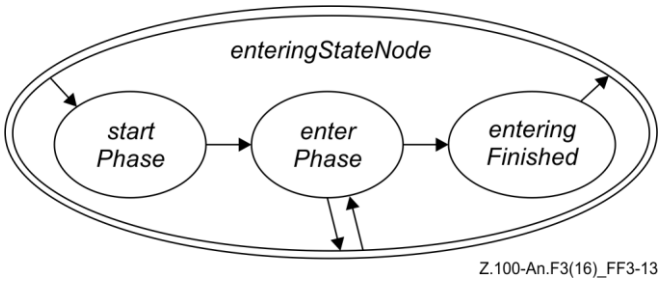

Figure F3-13 – Activity phases of SDL-2010 agents: entering state node (level 4)

```
ENTERSTATENODESSTARTPHASE ≡
    Self.agentMode4 := enterPhase
```

At the beginning of this phase, the set of entered state nodes is initialized. This set is updated every time another state node is entered, and evaluated at the end of the phase to determine the set of current state nodes of the agent.

```
ENTERSTATENODESENTERPHASE ≡
    if Self.stateNodesToBeEntered ≠ ∅ then
        choose snwen: snwen ∈ Self.stateNodesToBeEntered
            snwen.s-STATENODE.currentSubStates := ∅
            snwen.s-STATENODE.currentExitPoints := ∅
            snwen.s-STATENODE.previousSubStates := ∅
            if snwen.s-STATENODE.parentStateNode ≠ undefined then
                snwen.s-STATENODE.parentStateNode.currentSubStates :=
                    snwen.s-STATENODE.parentStateNode.currentSubStates ∪ {snwen.s-STATENODE}
            endif
            if snwen.s-STATENODE.stateNodeRefinement = undefined then
                REFINEMENTUNDEF(snwen)
            elseif snwen.s-STATENODE.stateNodeRefinement = stateAggregationNode then
                REFINEMENTSTATEAGGRNODE(snwen)
            elseif snwen.s-STATENODE.stateNodeRefinement = compositeStateGraph then
                REFINEMENTCOMPSTATENODE(snwen)
            endif
        endchoose
    else
        Self.agentMode4 := enteringFinished
    endif

where

    REFINEMENTUNDEF(snwen:STATENODEWITHENTRYPOINT) ≡
        let sn:[STATENODE] =
```

```
            take({sn1 ∈ STATENODE: directlyInheritsFrom(snwen.s-STATENODE,sn1)}) in
        if sn ≠ undefined then
            // refinement possibly inherited
            Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪
                {mk-STATENODEWITHENTRYPOINT(sn,
                snwen.s-implicit)}
        else
            Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen}
        endif
    endlet

REFINEMENTSTATEAGGRNODE(snwen:STATENODEWITHENTRYPOINT) ≡
    if snwen.s-implicit = HISTORY then
        Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪
            { mk-STATENODEWITHENTRYPOINT(s, HISTORY) |
                    s ∈ snwen.s-STATENODE.previousSubStates }
    else
        Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪
            { mk-STATENODEWITHENTRYPOINT(sp,
            entryConnection(snwen.s-implicit, sp)) |
            sp ∈ snwen.s-STATENODE.statePartitionSet}
    endif
    let aggr: State-aggregation-node = snwen.s-STATENODE. stateDefinitionAS1.s-implicit in
    if aggr.s1-Procedure-definition ≠ undefined // Entry-procedure-definition defined
    then  CREATEPROCEDURE(aggr.s1-Procedure-definition, undefined, undefined, undefined,
undefined)
    endif
    endlet // aggr

REFINEMENTCOMPSTATENODE(snwen:STATENODEWITHENTRYPOINT) ≡
    Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen}
    let comp: Composite-state-graph = snwen.s-STATENODE.stateDefinitionAS1.s-implicit in
    if comp.s-Procedure-definition ≠ undefined // Entry-procedure-definition defined
    then  CREATEPROCEDURE(comp.s-Procedure-definition, undefined, undefined, undefined, undefined)
    endif
    endlet // comp
    if snwen.s-implicit = HISTORY then
        Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪
            { mk-STATENODEWITHENTRYPOINT(s, HISTORY) |
                    s ∈ snwen.s-STATENODE.previousSubStates }
    else
        Self.currentStartNodes := Self.currentStartNodes ∪ {snwen}
    endif
endwhere
```

Entering of state nodes continues until the set *stateNodesToBeEntered* is empty. A distinction is made between state nodes with and without a refinement. If there is a refinement into a state aggregation node, then the entry procedure of that node is to be executed, and all state partitions are to be entered. If there is a refinement into a composite state graph, then a start transition has to be selected and executed, which determines a substate to be entered. Finally, if the state node is not refined, it may be belong to a composite state with a state type inheriting from another state type, where it is refined.

```
ENTERSTATENODESENTERINGFINISHED ≡
    Self.agentMode2 := selectingTransition
    Self.agentMode3 := startSelection
    RETURNEXECRIGHT
```

When the set *stateNodesToBeEntered* is empty, the transition selection is activated by setting the agent modes accordingly.

F3.2.3.2.16 Leaving of state nodes

```
LEAVESTATENODES ≡
    if Self.agentMode4 = leavePhase then
        LEAVESTATENODESLEAVEPHASE
    elseif Self.agentMode4 = leavingFinished then
        LEAVESTATENODESLEAVINGFINISHED
    endif
```

State nodes are left when transitions are fired. The set of state nodes to be left has already been determined when this rule macro is applied.

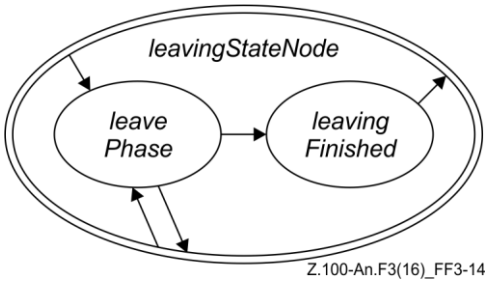

Figure F3-14 – Activity phases of SDL-2010 agents: leaving state node (level 4)

```
LEAVESTATENODESLEAVEPHASE ≡
    let sn = Self.stateNodesToBeLeft.selectNextStateNode in
        if sn = undefined then
            Self.agentMode4 := leavingFinished
        else
            Self.stateNodesToBeLeft := Self.stateNodesToBeLeft \ {sn}
            sn.parentStateNode.currentSubStates := sn.parentStateNode.currentSubStates \ {sn}
            sn.parentStateNode.previousSubStates := sn.parentStateNode.previousSubStates ∪ {sn}
            if sn.stateNodeRefinement = compositeStateGraph then
                let comp : Composite-state-graph = sn.stateAS1.s-Identifier.refersto1.s-implicit in
                if comp.s2-Procedure-definition ≠ undefined // Exit-procedure-definition defined
                then
                    CREATEPROCEDURE
                        (comp.s2-Procedure-definition, undefined, undefined, undefined, undefined)
                endif
                endlet // comp
            elseif sn.stateNodeRefinement = stateAggregationNode then
                let aggr: State-aggregation-node = sn.stateAS1.s-Identifier.refersto1.s-implicit in
                if aggr.s-Procedure-definition ≠ undefined // Exit-procedure-definition defined
                then
                    CREATEPROCEDURE
                        (aggr.s2-Procedure-definition, undefined, undefined, undefined, undefined)
                endif
                endlet // aggr
            endif
        endif
    endlet
```

In the leave phase, state nodes that have been collected are left, from bottom to top, with possible synchronization at state aggregation nodes. If defined, exit procedures are executed.

```
LEAVESTATENODESLEAVINGFINISHED ≡
        if Self.stateNodeToBeExited ≠ undefined then
            Self.currentExitStateNodes := {Self.stateNodeToBeExited}
            Self.stateNodeToBeExited := undefined
            Self.agentMode3 := exitingCompositeState
        else
            Self.agentMode3 := firingAction
```

```
        Self.currentLabel := Self.continueLabel
        Self.continueLabel := undefined
    endif
```

When the leaving of a state node has been completed, either the exiting of a state node or firing of the current transition has to be continued.

F3.2.3.2.17 Exiting of composite states

```
EXITCOMPOSITESTATE ≡
    if Self.stateNodeToBeExited ≠ undefined then
        let sn = Self.stateNodeToBeExited.s-STATENODE in
            if sn.stateNodeKind = stateNode then
                Self.currentExitStateNodes := {Self.stateNodeToBeExited}
                Self.stateNodeToBeExited := undefined
                Self.agentMode2 := selectingTransition
                Self.agentMode3 := startPhase
            elseif sn.stateNodeKind = statePartition then
                sn.parentStateNode.currentExitPoints := sn.parentStateNode.currentExitPoints
                    ∪ {Self.stateNodeToBeExited.s-STATEEXITPOINT}
                Self.stateNodesToBeLeft := {sn}
                Self.agentMode3 := leavingStateNode
                Self.agentMode4 := leavePhase
            endif
        endlet
    elseif Self.currentExitStateNodes ≠ ∅ then
        let snwex = take(Self.currentExitStateNodes) in
            let sn = snwex.s-STATENODE in
                if sn.parentStateNode.currentSubStates = ∅ then
                    let ep = take(sn.parentStateNode.currentExitPoints) in
                        Self.stateNodeToBeExited := mk-STATENODEWITHEXITPOINT(
                            sn.parentStateNode, exitConnection(ep,sn))
                        Self.currentExitStateNodes := ∅
                    endlet
                else
                    Self.currentExitStateNodes := ∅
                    Self.agentMode2 := selectingTransition
                    Self.agentMode3 := startPhase
                endif
            endlet
        endlet
    endif
```

F3.2.3.2.18 Stopping agent execution

An agent ceases to exist as soon as all contained agents have been terminated: that is, for all agent sets where the owner is *Self*, there no longer exists an agent where the owner is the agent set. Until all enclosed agent instances cease to exist, the agent continues to handle #get_ and #set_ remote procedure calls of global variables.

```
STOPPHASE ≡
    if ∀sas ∈ { SDLAGENTSET: sas.owner = Self }: (∃sa ∈ SDLAGENT: sa.owner = sas)
    then // handle CALL signals for #get_ and #set_ remote procedure calls of global variables
        Self.inport.schedule := // discard all signals except those ending in "#setCALL" or "#getCALL"
            <   siginst in Self.inport.schedule
            :       sigtoken = siginst.s-Identifier.s-Name.s-TOKEN // signal name as Token
                ∧   sigtokenend = substring(sigtoken, sigtoken.length-7, 8) // last 8 chars of name
                ∧   (sigtokenend = "#setCALL" ∨ sigtokenend = "#getCALL")
            >
        do forall siginst in ∈ Self.inport.queue
            let sigtoken = siginst.s-Identifier.s-Name.s-TOKEN in // signal name as Token
            let sigtokenend = substring(sigtoken, sigtoken.length-7, 8) in // last 8 chars of name
```

```
            let gname = mk-Name( // extract global variable name from signal name
                < c[i]:
                      c = substring(sigtoken, 1, sigtoken.length-8) // signal name without last 8 chars
                  ∧ j ∈ 1..sigtoken.length-8 ∧ c[j] = "#" // char = "#" in c
                  ∧ ¬ (∃ k ∈ j+1..sigtoken.length-8: c[k] = "#") // no char = "#" in rest of c
                  ∧ i ∈ j+1..sigtoken.length-8 // chars after last "#" in name without last 8 chars
                > ) // char sequence for global name, mk-Name
            in
            let gVarId = mk-Identifier(Self.agentAS1.qualifierWithinId1, gname) in
            if sigtokenend = "#setCALL"
            then // assign global variable from value in set signal, reply to sender
                ASSIGN(gVarId, siginst.s-VALUE[1], Self.stateAgent.state, Self.currentStateId)
                SIGNALOUTPUT(
                    mk-Identifier(siginst.s-Identifier.s-PATHITEM,
                        mk-Name(substring(sigtoken, 1, sigtoken.length-8)+"#setREPLY")) // signal Identifier
                    < siginst.s-VALUE[2] >, // signal parameters list - < n > where n in CALL
                    0.0, // activation delay
                    0, // signal priority
                    siginst.signalSender,  // to Arg – sender Pid
                    undefined // via Arg,
                ) // Signal Output
            else // get global variable value, and return to sender in reply
                SIGNALOUTPUT(
                    mk-Identifier(siginst.s-Identifier.s-PATHITEM,
                        mk-Name(substring(sigtoken, 1, sigtoken.length-8)+"#getREPLY")) // signal Identifier
                    < eval(gVarId, Self.stateAgent.state, Self.currentStateId) , // return value of variable
                      siginst.s-VALUE[1] // n  where n in CALL
                    >, // signal parameters list
                    0.0, // activation delay
                    0, // signal priority
                    siginst.signalSender,  // to Arg – sender Pid
                    undefined // via Arg,
                ) // Signal Output
            endif
            endlet // gVarId
            endlet // gname
            endlet // sigtokenend
            endlet // sigtoken
            Self.inport.schedule := delete(siginst, Self.inport.schedule)
        enddo
    else // remove agent sets for contained agents and agent for Self
        REMOVEALLAGENTSETS(Self)
        REMOVEAGENT(Self)
    endif
```

Reference sections

Clause F3.2.3.2.18 gives description of #get_ and #set_ remote procedure calls of global variables.

F3.2.3.3 Interface between execution and compilation

The execution of agents requires certain behaviour parts (called "compilation units") to be treated during compilation. Compilation units are sequences of actions of an agent that, once started, are executed without being interleaved by other actions of this agent or an agent belonging to the same set of nested agents:

- (Regular) transitions: Each transition starts with the evaluation of input parameters (if any), followed by an action *LEAVESTATENODE*, followed by *Transition* as defined in the abstract syntax. If the terminator of the transition is a *Nextstate-node*, the transition ends with an action "enterStateNode".
- Start transitions (*Named-start-node*, *State-start-node*, *Procedure-start-node*): These are associated with the containing state node.

- Exit transitions (*Named-return-node*): These are associated with the set of transitions of the containing state node.
- Expressions: During the selection phase, enabling conditions and continuous signals have to be evaluated. In these cases, the evaluation of an expression is a compilation unit.

Each compilation unit has a start label. Once a start label is assigned to the function *currentLabel* of an agent, the sequence of actions that begins with this label – the evaluation of an expression or the firing of a transition – is sequentially executed. This means that whenever an action has been executed, the compilation determines the continue label such that the next action follows. The termination of this sequence is "signalled" by having the continue label set to *undefined* after the last action of the sequence.

During compilation, a function *uniqueLabel*: *DEFINITIONAS1* × *NAT*→ *LABEL* associates unique labels with each node of the AST. The unique labels of nodes corresponding to compilation units are used as starting labels. Furthermore, labels are used to retrieve the result of the evaluation of expressions.

F3.3 Data semantics

F3.3.1 Predefined data

To determine if an operator is functional or if a procedure is in a given type, the qualifier of the identifier is used. A qualifier is a sequence of path items, each of which a name of an entity kind that can be a qualifier, such as a package name. Neither *Qualifier* nor *Path-item* is defined as a constructor in AS1. The domain *PATHITEM* is defined so that a qualifier can be selected as path item sequence.

```
PATHITEM =def    Agent-qualifier
    ∪ Agent-type-qualifier
    ∪ Compound-node-qualifier
    ∪ Data-type-qualifier
    ∪ Interface-qualifier
    ∪ Package-qualifier
    ∪ Procedure-qualifier
    ∪ State-qualifier
    ∪ State-type-qualifier
```

An operator is functional if it is predefined. The built-in procedures for structures and literals are treated as predefined. An <operation definition> represents a *Procedure-definition* in the *Procedure-definition-**set*** of the directly enclosing *Data-type-definition*. Therefore the procedure for a predefined operator is within the predefined data type, and the qualifier for the procedure identifier is "Predefined" followed by the data type name.

```
functional(oplitSig: OPLITSIGNATURE, values: VALUE*): BOOLEAN =def
    let entdefPaths =
        case oplitSig of
            | Static-operation-signature ∪ Dynamic-operation-signature then
                oplitSig.s-Operation-signature.s-Procedure-identifier.refersto1 // procedure definition
            | Literal-signature then
                oplitSig.s-Literal-signature.s-Result.s-Identifier.refersto1 // data type definition
        endcase.identifier1.s-PATHITEM-seq
    in
        (   entdefPaths.head ∈ Package-qualifier ∧ entdefPaths.head.s-Name.s-TOKEN = "Predefined"
        ∧   entdefPaths[2] ∈ Data-type-qualifier ∧ entdefPaths.length = 2) //
    ∨ isSpecialChoiceOp(oplitSig)
    ∨ isSpecialStructOp(oplitSig)
    ∨ isSpecialLiteralOp(oplitSig)
```

```
intype(oplitSig: OPLITSIGNATURE, name: Name): BOOLEAN =def
    oplitSig.identifier1.s-PATHITEM-seq.last.s-Data-type-qualifier = name

compute(oplitSig: OPLITSIGNATURE, values: VALUE* ): VALUEORXCEPTION =def
        if intype(oplitSig, ArrayType.s-Name) then computeArray(oplitSig, values)
        elseif intype(oplitSig, BagType.s-Name) then computeBag(oplitSig, values)
        elseif intype(oplitSig, BitType.s-Name) then computeBit(oplitSig, values)
        elseif intype(oplitSig, BitstringType.s-Name) then computeBitstring(oplitSig, values)
        elseif intype(oplitSig, BooleanType.s-Name) then computeBoolean(oplitSig, values)
        elseif intype(oplitSig, CharacterType.s-Name) then computeChar(oplitSig, values)
        elseif intype(oplitSig, CharstringType.s-Name) then computeCharstring(oplitSig, values)
        elseif intype(oplitSig, DurationType.s-Name) then computeDuration(oplitSig, values)
        elseif intype(oplitSig, IntegerType.s-Name) then computeInteger(oplitSig, values)
        elseif intype(oplitSig, OctetstringType.s-Name) then computeOctetstring(oplitSig, values)
        elseif intype(oplitSig, PowersetType.s-Name) then computePowerset(oplitSig, values)
        elseif intype(oplitSig, RealType.s-Name) then computeReal(oplitSig, values)
        elseif intype(oplitSig, StringType.s-Name) then computeString(oplitSig, values)
        elseif intype(oplitSig, TimeType.s-Name) then computeTime(oplitSig, values)
        elseif intype(oplitSig, VectorType.s-Name) then computeVector(oplitSig, values)
        elseif isSpecialChoiceOp(oplitSig) then computeChoice(oplitSig, values)
        elseif isSpecialStructOp(oplitSig) then computeStruct(oplitSig, values)
        elseif isSpecialLiteralOp (oplitSig) then computeLiteral(oplitSig, values)
        else
            raise(OutOfRange)
        endif
```

The *TOKEN* domain consists of character strings. The function *emptyToken* is therefore an empty character string.

```
emptyToken: TOKEN =def
    ""
```

The function *definingSort* computes the value data type definition scope in which an operator signature or literal signature is defined.

```
definingSort(opLitSig: OPLITSIGNATURE): Identifier =def
    oplitSig.parentAS1.identifier1
```

The function *opltName* computes the token of an operator or literal.

```
opltName(oplitSig: OPLITSIGNATURE): TOKEN =def
    case oplitSig of
    | Static-operation-signature ∪ Dynamic-operation-signature then
        oplitSig.s-Operation-signature.s-Name.s-TOKEN // Name is the Operation-name
    | Literal-signature then
        oplitSig.s-Literal-signature.s-Name.s-TOKEN // Literal-name
    endcase
```

F3.3.1.1 Predefined Data Types, Exceptions and Boolean Operations

A set of functions refers to predefined *Value-data-type-definition* nodes from the package Predefined.

```
BooleanType: Identifier =def
    mk-Identifier(<mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("Boolean"))

CharacterType: Identifier =def
    mk-Identifier(<mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("Character"))

StringType: Identifier =def
    mk-Identifier(<mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("String"))

CharstringType: Identifier =def
```

mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Charstring"))

IntegerType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Integer"))

RealType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Real"))

ArrayType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Array"))

VectorType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Vector"))

PowersetType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Powerset"))

DurationType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Duration"))

TimeType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Time"))

BagType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Bag"))

BitType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Bit"))

BitstringType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Bit"))

OctetstringType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Octetstring"))

PidType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Pid"))

NULLType: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("NULL"))

The unique literal value of NULL is:

nullNULL: *Identifier* =$_{def}$
mk-*Identifier*(
<**mk-***Package-qualifier*(**mk-***Name*("Predefined")), **mk-***Data-type-qualifier*(*NULLType*.**s-***Name*)>,
mk-*Name*("null"))

Furthermore, there are a number of predefined identifiers for exceptions.

OutOfRange: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("OutOfRange"))

InvalidReference: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>,**mk-***Name*("InvalidReference"))

NoMatchingAnswer: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>,**mk-***Name*("NoMatchingAnswer"))

UndefinedVariable: *Identifier* =$_{def}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>,**mk-***Name*("UndefinedVariable"))

UndefinedField: *Identifier* =$_{def}$

```
    mk-Identifier(<mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("UndefinedField"))

InvalidIndex: Identifier =def
    mk-Identifier(<mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("InvalidIndex"))

DivisionByZero: Identifier =def
    mk-Identifier(<mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("DivisionByZero"))

EmptyException: Identifier =def
    mk-Identifier(<mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("Empty"))

InvalidCall: Identifier =def
    mk-Identifier(<mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("InvalidCall"))

InvalidSort: Identifier =def
    mk-Identifier(<mk-Package-qualifier(mk-Name("Predefined"))>, mk-Name("InvalidSort"))
```

To raise an exception, the function *raise* is used. Each Predefined exception is an *Identifier* and is a member of the *EXCEPTION* domain (see clause F3.2.1.1.6). If *raise* is invoked the further behaviour of the system is not defined by SDL-2010.

```
raise(ex:Identifier): Identifier =def
    UNDEFINEDBEHAVIOUR
```

There are also the following predefined operation signatures:

```
sdlAnd: Static-operation-signature =def
    mk-Static-operation-signature(mk-Operation-signature(mk-Name("""and"""),
        < (BooleanType), (BooleanType)>,
        BooleanType,
        mk-Identifier(
            <   mk-Package-qualifier(mk-Name("Predefined")),
                mk-Data-type-qualifier(mk-Name("Boolean"))
            >,
            mk-Name("""and""")
        ) // Identifier
    )) // Operation-signature, Static-operation-signature

sdlOr: Static-operation-signature =def
    mk-Static-operation-signature(mk-Operation-signature(mk-Name("""or"""),
        < (BooleanType), (BooleanType)>
        BooleanType,
        mk-Identifier(
            <   mk-Package-qualifier(mk-Name("Predefined")),
                mk-Data-type-qualifier(mk-Name("Boolean"))
            >,
            mk-Name("""or""")
        ) ) // Identifier
    )) // Operation-signature, Static-operation-signature

sdlTrue: Literal-signature =def
    mk-Literal-signature (mk-Name("true"), mk-Result(BooleanType, PART), 0)

sdlFalse: Literal-signature =def
    mk-Literal-signature (mk-Name("false"), mk-Result(BooleanType, PART), 1)
```

F3.3.1.2 Boolean

The function *computeBoolean* determines the value of an application of a Predefined Boolean operator.

SDLBOOLEAN =def *BOOLEAN* × *Identifier*

computeBoolean(*oplitSig*: *OPLITSIGNATURE*, *values*: *VALUE**): *VALUEOREXCEPTION* =def
let *restype* = *definingSort*(*oplitSig*) **in**
if *oplitSig* ∈ *Literal-signature*
then mk-*SDLBOOLEAN*(**if** *oplitSig.opltName* = "true" **then** *true* **else** *false* **endif**, *restype*)
else case *oplitSig.opltName* **of**
| """not""" **then** **mk-***SDLBOOLEAN*(¬*values.head.semvalueBool*, *restype*)
| """and""" **then** **mk-***SDLBOOLEAN*(*values.head.semvalueBool* ∧ *values.tail.head.semvalueBool*, *restype*)
| """or""" **then** **mk-***SDLBOOLEAN*(*values.head.semvalueBool* ∨ *values.tail.head.semvalueBool*, *restype*)
| """xor""" **then** **mk-***SDLBOOLEAN*(¬(*values.head.semvalueBool* ⇔ *values.tail.head.semvalueBool*), *restype*)
| """=>""" **then** **mk-***SDLBOOLEAN*(*values.head.semvalueBool* ⇒ *values.tail.head.semvalueBool*, *restype*)
| "Null" **then** // For any sort, the operator Null returns the *Null-literal-signature* of that sort. Z.107 12.1.4
mk-*Literal-signature*(**mk-***Name*("null"), **mk-***Result*(*oplitSig.definingSort*, **REF**), 0), // 0 arbitrary
endcase endif
endlet // restype

semvalueBool(*v*:*SDLBOOLEAN*): *BOOLEAN* =def *v*.**s-***BOOLEAN*

F3.3.1.3 Integer

SDLINTEGER =def *NAT* × *Identifier*
// Nat is 2 * the integer value for positive integers, and (2* -value)-1 for negative integers.
// Identifier is the type name – normally IntegerType but could be a subtype

mkSDLInteger(*v*: *INT*, *restype*: *Identifier*): *SDLINTEGER* =def
mk-*SDLINTEGER*(**if** *v* < 0 **then** -v*2 -1 **else** *v**2 **endif**, *restype*)

semvalueInt(*v*:*SDLINTEGER*): *INT*=def **if** *isevenNat* (*v*.**s-***NAT*) **then** *v*.**s-***NAT*/2 **else** –((*v*.**s-***NAT*+1)/2) **endif**

computeInteger(*oplitSig*: *OPLITSIGNATURE*, *values*: *VALUE**): *VALUEOREXCEPTION* =def
let *restype* = *definingSort*(*oplitSig*) **in**
if *oplitSig* ∈ *Literal-signature* **then**
integerLiteral(0,*oplitSig.opltName*, *restype*) // opltName is the digits
elseif *oplitSig.opltName* = """-""" ∧ *values.length* = 1 **then**
mkSDLInteger(0 - *values.head.semvalueInt*, *restype*)
elseif *oplitSig.opltName* ∈
{ """+""", """-""", """*""", """/""",
"""mod""", """rem""",
"""<""", """>""", """<=""", """>=""",
"integer", "num", "power"
}
then
let *val1* = *values*[1].*semvalueInt* **in**
let *val2* = *values*[2].*semvalueInt* **in**
case *oplitSig.opltName* **of**
| """+""" **then** *mkSDLInteger*(*val1+val2*, *restype*)
| """-""" **then** *mkSDLInteger*(*val1 – val2*, *restype*)
| """*""" **then** *mkSDLInteger*(*val1 * val2*, *restype*)
| """/""" **then**
if *val2* = 0 **then**
raise(*DivisionByZero*)
else
mkSDLInteger(*intDiv*(*val1*,*val2*), *restype*)
endif
| """mod""" **then**
if *val2* = 0 **then**
raise(*DivisionByZero*)
else
mkSDLInteger(*intMod*(*val1*,*val2*), *restype*)

```
                endif
            | """rem""" then
                if val2 = 0 then
                    raise(DivisionByZero)
                else
                    mkSDLInteger(intRem(val1,val2), restype)
                endif
            | "integer" then mkSDLInteger(val1, restype) // cast Integer to sub-type
            | "num" then mkSDLInteger(val1, IntegerType) // cast sub type to Integer
            | "power" then mkSDLInteger(intPower(val1,val2), restype)
            | """<""" then mk-SDLBOOLEAN(val1 < val2, BooleanType)
            | """<=""" then mk-SDLBOOLEAN(val1 ≤ val2, BooleanType)
            | """>""" then mk-SDLBOOLEAN(val1 > val2, BooleanType)
            | """>=""" then mk-SDLBOOLEAN(val1 ≥ val2, BooleanType)
        | "Null" then
            // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
            mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
            endcase
        endlet // val2
        endlet // val1
        else raise(OutOfRange)
    endif
    endlet // restype
```

The function *numberValue* determines the *NAT* associated with a single character in the range "0" to "9".

```
numberValue(c:TOKEN): NAT =def
    case c of
    | "0" then 0
    | "1" then 1
    | "2" then 2
    | "3" then 3
    | "4" then 4
    | "5" then 5
    | "6" then 6
    | "7" then 7
    | "8" then 8
    | "9" then 9
    endcase
```

The function *integerLiteral* returns the *SDLINTEGER* value for an integer literal.

```
integerLiteral(num: NAT, digits: TOKEN, type: Identifier): SDLINTEGER =def
    if digits = emptyToken then
        mkSDLInteger(num, type)
    else
        integerLiteral(num*10 + numberValue(digits.head), digits.tail, type)
    endif
```

The function *intDiv* returns the result of integer-dividing its arguments.

```
intDiv(a: INT, b: INT):INT =def
    if a ≥ 0 ∧ b > a then                0
    elseif a ≥ 0 ∧ b ≤ a ∧ b > 0 then    1 + intDiv(a - b, b)
    elseif a ≥ 0 ∧ b < 0 then            - intDiv(a, -b)
    elseif a < 0 ∧ b < 0 then            intDiv (-a, -b)
    elseif a < 0 ∧ b > 0 then            - intDiv (-a, b)
    else raise(DivisionByZero)
    endif
```

The function *intMod* returns the result of the integer-modulo operation.

```
intMod(a: INT, b: INT):INT =def
    if a ≥ 0 ∧ b > 0 then                          intRem(a,b)
    elseif b < 0 then                              intMod(a, -b)
    elseif a < 0 ∧ b > 0 ∧ intRem(a,b) = 0 then    intRem(a,b)
    elseif a < 0 ∧ b >0 ∧ intRem(a,b) < 0 then     b + intRem(a,b)
    else raise(DivisionByZero)
    endif
```

The function *intRem* returns the result of the integer-remainder operation.

```
intRem(a: INT, b: INT):INT =def
    a - b * intDiv(a,b)
```

The function *intPower* returns the result of the integer-power operation.

```
intPower(a: INT, b: INT):INT =def
    if      b = 0  then  1
    elseif  a = 0  then  0
    elseif  b > 0  then  a * intPower(a, b-1)
    else                 intDiv(intPower(a, b+1), a)
    endif
```

F3.3.1.4 Character

Character values are represented by their name.

```
SDLCHARACTER =def Name × Identifier

computeChar(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
    let restype = definingSort(oplitSig) in
    if oplitSig ∈ Literal-signature then
        mk-SDLCHARACTER(oplitSig.s-Name, restype) // Name is Literal-name of Literal-signature
    elseif oplitSig.opltName = "num" then
        mkSDLInteger(charValue(values.head.s-Name), IntegerType)
    elseif oplitSig.opltName = "chr" then
        mk-SDLCHARACTER(values.head.semvalueInt.charChr, restype)
    elseif oplitSig.opltName = "Null" then
        // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
        mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
    else raise(OutOfRange)
    endif
    endlet // restype
```

The function *charValue* returns the numerical value of the character.

```
charValue(ch: Name): NAT =def
    let myDef: Value-data-type-definition = CharacterType.refersto1 in
    let literals = myDef.s-Literal-signature-set in
        take({L.s-NAT | L ∈ literals: L.s-Name = ch})
        // Nat is Literal-natural of Literal-signature. Name is Literal-name of Literal-signature
    endlet // literals
    endlet // myDef
```

The function *charChr* returns the character for a given Integer.

```
charChr(a: INT): Name =def
    if a > 128 then charChr(a - 128)
    elseif a < 0 then charChr(a+128)
    else
        let char: Value-data-type-definition = CharacterType.refersto1 in
        let literals = char.s-Literal-signature-set in
            take({L.s-Name | L ∈ literals: L.s-NAT = a})
        // Name is Literal-name of Literal-signature. Nat is Literal-natural of Literal-signature.
        endlet // literals
```

endlet // char
endif

F3.3.1.5 Real

The Predefined type Real is a rational number, with numerator and denominator, and constrained such that there is no common divisor between the numerator and denominator, and any intermediate number with a common divisor between the numerator and denominator has both divided by the divisor to produce a Real. Therefore, 5/3, 10/6 … 35/21 … are all denotations of the same Real value. This includes all the values that can be denoted by the <name> for Real literals of the form <integer name> <full stop> <integer name> [{e|E}[<hyphen> | <plus sign>] <integer name> (see clause 6.1 of [ITU-T Z.101]), and other rational values (such as 1/3) that cannot be denoted precisely in the <real name> form. Negative real values do not have a <real name> denotation and have to be written using a minus operator.

Real literals and other Real constant expressions (such as -2.0/7.0) are statically evaluated as described in clause F2.2.10 Computing constants; in particular the function *computeReal1*. The evaluation result is a *Literal-signature*, where the *Literal-name* contains a *TOKEN* with an optional "-" (indicating the value is negative), followed by a digit string for the rational numerator, followed by a "/" character, followed by a digit string for the rational denominator. The *Literal-signature* also has a *Result* that has a *Sort-reference-identifier* (for the Real data type in a context Real is needed, but for Time/Duration where needed), and a *NAT* value that is a unique mapping from the canonical (no common divisor) rational. The *Sort-reference-identifier* of the *Result* being a Real data type, invokes the *computeReal* function.

The domain *SDLREAL* contains the Real value as an *INT* for the rational numerator (negative for negative real values) and a *NAT* for the rational denominator. This differs from *NAT* used to represent the value in a *Literal-signature* because it is simpler, and there is not the constraint (that applied for *Literal-signature*) that all values should map to a unique *NAT*.

SDLREAL =$_{\text{def}}$ $r \in \{$ *INT* × *NAT* × *Identifier* : ((*r*.**s**-*INT* = 0 ⇒ (*r*.**s**-*NAT* = 1))
∨ (*r*.**s**-*INT* > 0 ∧ *r*.**s**-*NAT* > 0 ∧ *gcd*(*r*.**s**-*INT*, *r*.**s**-*NAT*) = 1))
∧ (*r*.**s**-*NAT* ≠ 0)}

semvalueRealNum(*v*: *SDLREAL*): *INT* =$_{\text{def}}$ *v*.**s**-*INT*

semvalueRealDen(*v*: *SDLREAL*): *NAT* =$_{\text{def}}$ *v*.**s**-*NAT*

semvalueReal(*v*: *SDLREAL*): *REAL*=$_{\text{def}}$
let *res*: *REAL* = *v.semvalueRealNum* / *v.semvalueRealDen* **in** // REAL division (not integer division)
res
endlet // res

computeReal(*oplitSig*: *OPLITSIGNATURE*, *values*: *VALUE**): *VALUEORECEPTION* =$_{\text{def}}$
let *restype* = *definingSort*(*oplitSig*) **in**
if *oplitSig* ∈ *Literal-signature* **then**
realLiteral(0,1,*oplitSig.opltName*, *restype*) // the opltName is the <real name>
elseif *oplitSig.opltName* = "Null" ∧ *values.length* = 0 **then**
// For any sort, the operator Null returns the *Null-literal-signature* of that sort. Z.107 12.1.4
mk-*Literal-signature*(**mk**-*Name*("null"), **mk**-*Result*(*oplitSig.definingSort*, **REF**), 0), // 0 arbitrary
elseif *oplitSig.opltName* = """-""" ∧ *values.length* = 1 **then**
mk-*SDLREAL*(0 - *values.head.semvalueRealNum*, *values.head.semvalueRealDen*, *restype*)
elseif *oplitSig.opltName* ∈ {"""+""", """-""", """*""", """/""", """<""", """>""", """<=""", """>="""}
then
let *num1* = *values*[1].*semvalueRealNum* **in**
let *den1* = *values*[1].*semvalueRealDen* **in**
let *num2* = *values*[2].*semvalueRealNum* **in**
let *den2* = *values*[2].*semvalueRealDen* **in**
case *oplitSig.opltName* **of**

```
        | """+""" then // note that correct for num1 and/or num2 being negative
        let gcdn = gcd(num1*den2 + num2*den1, den1*den2) in
            mk-SDLREAL(num1*den2 + num2*den1 / gcdn, den1*den2 / gcdn, restype)
        endlet // gcdn
        | """-""" then // note that correct for num1 and/or num2 being negative
        let gcdn = gcd(num1*den2 - num2*den1, den1*den2) in
            mk-SDLREAL(num1*den2 - num2*den1 / gcdn, den1*den2 / gcdn, restype)
        endlet // gcdn
        | """*""" then  // note that correct for num1 and/or num2 being negative
        let gcdn = gcd(num1*num2, den1*den2) in
            mk-SDLREAL(num1*num2 / gcdn, den1*den2 / gcdn, restype)
        endlet // gcdn
        | """/""" then  // note that correct for num1 and/or num2 being negative
            if num2 = 0 then
                raise(DivisionByZero)
            else
            let gcdn = gcd(num1/num2, den1*den2) in
                mk-SDLREAL(num1/num2 / gcdn, den1*den2 / gcdn, restype)
            endlet // gcdn
            endif
        | """<""" then mk-SDLBOOLEAN(num1*den2 < num2*den1, BooleanType)
        | """<=""" then mk-SDLBOOLEAN(num1*den2 ≤ num2*den1, BooleanType)
        | """>""" then mk-SDLBOOLEAN(num1*den2 ≥ num2*den1, BooleanType)
        | """>=""" then mk-SDLBOOLEAN(num1*den2 ≥ num2*den1, BooleanType)
        endcase
    endlet // den2
    endlet // num2
    endlet // den1
    endlet // num1
    elseif oplitSig.opltName = "float" then
        mk-SDLREAL(semvalueInt(values.head), 1, restype)
    elseif oplitSig.opltName = "fix" then
        mkSDLInteger(computeFix(values.head.semvalueRealNum,
            values.head.semvalueRealDen), IntegerType)
    else raise(OutOfRange)
    endif
    endlet // restype
```

The function *realLiteral* returns the *SDLREAL* value for a real literal.

```
realLiteral(num: NAT, den: NAT, realName: TOKEN, type: Identifier): SDLREAL =def
 // The realName is initially the character string for a <real name> in a Literal-signature from F2
// of the form: <integer name> / <integer name>
// num is initially 0, den is initially 1
    if "-" = realName.head
    then // name starts with "-", negative value
        mk-SDLREAL(-realLiteral(0, 1, realName.tail, type).semvalueRealNum, // negate numerator
            realLiteral(0, 1, realName.tail, type).semvalueRealDen, type)
    elseif "/" in realName then  // "/" in realName, therefore before "/"
        realLiteral(num*10 + numberValue(realName.head), den, realName.tail, type) // digits before "/"
    elseif realName.head = "/" then
        realLiteral(num, den, realName.tail, type ) // removes "/"
    elseif realName ≠ emptyToken then // more digits to handle after "/"
        realLiteral(num*10 + numberValue(realName.head), den*10, realName.tail, type)
    else // realName = emptyToken handled every character of name
        mk-SDLREAL(num/gcd(num, den), den/gcd(num, den), type) // final result (except negation if needed)
    endif
```

The function *computeFix* returns the *INT* value for a given numerator and denominator.

```
computeFix(num: INT, den: NAT): INT =def
    if num < 0 then
        - computeFix(- num, den) - 1
```

```
    elseif num < den then
        0
    else
        computeFix (num - den, den) + 1
    endif
```

F3.3.1.6 Duration

The domain *SDLDURATION* is based on the domain *REAL*.

```
SDLDURATION =def REAL × Identifier

computeDuration(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUE =def
    let restype = oplitSig.definingSort in
    if oplitSig ∈ Literal-signature then
        realLiteral(0, 1, oplitSig.opltName, restype)
    else
        case oplitSig.opltName of
        | "duration" then
            let val: SDLREAL = values.head in
                mk-SDLDURATION(val.s-INT, val.s-NAT, DurationType)
            endlet // val
        | "Null" then
            // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
            mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
        otherwise computeReal(oplitSig, values) // result sort Boolean or DurationType derived from oplitSig
        endcase
    endif
    endlet // restype
```

F3.3.1.7 Time

The domain *SDLTIME* is based on the domain *REAL*.

```
SDLTIME=def REAL × Identifier

computeTime(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUE =def
    let restype = oplitSig.definingSort in
    if oplitSig ∈ Literal-signature then
        realLiteral(0, 1, 1, oplitSig.opltName, restype)
    else
        case oplitSig.opltName of
        | "time" then
            let val: SDLREAL = values.head in
                mk-SDLTIME(val.s-INT, val.s-NAT, TimeType)
            endlet // val
        | """-""" then
        let res: SDLREAL = computeReal(oplitSig,values) in
            if values.head ∈ SDLTIME ∧ values.tail.head ∈ SDLDURATION then
                mk-SDLTIME(res.s-INT, res.s-NAT, TimelType)
            else // values.head SDLTime and values.tail.head SDLTime
                mk-SDLREAL(res.s-INT, res.s-NAT, RealType)
            endif
        endlet // res
        | "Null" then
            // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
            mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
        otherwise computeReal(oplitSig, values) // result sort Boolean or TimeType derived from oplitSig
        endcase
    endif
    endlet // restype
```

F3.3.1.8 String

A string type is defined as a sequence of its element type. String is a parameterized sort of data with an unbound parameter for the element and is therefore abstract and cannot be used directly.

```
SDLSTRING =def VALUE* × Identifier

computeString(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
let restype = definingSort(oplitSig) in
    case oplitSig.opltName of
    | "emptystring" then  mk-SDLSTRING(empty, restype)
    | "mkstring" then     mk-SDLSTRING(< values.head >, restype)
    | "Make" then         mk-SDLSTRING(< values.head >, restype)
    | "length" then       mkSDLInteger(values.head.s-VALUE-seq.length, IntegerType)
    | "first" then        values.head.s-VALUE-seq.head
    | "last" then         values.head.s-VALUE-seq.last
    | """//""" then       mk-SDLSTRING(values[1].s-VALUE-seq ⌢ values[2].s-VALUE-seq, restype)
    | "Extract" then
        let string = values[1].s-VALUE-seq in
        let intval: SDLINTEGER = values[2] in
        let index = intval.s-NAT/2 in
            if index < 0 ∨ index > string.length then
                raise(InvalidIndex)
            else
                string[index]
            endif
        endlet // index
        endlet // intval
        endlet // string
    | "Modify" then
        let intval: SDLINTEGER = values[2] in
        let index = intval.s-NAT/2  in
        let front = substr(values[1].s-VALUE-seq, 1, index-1) in
        let back = substr(values[1].s-VALUE-seq, index+1, values[1].s-VALUE-seq.length - index) in
            if InvalidIndex = front ∨ InvalidIndex = back then raise(InvalidIndex)
            else
                mk-SDLSTRING(front ⌢ <values[3]> ⌢ back, restype)
            endif
        endlet // back
        endlet // front
        endlet // index
        endlet // intval
    | "substring" then
        let from: SDLINTEGER = values[2] in
        let to: SDLINTEGER = values[3] in
        let val = substr(values[1].s-VALUE-seq, from.s-NAT, to.s-NAT) in
            if InvalidIndex = val then raise(InvalidIndex)
            else mk-SDLSTRING(val, restype) endif
        endlet // val
        endlet // to
        endlet // from
    | "remove" then
        let intval: SDLINTEGER = values[2] in
        let index = intval.s-NAT/2  in
        let front = substr(values[1].s-VALUE-seq, 1, index-1) in
        let back = substr(values[1].s-VALUE-seq, index+1, values[1].s-VALUE-seq.length - index) in
            if InvalidIndex = front ∨ InvalidIndex = back then raise(InvalidIndex) else
                mk-SDLSTRING(front ⌢ back, restype)
            endif
        endlet // back
        endlet // front
```

```
            endlet // index
            endlet // intval
        | "Null" then // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
            mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
        endcase
    endlet // restype
```

The function *substr* computes the substring of a string value.

```
substr(str: VALUE*,start: NAT, len: NAT): VALUE* ∪ EXCEPTION =def
    if start ≤ 0 ∨ len < 0 ∨ start+len-1 > str.length then
        raise(InvalidIndex)
    elseif len = 0 then
        empty
    else
        substr(str,start,len-1) ⌢ <str[start+len-1] >
    endif
```

F3.3.1.9 Charstring

A character string (sort name Charsting) inherits the string type, with the item sort parameter bound to the character sort the emptystring operator renamed as " and literals defined as character strings of any length starting with apostrophe and ending with an apostrophe, and an apostrophe in the string represented by an apostrophe pair.

```
SDLCHARSTRING =def SDLCHARACTER* × Identifier

computeCharstring(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
let restype = definingSort(oplitSig) in
    if oplitSig ∈ Literal-signature then
        charstringLiteral(oplitSig.opltName, restype)
    else
        case oplitSig.opltName of
        | """ then  mk-SDLCHARSTRING(empty, restype) // name is <apostrophe><apostrophe>
        | "mkstring" then      mk-SDLCHARSTRING(< values.head >, restype)
        | "Make" then      mk-SDLCHARSTRING(< values.head >, restype)
        | "length" then    mkSDLInteger(values.head.s-VALUE-seq.length, IntegerType)
        | "first" then   values.head.s-SDLCHARACTER-seq.head
        | "last" then    values.head.s-SDLCHARACTER-seq.last
        | """//""" then      mk-SDLCHARSTRING(values[1].s-VALUE-seq ⌢ values[2].s-VALUE-seq, restype)
        | "Extract" then
            let string = values[1].s-VALUE-seq in
            let intval: SDLINTEGER = values[2] in
            let index = intval.s-NAT/2 in
                if index < 0 ∨ index > string.length then
                    raise(InvalidIndex)
                else
                    string[index]
                endif
            endlet // index
            endlet // intval
            endlet // string
        | "Modify" then
            let intval: SDLINTEGER = values[2] in
            let index = intval.s-NAT/2  in
            let front = substr(values[1].s-VALUE-seq, 1, index-1) in
            let back = substr(values[1].s-VALUE-seq, index+1, values[1].s-VALUE-seq.length - index) in
                if InvalidIndex = front ∨ InvalidIndex = back then raise(InvalidIndex)
                else
                    mk-SDLCHARSTRING(front ⌢ <values[3]> ⌢ back, restype)
                endif
```

```
            endlet // back
            endlet // front
            endlet // index
            endlet // intval
        | "substring" then
            let from: SDLINTEGER = values[2] in
            let to: SDLINTEGER = values[3] in
            let val = substr(values[1].s-VALUE-seq, from.s-NAT, to.s-NAT) in
                if InvalidIndex = val then raise(InvalidIndex)
                else mk-SDLCHARSTRING(val, restype) endif
            endlet // val
            endlet // to
            endlet // from
        | "remove" then
            let intval: SDLINTEGER = values[2] in
            let index = intval.s-NAT/2  in
            let front = substr(values[1].s-VALUE-seq, 1, index-1) in
            let back = substr(values[1].s-VALUE-seq, index+1, values[1].s-VALUE-seq.length - index) in
                if InvalidIndex = front ∨ InvalidIndex = back then raise(InvalidIndex) else
                    mk-SDLCHARSTRING(front ⌢ back, restype)
                endif
            endlet // back
            endlet // front
            endlet // index
            endlet // intval
        | "Null" then
            // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
            mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
        endcase
    endif
endlet // restype
```

The function *charstringLiteral* returns the *SDLCHARSTRING* value for a character string.

```
charstringLiteral(t: TOKEN, restype: Identifier): SDLCHARSTRING =def
// first and last characters of t are apostrophes
    if t = """ // empty charstring - <apostophe><apostophe>
    then    mk-SDLCHARSTRING(empty, restype)
    elseif t = """""  //  <apostophe><apostophe><apostophe><apostophe> - represents apostophe
    then    mk-SDLCHARSTRING(< mk-SDLCHARACTER(t[2], characterType) >, restype) // apostophe
    elseif t.length = 3 //  single character string
    then    mk-SDLCHARSTRING(< mk-SDLCHARACTER(t[2], characterType) >, restype)
    else // all strings more than one one character
        charstingLiteral(""" ⌢ t[2] ⌢ """, restype)
        ⌢
        charstingLiteral(""" ⌢ t.tail.tail, restype) // t without old t[2]
    endif
```

F3.3.1.10 Array

An array is represented as a set of indexsort/itemsort pairs, with an optional default value. Array is a parameterized sort of data with unbound parameters for both the indexsort and itemsort. It is therefore abstract and cannot be used directly.

```
SDLARRAY =def VALUEPAIR-set × [VALUE] × Identifier

VALUEPAIR =def VALUE × VALUE // index  × item

computeArray(oplitSig: OPLITSIGNATURE, valueseq: VALUE*): VALUEOREXCEPTION =def
    let  restype = definingSort(oplitSig) in
    if oplitSig.opltName = mk-Name("Make") then
```

```
        if valueseq.length = 0 then
            mk-SDLARRAY(∅, undefined, restype)
        else
            mk-SDLARRAY(∅,values.head, restype)
        endif
    elseif oplitSig.opltName = mk-Name("Modify") then
        let a = valueseq[1] in
        let index = valueseq[2] in
        let value = valueseq[3] in
            mk-SDLARRAY(modifyArray(a.s-VALUEPAIR-set, index, value), a.s-VALUE, restype)
        endlet // value
        endlet // index
        endlet // a
    elseif oplitSig.opltName = mk-Name("Extract") then
        let v = take({ f.s2-VALUE | f ∈ valueseq[1].s-VALUEPAIR-set: f.s1-VALUE = valueseq[2]}) in
            if v = undefined
            then if valueseq[1].s-VALUE ≠ undefined then valueseq[1].s-VALUE else raise(InvalidIndex) endif
            else v
            endif
        endlet // v
    elseif oplitSig.opltName = mk-Name("Null") then
        // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
        mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
    else raise(OutOfRange)
    endif
    endlet // restype

modifyArray(a: VALUEPAIR-set, index: VALUE, value: VALUE): VALUEPAIR-set =def
    { item | item ∈ a: item.s1-VALUE ≠ index } ∪ { mk-VALUEPAIR(index,value)}
```

F3.3.1.11 Vector

A vector is a parameterized sort of data that inherits *SDLARRAY*, where the index sort is bound to a syntype of *SDLINTEGER* with values in the range 1 to a maximum index. The maximum index is defined as an additional (synonym context) parameter, *maxindex*. *SDLVECTOR* remains an abstract sort until the item sort and maximum index is bound, and therefore cannot be used directly. When a Modify or Extract operator is called for an un-parameterized sort of data that inherits Vector, the index parameter is checked to be in range (1..*max*, where *maxindex* has been bound to *max*), as part of the assignment of the actual parameter of the operator.

```
SDLVECTOR =def VECTORPAIR-set × [VALUE] × Identifier

VECTORPAIR =def NAT × VALUE

computeVector(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
    let restype = definingSort(oplitSig) in
    if oplitSig.opltName = mk-Name("Make") then
        if values.length = 0 then
            mk-SDLVECTOR(∅, undefined, restype)
        else
            mk-SDL VECTOR(∅,values.head, restype)
        endif
    elseif oplitSig.opltName = mk-Name("Modify") then
        let a = values[1] in
        let index = values[2] in
        let value = values[3] in
            mk-SDLVECTOR(modifyVector(a.s-VECTORPAIR-set, index, value), a.s-VALUE, restype)
        endlet // value
        endlet // index
        endlet // a
    elseif oplitSig.opltName = mk-Name("Extract") then
```

```
        let v = take({ f.s-VALUE | f ∈ values[1].s-VEXTORPAIR-set: f.s-NAT = values[2]}) in
            if v = undefined then
            then if values[1].s-VALUE ≠ undefined then values[1].s-VALUE else raise(InvalidIndex) endif
            else v
            endif
        endlet // v
    elseif oplitSig.opltName = mk-Name("Null") then
        // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
        mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
    else raise(OutOfRange)
    endif
    endlet // restype

modifyVector(a: VECTORPAIR-set, index: NAT, value: VALUE): VALUEPAIR-set =def
    { item | item ∈ a: item.s-NAT ≠ index } ∪ { mk-VALUEPAIR(index,value)}
```

F3.3.1.12 Powerset

A Powerset is represented as a set.

```
SDLPOWERSET =def VALUE-set × Identifier

computePowerset (oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
    let restype = definingSort(oplitSig) in
    case oplitSig.opltName of
    | "empty" then mk-SDLPOWERSET(∅,restype)
    | """in""" then mk-SDLBOOLEAN(values[1] ∈ values[2].s-VALUE-set, BooleanType)
    | "incl" then mk-SDLPOWERSET(values[2].s-VALUE-set ∪ {values[1] }, restype)
    | "del" then mk-SDLPOWERSET(values[2].s-VALUE-set \ {values[1] }, restype)
    | """<""" then mk-SDLBOOLEAN(values[1].s-VALUE-set ⊂ values[2].s-VALUE-set, BooleanType)
    | """<=""" then mk-SDLBOOLEAN(values[1].s-VALUE-set ⊆ values[2].s-VALUE-set, BooleanType)
    | """>""" then mk-SDLBOOLEAN(values[2].s-VALUE-set ⊂ values[1].s-VALUE-set, BooleanType)
    | """>=""" then mk-SDLBOOLEAN(values[2].s-VALUE-set ⊆ values[1].s-VALUE-set, BooleanType)
    | """and""" then mk-SDLPOWERSET(values[1].s-VALUE-set ∩ values[2].s-VALUE-set, restype)
    | """or""" then mk-SDLPOWERSET(values[1].s-VALUE-set ∪ values[2].s-VALUE-set, restype)
    | "length" then mkSDLInteger( | values[1].s-VALUE-set |, IntegerType)
    | "take" then if values[1].s-VALUE-set = ∅ then
                    raise(EmptyException)
                else
                    values[1]. s-VALUE-set.take
                endif
    | "Null" then // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
        mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
    endcase
    endlet // restype
```

F3.3.1.13 Bag

A Bag is represented as a set of value-frequency pairs.

```
SDLBAG =def FREQUENCY-set × Identifier

FREQUENCY =def VALUE × NAT

computeBag (oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
    let restype = definingSort(oplitSig) in
    case oplitSig.opltName of
    | "empty" then mk-SDLBAG(∅,restype)
    | """in""" then mk-SDLBOOLEAN(bagcount(values[1], values[2]) ≠ 0, BooleanType)
    | "incl" then mk-SDLBAG(bagincl(values[1], values[2]), restype)
    | "del" then mk-SDLBAG(bagdel(values[1], values[2]), restype)
    | """<""" then mk-SDLBOOLEAN(baginbag(values[1], values[2]), BooleanType)
```

```
    | """<=""" then mk-SDLBOOLEAN(¬ baginbag(values[2], values[1]), BooleanType)
    | """>""" then mk-SDLBOOLEAN(baginbag(values[2], values[1]), BooleanType)
    | """>=""" then mk-SDLBOOLEAN(¬ baginbag(values[1], values[2]), BooleanType)
    | """and""" then mk-SDLBAG(bagand(values[1], values[2]), restype)
    | """or""" then mk-SDLBAG(bagor(values[1], values[2]), restype)
    | "length" then mkSDLInteger(baglength(values[1].s-FREQUENCY-set), IntegerType)
    | "take" then values[1].s-FREQUENCY-set.take.s-VALUE
    | "Null" then // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
        mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
    endcase
    endlet // restype

bagcount(item: VALUE, bag: SDLBAG): NAT =def
    let elem1 = {elem.s-NAT | elem ∈ bag.s-FREQUENCY-set: elem.s-VALUE = item } in
        if elem1 = ∅ then 0 else elem1.take endif
    endlet // elem1

bagincl(item: VALUE, bag: SDLBAG): FREQUENCY-set =def
    if bagcount(item, bag) ≠ 0 then
        {if elem.s-VALUE = item then mk-FREQUENCY(item, elem.s-NAT+1) else elem endif |
            elem ∈ bag.s-FREQUENCY-set}
    else
        bag.s-FREQUENCY-set ∪ {mk-FREQUENCY (item, 1)}
    endif

bagdel(item: VALUE, bag: SDLBAG): FREQUENCY-set =def
    if bagcount(item, bag) ≠ 1 then
        {if elem.s-VALUE = item then mk-FREQUENCY(item, elem.s-NAT - 1) else elem endif |
            elem ∈ bag.s-FREQUENCY-set}
    else
        bag.s-FREQUENCY-set \ { mk-FREQUENCY(item, 1)}
    endif

baginbag(smaller: SDLBAG, larger: SDLBAG): BOOLEAN =def
    ∀ elem ∈ smaller.s-FREQUENCY-set: bagcount(elem.s-VALUE, larger) < elem.s-NAT

bagand(a: SDLBAG,b: SDLBAG): FREQUENCY-set =def
    { mk-FREQUENCY (x.s-VALUE, min(bagcount(x.s-VALUE,a),bagcount(x.s-VALUE,b))) |
        x ∈ a.s-FREQUENCY-set: bagcount(x.s-VALUE, b) > 0}

min(a: NAT,b: NAT ): NAT =def if a>b then a else b endif

bagor(a: SDLBAG,b: SDLBAG): FREQUENCY-set =def
    { mk-FREQUENCY(x.s-VALUE, bagcount(x.s-VALUE,a) + bagcount(x.s-VALUE, b) )
        | x ∈ a.s-FREQUENCY-set }
    ∪ { x | x ∈ b.s-FREQUENCY-set: bagcount(x.s-VALUE, a) = 0}

baglength(a: FREQUENCY-set):NAT =def
    if a = ∅ then 0
    else let x = a.take in
        x.s-NAT + baglength(a \ {x})
        endlet
    endif
```

F3.3.1.14 Bit

```
SDLBIT =def BOOLEAN × Identifier
```

The function *computeBit* determines the value of an application of a Predefined Bit operator.

```
computeBit(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
    let restype = definingSort(oplitSig) in
```

```
    if oplitSig ∈ Literal-signature
    then mk-SDLBIT(if oplitSig.opltName = "1" then true else false endif, restype)
    else case oplitSig.opltName of
    // inherited from SDL Boolean
    | """not""" then   mk-SDLBIT(¬values.head.semvalueBit, restype)
    | """and""" then   mk-SDLBIT(values.head.semvalueBit ∧ values.tail.head.semvalueBit restype)
    | """or""" then    mk-SDLBIT(values.head.semvalueBit ∨ values.tail.head.semvalueBit, restype)
    | """xor""" then   mk-SDLBIT(¬(values.head.semvalueBit ⇔ values.tail.head.semvalueBit), restype)
    | """=>""" then    mk-SDLBIT(values.head.semvalueBit ⇒ values.tail.head.semvalueBit, restype)
    // added
    | "num" then
        mk-SDLINTEGER(
            mk-NAT( if values.head.semvalueBit = true then 2 else 0 endif), // mk-Nat
            IntegerType
        ) // mk-SDLInteger
    | "bit" then
        case values.head.semvalueInt
        | 0 then mk-SDLBIT(false, restype)
        | 1 then mk-SDLBIT(true, restype)
        otherwise raise(OutOfRange)
        endcase
    | "Null" then // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
        mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
    endcase endif
    endlet // restype

semvalueBit(v:SDLBIT): BOOLEAN =def v.s-BOOLEAN
```

F3.3.1.15 Bitstring

The function *computeBitstring* determines the value of an application of a Predefined Bitstring operator.

```
SDLBITSTRING =def BOOLEAN* × Identifier

computeBitstring(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
    let restype = definingSort(oplitSig) in
    if oplitSig ∈ Literal-signature
    then bitstringLiteral(mk-SDLBITSTRING(<>, restype), oplitSig.opltName)
    else case oplitSig.opltName of
    // The following operators that are the same as String except Bitstring is indexed from zero
    | "mkstring" then      mk-SDLBITSTRING(< values.head.semvalueBit >, restype)
    | "Make" then      mk-SDLBITSTRING(< values.head.semvalueBit >, restype)
    | "length" then    mk-SDLINTEGER(values.head.s-VALUE-seq.length, IntegerType) // mk-SDLInteger
    | "first" then   mk-SDLBIT(values.head.semvalueBit, Bittype)
    | "last" then    mk-SDLBIT(values.last.semvalueBit, Bittype)
    | """//""" then mk-SDLBITSTRING(values.head.s-VALUE-seq ⌢ values.tail.head.s-VALUE-seq, restype)
    | "Extract" then
        if  values.tail.head.semvalueInt ≥ 0 ∧ values.tail.head.semvalueInt < values.head.s-VALUE-seq.length
        then mk-SDLBIT(values.head.s-VALUE-seq[values.tail.head.semvalueInt+1].semvalueBit, Bittype)
        else raise(InvalidIndex)
        endif
    | "Modify" then
        if   values.tail.head.semvalueInt ≥ 0 ∧ values.tail.head.semvalueInt < values.head.s-VALUE-seq.length
        then
            mk-SDLBITSTRING(
                <   if  i ≠ values.tail.head.semvalueInt
                    then values.head.s-VALUE-seq[i+1].semvalueBit
                    else values.tail.tail.head.semvalueBit
                    endif
                : i ∈ 0..(values.head.s-VALUE-seq.length-1)
                >,
```

```
            restype
        ) //  mk-SDLBitstring
    else raise(InvalidIndex)
    endif
| "substring" then
    if      values.tail.head.semvalueInt ≥ 0
        ∧   values.tail.head.semvalueInt < values.head.s-VALUE-seq.length
        ∧   values.tail.tail.head.semvalueInt > 0
        ∧   values.tail.head.semvalueInt+values.tail.tail.head.semvalueInt≤values.head.s-VALUE-
seq.length
    then
    mk-SDLBITSTRING(
        <   values.head.s-VALUE-seq[i+1].semvalueBit
        : i ∈ values.tail.head.semvalueInt..
            (values.tail.head.semvalueInt+values.tail.tail.head.semvalueInt-1)
        >,
        restype
    ) //  mk-SDLBitstring
    else raise(InvalidIndex)
    endif
| "remove" then
    if      values.tail.head.semvalueInt ≥ 0
        ∧   values.tail.head.semvalueInt < values.head.s-VALUE-seq.length
        ∧   values.tail.tail.head.semvalueInt > 0
        ∧   values.tail.head.semvalueInt+values.tail.tail.head.semvalueInt≤values.head.s-VALUE-
seq.length
    then
    mk-SDLBITSTRING(
        <   values.head.s-VALUE-seq[i+1].semvalueBit
        : i ∈ ((0..values.tail.head.semvalueInt-1)
            ∪
            (values.tail.head.semvalueInt+values.tail.tail.head.semvalueInt
                ..values.head.s-VALUE-seq.length-1)
        >,
        restype
    ) //  mk-SDLBitstring
    else raise(InvalidIndex)
    endif
| """not""" then
    if      values.head.s-VALUE-seq ≠ < >
    then
        mk-SDLBITSTRING(
            < ¬values.head.s-VALUE-seq[i].semvalueBit : i ∈ 1..values.head.s-VALUE-seq.length >,
            restype
        ) //  mk-SDLBitstring
    else mk-SDLBITSTRING( < >, restype)
    endif
| """or""" then
    if   values.head.s-VALUE-seq = < > ∧ values.tail.head.s-VALUE-seq = < >
    then mk-SDLBITSTRING( < >, restype)
    else mk-SDLBITSTRING(
            <   if i ≤ values.head.s-VALUE-seq.length
                then values.head.s-VALUE-seq[i]
                else false
                endif
                ∨
                if i ≤ values.tail.head.s-VALUE-seq.length
                then values.tail.head.s-VALUE-seq[i]
                else false
                endif
            : i ∈ 1..
                if values.head.s-VALUE-seq.length > values.tail.head.s-VALUE-seq.length
```

```
            then values.head.s-VALUE-seq.length
            else values.tail.head.s-VALUE-seq.length
            endif
        >,
        restype
    ) //  mk-SDLBitstring
  endif
| """and""" then
  if   values.head.s-VALUE-seq = < > ∧ values.tail.head.s-VALUE-seq = < >
  then mk-SDLBITSTRING( < >, restype)
  else mk-SDLBITSTRING(
        <  if i ≤ values.head.s-VALUE-seq.length
           then values.head.s-VALUE-seq[i]
           else false
           endif
           ∧
           if i ≤ values.tail.head.s-VALUE-seq.length
           then values.tail.head.s-VALUE-seq[i]
           else false
           endif
        : i ∈ 1..
           if values.head.s-VALUE-seq.length > values.tail.head.s-VALUE-seq.length
           then values.head.s-VALUE-seq.length
           else values.tail.head.s-VALUE-seq.length
           endif
        >,
        restype
    ) //  mk-SDLBitstring
  endif
| """xor""" then
  if   values.head.s-VALUE-seq = < > ∧ values.tail.head.s-VALUE-seq = < >
  then mk-SDLBITSTRING( < >, restype)
  else mk-SDLBITSTRING(
        <  (if i ≤ values.head.s-VALUE-seq.length
           then values.head.s-VALUE-seq[i]
           else false
           endif
           ∨
           if i ≤ values.tail.head.s-VALUE-seq.length
           then values.tail.head.s-VALUE-seq[i]
           else false
           endif) // a or b
           ∧ ¬( // and not(a and b)
           if i ≤ values.head.s-VALUE-seq.length
           then values.head.s-VALUE-seq[i]
           else false
           endif
           ∧
           if i ≤ values.tail.head.s-VALUE-seq.length
           then values.tail.head.s-VALUE-seq[i]
           else false
           endif) // not(a and b)
        : i ∈ 1..
           if values.head.s-VALUE-seq.length > values.tail.head.s-VALUE-seq.length
           then values.head.s-VALUE-seq.length
           else values.tail.head.s-VALUE-seq.length
           endif
        >,
        restype
    ) //  mk-SDLBitstring
  endif
| """=>""" then
```

```
        if   values.head.s-VALUE-seq = < > ∧ values.tail.head.s-VALUE-seq = < >
        then mk-SDLBITSTRING( < >, restype)
        else mk-SDLBITSTRING(
                <    ¬( // not(a)
                     if i ≤ values.head.s-VALUE-seq.length
                     then values.head.s-VALUE-seq[i]
                     else false
                     endif ) // not
                     ∨
                     if i ≤ values.tail.head.s-VALUE-seq.length
                     then values.tail.head.s-VALUE-seq[i]
                     else false
                     endif// not(a) or b
                : i ∈ 1..
                     if values.head.s-VALUE-seq.length > values.tail.head.s-VALUE-seq.length
                     then values.head.s-VALUE-seq.length
                     else values.tail.head.s-VALUE-seq.length
                     endif
                >,
                restype
             ) //  mk-SDLBitstring
        endif
    | "Null" then // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
        mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
// added cf String
    | "num" then  mkSDLInteger(numBitstring( values.head.s-VALUE-seq, 0),IntegerType)
    | "bitstring" then
        if values.head.semvalueInt ≥ 0
        then mk-SDLBITSTRING(mkBitstringBools(values.head.semvalueInt), restype)
        else    raise(OutOfRange)
        endif
    endcase endif
    endlet // restype

numBitstring(bs: BOOLEAN*, n: NAT): NAT =def
    if bs.length = 0
    then n
    else numBitstring(bs.tail, 2*n + if bs.head = true then 1 else 0 endif)
    endif

mkBitstringBools(n: NAT): BOOLEAN* =def
    if n > 2
    then mkBitstringBools(n/2) ^  < if n.isevenNat then false else true endif >
    else < if n = 1 then true else false endif >
    endif
```

The function *bitstringLiteral* returns the *SDLBITSTRING* value for a bitstring literal.

```
bitstringLiteral(bs: SDLBITSTRING, nameToken: TOKEN): SDLBITSTRING =def
    if nameToken = ""B" ∨ nameToken = ""H" // two apostrophes followed by B  or H
    then bs
    else
        bitstringLiteral(
            mk-SDLBITSTRING(
                bs.s-BOOLEAN-seq ^
                if nameToken.last= "B"  //B notation : <apostrophe> { 0 | 1 }* <apostrophe> B
                then
                    < if nameToken[2] = "1"  then true else false endif >
                else // H notation: <apostrophe> <hexdigit> * <apostrophe> H
                    case nameToken[2] of
                    | "0" then < false, false, false, false >
```

```
                | "1" then < false, false, false, true >
                | "2" then < false, false, true, false >
                | "3" then < false, false, true, true >
                | "4" then < false, true, false, false >
                | "5" then < false, true, false, true >
                | "6" then < false, true, true, false >
                | "7" then < false, true, true, true >
                | "8" then < true, false, false, false >
                | "9" then < true, false, false, true >
                | "A" then < true, false, true, false >
                | "B" then < true, false, true, true >
                | "C" then < true, true, false, false >
                | "D" then < true, true, false, true >
                | "E" then < true, true, true, true >
                | "F" then < true, true, true, true >
                endcase
            endif,
            bs.s-Identifier
        ),  // mk-SDLBitstring
        """ + nameToken.tail.tail // <apostrophe> + tail from 3rd character
    )  // bitstringLiteral
endif
```

F3.3.1.16 Octet and Octetstring

Octet is a syntype of Bitstring with a constraint that the length is 8.

Octetstring is a String with each String item is an Octet. The literals for an Octetstring are the same as the literals for a Bitstring except for the B notation there has to be zero or a multiple of 8 bits, and the H notation there has to be zero or an even number of hexadecimal digits.

```
SDLOCTETSTRING =def {(octets, id) ∈ SDLBITSTRING* × Identifier
    : (∀ i ∈ 1..octets.length : octets[i].length = 8)}

computeOctetstring(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
let restype = definingSort(oplitSig) in
    if oplitSig ∈ Literal-signature
    then octetstringLiteral(mk-SDLOCTETSTRING(<>, restype), oplitSig.opltName)
    else case oplitSig.opltName of
// inherited from String
    // "emptystring" is replaced by "B
    | "mkstring" then      mk-SDLOCTETSTRING(< values.head >, restype)
    | "Make" then      mk-SDLOCTETSTRING(< values.head >, restype)
    | "length" then      mkSDLInteger(values.head.s-SDLBITSTRING-seq.length, IntegerType)
    | "first" then   values.head.s-SDLBITSTRING-seq.head
    | "last" then    values.head.s-SDLBITSTRING-seq.last
    | """//""" then      mk-SDLOCTETSTRING(values[1].s-SDLBITSTRING-seq
        ⌢ values[2].s-SDLBITSTRING-seq, restype)
    | "Extract" then
        let string = values[1].s-SDLBITSTRING-seq in
        let intval: SDLINTEGER = values[2] in
        let index = intval.s-NAT/2 in
            if index < 0 ∨ index > string.length then
                raise(InvalidIndex)
            else
                string[index]
            endif
        endlet // index
        endlet // intval
        endlet // string
    | "Modify" then
        let intval: SDLINTEGER = values[2] in
```

```
            let index = intval.s-NAT/2  in
            let front = substr(values[1].s-SDLBITSTRING-seq, 1, index-1) in
            let back = substr(values[1].s-SDLBITSTRING-seq, index+1,
                values[1].s-SDLBITSTRING-seq.length - index)
            in
                if InvalidIndex = front ∨ InvalidIndex = back then raise(InvalidIndex)
                else
                    mk-SDLOCTETSTRING(front ⌢ <values[3]> ⌢ back, restype)
                endif
            endlet // back
            endlet // front
            endlet // index
            endlet // intval
        | "substring" then
            let from: SDLINTEGER = values[2] in
            let to: SDLINTEGER = values[3] in
            let val = substr(values[1].s-SDLBITSTRING-seq, from.s-NAT, to.s-NAT) in
                if InvalidIndex = val then raise(InvalidIndex)
                else mk-SDLOCTETSTRING(val, restype) endif
             endlet
            endlet // val
            endlet // to
            endlet // from
        | "remove" then
            let intval: SDLINTEGER = values[2] in
            let index = intval.s-NAT/2  in
            let front = substr(values[1].s-SDLBITSTRING-seq, 1, index-1) in
            let back = substr(values[1].s-SDLBITSTRING-seq,
                index+1, values[1].s-SDLBITSTRING-seq.length - index)
            in
                if InvalidIndex = front ∨ InvalidIndex = back then raise(InvalidIndex) else
                    mk-SDLOCTETSTRING(front ⌢ back, restype)
                endif
            endlet // back
            endlet // front
            endlet // index
            endlet // intval
// end of operations inherited from String – add operations unique to Octetstring
        | "bitstring" then mk-SDLBITSTRING(bitstringOctets(values[1].s-SDLBITSTRING-seq), bitstringType)
        | "octetstring" then mk-SDLOCTETSTRING(octetstringBits(values[1].s-BOOLEAN-seq), restype)
        | "Null" then // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
            mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
        endcase endif
endlet // restype

bitstringOctets(octets: SDLBITSTRING*): BOOLEAN* =def
    if octets.length = 0 then < > // empty OctetString gives an empty Bitstring
    else octets[1].s-BOOLEAN-seq // Boolean sequence for first octet
        ⌢ bitstringOctets(octets.tail) // Boolean sequence for remaining octets
    endif

octetstringBits(bits: BOOLEAN*): SDLBITSTRING* =def
    if bits.length = 0 then < > // empty BitString gives an empty Octetstring
    else < < if i < bits.length then bits[i] else false endif: i ∈ 1..8 > > // Octetstring sequence for first bits
        // if there are fewer than 8 bits, the octet is padded with false values and the remaining bits are empty
        ⌢ octetstringBits(bits.tail.tail.tail.tail.tail.tail.tail.tail) // Octetstring sequence for remaining bits
    endif

octetstringLiteral(os: SDLOCTETSTRING, nameToken: TOKEN): SDLOCTETSTRING =def
    if nameToken = ""B" ∨ nameToken = ""H" // two apostrophes followed by B  or H
```

```
then os
else
    octetstringLiteral(
        mk-SDLOCTETSTRING(
            os.s-SDLBITSTRING-seq ^
            if nameToken.last= "B"  //B notation : <apostrophe> { (0 | 1) 8 }* <apostrophe> B
            then
                mk-SDLBITSTRING(
                    <   if nameToken[2] = "1"  then true else false endif,
                        if nameToken[3] = "1"  then true else false endif,
                        if nameToken[4] = "1"  then true else false endif,
                        if nameToken[5] = "1"  then true else false endif,
                        if nameToken[6] = "1"  then true else false endif,
                        if nameToken[7] = "1"  then true else false endif,
                        if nameToken[8] = "1"  then true else false endif,
                        if nameToken[9] = "1"  then true else false endif,
                    >, // Boolean sequence for Bitstring length 8
                    bitstringType
                )   // mk-SDLBitstring length 8
            else // H notation: <apostrophe> (<hexdigit>  <hexdigit>) * <apostrophe> H
                mk-SDLBITSTRING(
                    case nameToken[2] of // first of hexdigit pair
                    | "0" then < false, false, false, false >
                    | "1" then < false, false, false, true >
                    | "2" then < false, false, true, false >
                    | "3" then < false, false, true, true >
                    | "4" then < false, true, false, false >
                    | "5" then < false, true, false, true >
                    | "6" then < false, true, true, false >
                    | "7" then < false, true, true, true >
                    | "8" then < true, false, false, false >
                    | "9" then < true, false, false, true >
                    | "A" then < true, false, true, false >
                    | "B" then < true, false, true, true >
                    | "C" then < true, true, false, false >
                    | "D" then < true, true, false, true >
                    | "E" then < true, true, true, true >
                    | "F" then < true, true, true, true >
                    endcase
                    ^

                    case nameToken[3] of // second of hexdigit pair
                    | "0" then < false, false, false, false >
                    | "1" then < false, false, false, true >
                    | "2" then < false, false, true, false >
                    | "3" then < false, false, true, true >
                    | "4" then < false, true, false, false >
                    | "5" then < false, true, false, true >
                    | "6" then < false, true, true, false >
                    | "7" then < false, true, true, true >
                    | "8" then < true, false, false, false >
                    | "9" then < true, false, false, true >
                    | "A" then < true, false, true, false >
                    | "B" then < true, false, true, true >
                    | "C" then < true, true, false, false >
                    | "D" then < true, true, false, true >
                    | "E" then < true, true, true, true >
                    | "F" then < true, true, true, true >
                    endcase, // Boolean seqeunce for Bitstring length 8
                    bitstringType
                )   // mk-SDLBitstring length 8
            endif, // B or H notation
            os.s-Identifier
```

```
        ),  // mk-SDLOctetstring
        """ + // <apostrophe> +
            if nameToken.last= "B"
            then nameToken.tail.tail.tail.tail.tail.tail.tail.tail.tail //  B notation tail from 10 th character
            else nameToken.tail.tail.tail //  H notation tail from 4th character
            endif
    )  // octettstringLiteral
endif // empty Octetstring or literal
```

F3.3.2 Pid types

A *PID* value is represented by an agent and an interface.

PID =def *VALIDPID* ∪ *NULLPID*

NULLPID =def { **mk-***Literal-signature*(**mk-***Name*("null"), **mk-***Resul*(*Pidtype*, **REF**), 0) }

VALIDPID =def *SDLAGENT* × [*Interface-definition*]

static *nullPid*: *PID* =def *take*(*NULLPID*)

The static function *nullPid* is the special *PID* value for the unique named element of the Pid sort (denoted by "null") that does not identify any agent and is the unique element of *NULLPID*. See F2.2.4.2. Package, Auxiliary function: implicitInterface1, and F2.2.9.3 Interface type, Mapping to abstract syntax.

F3.3.3 Constructed types

F3.3.3.1 Literals

Values of a literal sort are represented by the literal signature and the type in which the literal is defined:

SDLLITERAL =def { (*lit*, *id*) ∈ *Literal-signature* × *Identifier* : *id.refersto1* ∈ *Value-data-type-definition* }

```
isSpecialLiteralOp(oplitSig: OPLITSIGNATURE): BOOLEAN =def
if oplitSig ∈ Literal-signature
then false
else // Operation-signature
    let  procsort = oplitSig.definingSort in
    let  pn = oplitSig.opltName in
    (∃ lit ∈ SDLLITERAL: (procsort = lit.s-Identifier )) ∧
    (    pn ∈ { """<""", """>""","""<=""","""">=""", "first", "last", "succ", "pred", "num","Null" })
    endlet // pn
    endlet // procsort
endif // Literal-signature or Operation-signature
```

The function *computeLiteral* gives the value of applying the language-defined operators for literals.

```
computeLiteral(oplitSig:OPLITSIGNATURE, values:VALUE*): [VALUE ]=def
if oplitSig ∈ Literal-signature
then undefined
else // Operation-signature
    let  restype = oplitSig.definingSort in
    let  defi: Value-data-type-definition = restype.refersto1 in
    let  v1 = values.head.s-Literal-signature.literalNum in
    let  v2 = values.tail.head.s-Literal-signature.literalNum in
    case oplitSig.opltName of
    | """>""" then mk-SDLBOOLEAN(v1 > v2, BooleanType)
    | """>=""" then mk-SDLBOOLEAN(v1 ≥ v2, BooleanType)
    | """<""" then mk-SDLBOOLEAN(v1 < v2, BooleanType)
    | """<=""" then mk-SDLBOOLEAN(v1 ≤ v2, BooleanType)
    | "first" then literalMinimum (defi.s-Literal-signature-set)
```

```
    | "last" then literalMaximum (defi.s-Literal-signature-set)
    | "succ" then literalSucc(defi.s-Literal-signature-set, values.head)
    | "pred" then literalPred(defi.s-Literal-signature-set, values.head)
    | "num" then mkSDLInteger(literalNum(values.head).semvalueInt, IntegerType)
    | "Null" then // For any sort, the operator Null returns the Null-literal-signature of that sort. Z.107 12.1.4
        mk-Literal-signature(mk-Name("null"), mk-Result(oplitSig.definingSort, REF), 0), // 0 arbitrary
    otherwise undefined
    endcase
    endlet // v2
    endlet // v1
    endlet // defi
    endlet // restype
endif // Literal-signature or Operation-signature

literalNum(s: Literal-signature): NAT =def
    s.s-NAT

literalValue(s: Literal-signature): VALUE =def
    mk-SDLLITERAL(s, s.s-Result.s-Identifier) // sort id of Value-data-type-definition

literalMinimum(s: Literal-signature-set): VALUE =def
    take({s1.literalValue | s1 ∈ s: ∀ s2 ∈ s:s1.literalNum ≤ s2.literalNum})

literalMaximum(s: Literal-signature-set): VALUE =def
    take({s1.literalValue | s1 ∈ s: ∀ s2 ∈ s:s2.literalNum ≤ s1.literalNum})

literalSucc(s: Literal-signature-set, val: SDLLITERAL): VALUE =def
    if val = literalMaximum (s, val.s-Identifier) then literalMinimum(s, val.s-Identifier)
    else
        take(
            {   s1.literalValue | s1 ∈ s ∧
                (s1.literalNum > val.s-NAT) ∧
                (∀s2 ∈ s: (s2.literalNum > val.s-NAT) ∧ (s1.literalNum < s2.literalNum ))
            })
    endif

literalPred(s: Literal-signature-set, val: SDLINTEGER): VALUE =def
    if val = literalMinimum(s, val.s-Identifier) then literalMaximum (s, val.s-Identifier)
    else
        take(
            {   s1.literalValue | s1 ∈ s ∧
                (s1.literalNum < val.s-NAT) ∧
                (∀s2 ∈ s: ( s2.literalNum < val.s-NAT ) ∧ (s2.literalNum < s1.literalNum ))
            })
    endif
```

F3.3.3.2 Structures

A structure value is identified by its type name, and the field list. The field names are a list, rather than a set because the Make operator uses the order of the fields rather than the field names.

```
SDLSTRUCTURE =def FIELD* × Identifier
FIELD =def Name × VALUE
```

The function *isStructSort* determines if a sort identifier refers to a value data type definition for a structure. Such a data type definition is characterised by having a "Make" operator, an "Undefined" operator, and for each field operators with the field name concatenated with "Modify", "Extract" and "Present". If it has a "PresentExtract" operator the sort is a choice.

```
isStructSort(sortid: Identifier): BOOLEAN =def
let  dtd = sortid.refersto1 in // data type definition
if dtd ∉ Value-data-type-definition
```

```
    ∨  (∃ opSig ∈ dtd.s-Static-operation-signature-set:
            opSig.s-Operation-signature.s-Name.s-TOKEN = "PresentExtract"
        ∧   opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
        // Operation result is not checked
       )
then false
elseif // check Operation-signatures
       ( ¬(∃ opSig ∈ dtd.s-Static-operation-signature-set:
            opSig.s-Operation-signature.s-Name.s-TOKEN = "PresentExtract"
        ∧   opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
        ) // if "PresentExtract" exists, it is a Choice sort
       )
    ∧  (∃ opSig ∈ dtd.s-Static-operation-signature-set:
            opSig.s-Operation-signature.s-Name.s-TOKEN = "Make"
        // Formal arguments not checked
        ∧   opSig.s-Operation-signature.s1-Identifier = sortid // Operation result
       )
    ∧  (∃ opSig ∈ dtd.s-Static-operation-signature-set:
            opSig.s-Operation-signature.s-Name.s-TOKEN = "Undefined"
        ∧   opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
        ∧   opSig.s-Operation-signature.s1-Identifier = BooleanType // Operation result
       )
    ∧  (∃ opSig ∈ dtd.s-Static-operation-signature-set:
            substring(opSig.s-Operation-signature.s-Name.s-TOKEN,
                opSig.s-Operation-signature.s-Name.s-TOKEN.length-5,
                6) = "Modify"
        ∧   opSig.s-Operation-signature.s-Identifier-seq[1] = sortid  // Formal argument for original value
        // Formal argument for the new field value not checked
        ∧   opSig.s-Operation-signature.s1-Identifier = sortid // Operation result
       )
    ∧  (∃ opSig ∈ dtd.s-Static-operation-signature-set:
            substring(opSig.s-Operation-signature.s-Name.s-TOKEN,
                opSig.s-Operation-signature.s-Name.s-TOKEN.length-6,
                7) = "Extract"
        ∧   opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
        // Operation result not checked
       )
    ∧  (∃ opSig ∈ dtd.s-Static-operation-signature-set:
            substring(opSig.s-Operation-signature.s-Name.s-TOKEN,
                opSig.s-Operation-signature.s-Name.s-TOKEN.length-6,
                7) = "Present"
        ∧   opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
        ∧   opSig.s-Operation-signature.s1-Identifier = BooleanType// Operation result
       )
    ∧  (∀ fn ∈ TOKEN : (
        ((fn + "Modify") ∈ { opTok ∈ opSig.s-Operation-signature.s-Name.s-TOKEN
            : opSig ∈ dtd.s-Static-operation-signature-set } // set for opTok
        )
        ⇒
        ((fn + "Extract") ∈ { opTok ∈ opSig.s-Operation-signature.s-Name.s-TOKEN
            : opSig ∈ dtd.s-Static-operation-signature-set } // set for opTok
        )
        ∧
        ((fn + "Present") ∈ { opTok ∈ opSig.s-Operation-signature.s-Name.s-TOKEN
            : opSig ∈ dtd.s-Static-operation-signature-set } // set for opTok
        )
       )) // for all fn
endif
endlet // dtd
```

The function *isSpecialStructOp* determines if the operator given is one of the language defined structure operators.

```
isSpecialStructOp(oplitSig: OPLITSIGNATURE): BOOLEAN =def
let sortid = oplitSig.definingSort in // the identifier of the sort (value data type) op defined in
if (oplitSig ∈ Literal-signature) ∨ ( isStructSort(sortid) = false)
then false
else // Operation-signature defined in struct sort
    let nt = oplitSig.opltName in // nt is Token (character string) for name
    (       (nt = "Make"
            // Formal arguments not checked
            ∧ opSig.s-Operation-signature.s1-Identifier = sortid // Operation result
            )
        ∨   (nt = "Undefined"
            ∧ opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
            ∧ opSig.s-Operation-signature.s1-Identifier = BooleanType // Operation result
            )
        ∨   (substring(nt, nt.length-5, 6) = "Modify"
            ∧ opSig.s-Operation-signature.s-Identifier-seq[1] = sortid  // Formal argument for original value
              // Formal argument for the new field value not checked
            ∧ opSig.s-Operation-signature.s1-Identifier = sortid // Operation result
            )
        ∨   (substring(nt, nt.length-6, 7) = "Extract"
            ∧ opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
            )
        ∨   (substring(nt, nt.length-6, 7) = "Present"
            ∧ opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
            ∧ opSig.s-Operation-signature.s1-Identifier = BooleanType// Operation result
            )
    ) // alternative operator names
    endlet // nt
endif // Literal-signature/not-struct or Operation-signature
endlet // sortid
```

The function *computeStruct* gives the value of applying the language-defined operators for structures. The "Make" operator does not require an existing SDLStructure value, only the identifier of the structure sort. The "Undefined", "Modify", "Extract" and "Present" operators all act on an SDLStructure value that is the head of the values passed as parameters.

```
computeStruct(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEOREXCEPTION =def
if oplitSig ∈ Literal-signature
then undefined
else // Operation-signature
    let structsort = oplitSig.definingSort in // the identifier of the sort (value data type) op defined in
    let nt = oplitSig.opltName in // nt is Token (character string) for op name
    let proc = oplitSig.s-Operation-signature.s2-Identifier.refersto1 in // proc for operation
    let procparams = proc.s-PROCPARAM-seq in
    if nt = "Make"
    then
        structMake(structsort,
            empty,
            <   mk-FIELD(procparams[i].s-Parameter.s-Name, undefined): i ∈ 1.. procparams.length
            >, // list of fields – names = param names of procedure for Make signature, values undefined
            values
        ) // structMake
    elseif nt = "Undefined"
    then    structUndefined(values.head) // values.head is the SDLStructure value
    elseif substring(nt, nt.length-5, 6) =  "Modify")
    then    structModify(nt, values.head, values.tail.head, empty, values.head.s-FIELD-seq)
        // values.head is the SDLStructure value, values.tail.head is the new field value
    elseif substring(nt, nt.length-6, 7) = "Extract"))
```

```
    then    structExtract(nt, values.head) // values.head is the SDLStructure value
    elseif substring(nt, nt.length-6, 7) = "Present"))
    then    structFieldPresent(nt, values.head) // values.head is the SDLStructure value
    else raise(OutOfRange) // should not occur
    endif
    endlet // procparams
    endlet // proc
    endlet // nt
    endlet // structsort
endif // Literal-signature or Operation-signature
```

The function *structMake* creates a structure value with the fields initialized to the list of values. It should be called externally (internally it is recursive) with a structure sort identifier, an empty list of new fields (*newflds*) and a list of old fields (*oldflds*) that each has a field name defined, and a list of one or more values. If the length of the list of values is less than the number of fields, the value list is extended with undefined items. The new fields (*newflds*) and old fields (*oldflds*) are used in the internal recursion.

```
structMake(structsort: Identifier, newflds: FIELD*, oldflds: FIELD*, values: VALUE*): SDLSTRUCTURE =def
    if values.length < oldflds.length then structMake(structsort, newflds, oldflds, values ⌢ undefined)
    elseif values.length = 0 ∨ oldflds.length = 0 then mk-SDLSTRUCTURE(newflds, structsort)
    else structMake(structsort,newflds ⌢ mk-FIELD( oldflds.head.s-
Name,values.head),oldflds.tail,values.tail)
    endif
```

The function *structUndefined* returns true if (and only if) all the fields are undefined.

```
structUndefined(struct: SDLSTRUCTURE): SDLBOOLEAN =def
    mk-SDLBOOLEAN(
        semvalueBool(∀ field ∈ FIELD : (field in struct.s-FIELD-seq ∧ field.s-VALUE  = undefined)),
     BooleanType) // mk SDLBoolean
```

The function *structModify* returns a new structure value with one field changed. It should be called externally (internally it is recursive) with the field name (as a token), a structure value, the new value for the field, an empty list of new fields (*newflds*) and a list of old fields (*oldflds*) that each have a field name defined. The new fields (*newflds*) and old fields (*oldflds*) are used in the internal recursion.

```
structModify(nt:TOKEN,struct:SDLSTRUCTURE,val:VALUE,newflds:FIELD*,oldflds:FIELD*):SDLSTRUCTURE =def
if oldflds.length = 0 then mk-SDLSTRUCTURE(newflds, struct.s-Identifier)
else
    structModify(fn, struct, val,
        newflds ⌢
            mk-FIELD(oldflds.head.s-Name,
                if oldflds.head.s-Name.s-TOKEN + "Modify" = nt
                then val
                else oldflds.head.s-VALUE
                endif
            ), // mk-Field
        oldflds.tail)
endif
```

The function *structExtract* returns the field with a given name from a list of fields.

```
structExtract(nt: TOKEN, struct: SDLSTRUCTURE): VALUE =def
let valueset = { f.s-VALUE : f in struct.s-FIELD-seq ∧ f.s-Name.s-TOKEN + "Extract" = nt} in
if valueset ≠ ∅ ∧ valueset.take ≠ undefined then valueset.take else raise(UndefinedField) endif
endlet // valueset
```

The function *structFieldPresent* returns true if the specified field has a value.

structFieldPresent(*nt*: *TOKEN*, *struct*: *SDLSTRUCTURE*): *SDLBOOLEAN* =def
let *valueset* = { *f*.**s**-*VALUE* : *f* **in** *struct*.**s**-*FIELD*-**seq** ∧ *f*.**s**-*Name*.**s**-*TOKEN* + "Present" = *nt*} **in**
mk-*SDLBOOLEAN*(*semvalueBool*(¬(*valueset* = ∅ ∨ *valueset.take* = *undefined*)), *BooleanType*)

F3.3.3.3 Choice

A choice value is identified by its type name, and the field list. The field names are a list, rather than a set so that a choice is similar to a structure with the constraint that only one field is present at any time, so that modifying one field makes all other field undefined. A choice is also characterised by having a "PresentExtract" operator that delivers a literal value for a literal constructor data type that uses the field names of the choice as literals.

SDLCHOICE =def *FIELD** × *Identifier*

The function *isChoiceSort* determines if a sort identifier refers to a value data type definition for a choice. Such a data type definition is characterised by having a "Make" operator, an "Undefined" operator, a "PresentExtract", and for each field operators with the field name concatenated with "Modify", "Extract" and "Present".

isChoiceSort(*sortid*: *Identifier*): *BOOLEAN* =def
let *dtd* = *sortid.refersto1* **in** // data type definition
if *dtd* ∉ *Value-data-type-definition*
then *false*
else // check Operation-signatures – true if all the following are true
(∃ *opSig* ∈ *dtd*.**s**-*Static-operation-signature*-**set**:
opSig.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN* = "PresentExtract"
∧ *opSig*.**s**-*Operation-signature*.**s**-*Identifier*-**seq** = < *sortid* > // Formal argument list
// Operation result is not checked – is "AnonPresent"
)
∧ (∃ *opSig* ∈ *dtd*.**s**-*Static-operation-signature*-**set**:
opSig.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN* = "Make"
∧ *opSig*.**s**-*Operation-signature*.**s1**-*Identifier* = *sortid* // Operation result
)
∧ (∃ *opSig* ∈ *dtd*.**s**-*Static-operation-signature*-**set**:
opSig.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN* = "Undefined"
∧ *opSig*.**s**-*Operation-signature*.**s**-*Identifier*-**seq** = < *sortid* > // Formal argument list
∧ *opSig*.**s**-*Operation-signature*.**s1**-*Identifier* = *BooleanType* // Operation result
)
∧ (∃ *opSig* ∈ *dtd*.**s**-*Static-operation-signature*-**set**:
substring(*opSig*.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN*,
opSig.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN*.*length*-5,
6) = "Modify"
∧ *opSig*.**s**-*Operation-signature*.**s**-*Identifier*-**seq**[1] = *sortid* // Formal argument for original value
// Formal argument for the new field value not checked
∧ *opSig*.**s**-*Operation-signature*.**s1**-*Identifier* = *sortid* // Operation result
)
∧ (∃ *opSig* ∈ *dtd*.**s**-*Static-operation-signature*-**set**:
substring(*opSig*.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN*,
opSig.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN*.*length*-6,
7) = "Extract"
∧ *opSig*.**s**-*Operation-signature*.**s**-*Identifier*-**seq** = < *sortid* > // Formal argument list
// Operation result not checked
)
∧ (∃ *opSig* ∈ *dtd*.**s**-*Static-operation-signature*-**set**:
substring(*opSig*.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN*,
opSig.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN*.*length*-6,
7) = "Present"
∧ *opSig*.**s**-*Operation-signature*.**s**-*Identifier*-**seq** = < *sortid* > // Formal argument list
∧ *opSig*.**s**-*Operation-signature*.**s1**-*Identifier* = *BooleanType*// Operation result
)
∧ (∀ *fn* ∈ *TOKEN* : (

```
            ((fn + "Modify") ∈ { opTok ∈ opSig.s-Operation-signature.s-Name.s-TOKEN
                : opSig ∈ dtd.s-Static-operation-signature-set } // set for opTok
            )
            ⇒
            ((fn + "Extract") ∈ { opTok ∈ opSig.s-Operation-signature.s-Name.s-TOKEN
                : opSig ∈ dtd.s-Static-operation-signature-set } // set for opTok
            )
            ∧
            ((fn + "Present") ∈ { opTok ∈ opSig.s-Operation-signature.s-Name.s-TOKEN
                : opSig ∈ dtd.s-Static-operation-signature-set } // set for opTok
            )
        )) // for all fn
    endif // dtd not member or member of Value-data-type-definition
    endlet // dtd
```

The function *isSpecialChoiceOp* determines if the operator given is one of the language defined choice operators.

```
isSpecialChoiceOp(oplitSig: OPLITSIGNATURE): BOOLEAN =def
let sortid = oplitSig.definingSort in // the identifier of the sort (value data type) op defined in
if (oplitSig ∈ Literal-signature) ∨ ( isChoiceSort(sortid) = false)
then false
else // Operation-signature defined in choice sort
    let nt = oplitSig.opltName in // nt is Token (character string) for name
    (       (nt = "Make"
            ∧   opSig.s-Operation-signature.s1-Identifier = sortid // Operation result
            )
        ∨   (nt = "Undefined"
            ∧   opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
            ∧   opSig.s-Operation-signature.s1-Identifier = BooleanType // Operation result
            )
        ∨   (substring(nt, nt.length-5, 6) = "Modify"
            ∧   opSig.s-Operation-signature.s-Identifier-seq[1] = sortid  // Formal argument for original value
                // Formal argument for the new field value not checked
            ∧   opSig.s-Operation-signature.s1-Identifier = sortid // Operation result
            )
        ∨   (substring(nt, nt.length-6, 7) = "Extract"
            ∧   opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
            ∧   opSig.s-Operation-signature.s1-Identifier = sortid // Operation result
            )
        ∨   (substring(nt, nt.length-6, 7) = "Present"
            ∧   opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
            ∧   opSig.s-Operation-signature.s1-Identifier = BooleanType// Operation result
            )
        ∨   nt  = "PresentExtract"
            ∧   opSig.s-Operation-signature.s-Identifier-seq = < sortid > // Formal argument list
            // Result is literal value from "AnonPresent"
            )
    ) // alternative operator names
    endlet // nt
endif // Literal-signature or Operation-signature
endlet // sortid
```

The function *computeChoice* gives the value of applying the language-defined operators for choice.

```
computeChoice(oplitSig: OPLITSIGNATURE, values: VALUE*): VALUEORECEPTION =def
if oplitSig ∈ Literal-signature
then undefined
else // Operation-signature
    let choicesort = oplitSig.definingSort in // the identifier of the sort (value data type) op defined in
    let nt = oplitSig.opltName in // nt is Token (character string) for op name
```

```
    if nt = "Make"
    then choiceMake(choicesort) // choiceMake
    elseif nt = "Undefined"
    then   choiceUndefined(values.head) // values.head is the SDLChoice value
    elseif substring(nt, nt.length-5, 6) =  "Modify")
    then   choiceModify(nt, values.head, values.tail.head)
        // values.head is the SDLChoice value, values.tail.head is the new field value
    elseif substring(nt, nt.length-6, 7) = "Extract"))
    then   choiceExtract(nt, values.head) // values.head is the SDLChoice value
    elseif substring(nt, nt.length-6, 7) = "Present"))
    then   choiceFieldPresent(nt, values.head) // values.head is the SDLChoice value
    elseif  nt = "PresentExtract"
    then   choicePresentExtract(values.head) // values.head is the SDLChoice value
    else raise(OutOfRange) // should not occur
    endif
    endlet // nt
    endlet // choicesort
endif // Literal-signature or Operation-signature
```

The function *choiceMake* creates a choice value with all the fields initialized to undefined.

```
choiceMake(choicesort: Identifier): SDLCHOICE =def
    mk-SDLCHOICE(
        < mk-FIELD(choicesort.choiceFieldlist[i], undefined) : i ∈ 1..choiceFieldlist.length >,
        choicesort
    ) // mk-SDLChoice
```

The function *choiceFieldlist* derives a list of field names from the "AnonPresent" sort that is the result sort of "PresentExtract". The names are in an arbitrary order dependent on the string construction from the literal signatures of AnonPresent, but this does not matter as long as the same order is always used.

```
choiceFieldlist(choicesort: Identifier): Name* =def
let opSigPE =
    { opSig ∈ choicesort.refersto1.s-Static-operation-signature-set
    :       opSig.s-Operation-signature.s-Name.s-TOKEN = "PresentExtract"
        ∧   opSig.s-Operation-signature.s-Identifier-seq = < choicesort > // Formal argument list
    }.take
in
let litSigSet = opSigPE.s-Operation-signature.s1-Identifier.refersto1.s-Literal-signature-set in
    // set of literal signatures in AnonPresent definition
names =
    <   litSig.s-Name : litSig ∈ litSigSet
        ^ ((i ∈ Integer ∧ j ∈ Integer ∧ 1 ≤ i ∧ j ≤ | litSigSet | ∧ i < j ) ⇒  names[i] < names[i] )
    > // names sorted
endlet // litSigSet
endlet // opSigPE
```

The function *choiceUndefined* returns true if (and only if) all the fields are undefined.

```
choiceUndefined(choice: SDLCHOICE): SDLBOOLEAN =def
    mk-SDLBOOLEAN(
        semvalueBool(∀ field ∈ FIELD : (field in choice.s-FIELD-seq ∧ field.s-VALUE  = undefined)),
     BooleanType) // mk SDLBoolean
```

The function *choiceModify* returns a new choice value with one field set to the value for that field and other fields undefined.

```
choiceModify(nt: TOKEN, choice: SDLCHOICE, val: VALUE): SDLCHOICE =def
    mk-SDLCHOICE(
        <   mk-FIELD( choice.s-FIELD-seq[i].s-Name,
                if   choice.s-FIELD-seq[i].s-Name.s-TOKEN + "Modify" = nt
                then   val
```

```
            else    undefined
            endif
        ) // mk-Field
        : i ∈ 1..choice.s-FIELD-seq.length
    > // list of fields
    choice.s-Identifier
) // mk-SDLChoice
```

The function *choiceExtract* returns the field with a given name from a list of fields.

```
choiceExtract(nt: TOKEN, choice: SDLCHOICE): VALUE =def
let valueset = { f.s-VALUE : f in choice.s-FIELD-seq ∧ f.s-Name.s-TOKEN + "Extract" = nt} in
if valueset = ∅ ∨ valueset.take = undefined then raise(UndefinedField) else valueset.take endif
endlet // valueset
```

The function *choiceFieldPresent* returns true if the specified field has a value.

```
choiceFieldPresent(nt: TOKEN, choice: SDLCHOICE): SDLBOOLEAN =def
let valueset = { f.s-VALUE : f in choice.s-FIELD-seq ∧ f.s-Name.s-TOKEN + "Present" = nt} in
mk-SDLBOOLEAN(semvalueBool(¬(valueset = ∅ ∨ valueset.take = undefined)), BooleanType)
endlet // valueset
```

The function *choicePresentExtract* returns the *VALUE* that is an *SDLLITERAL* for the *Literal-signature* of the AnonPresent data type for the field that is present in an *SDLCHOICE* value.

```
choicePresentExtract(choice: SDLCHOICE): SDLLITERAL =def
let choicesort = choice.s-Identifier
let opSigPE =
    { opSig ∈ choicesort.refersto1.s-Static-operation-signature-set
    :       opSig.s-Operation-signature.s-Name.s-TOKEN = "PresentExtract"
        ∧   opSig.s-Operation-signature.s-Identifier-seq = < choicesort > // Formal argument list
    }.take
in
let anonPresId = opSigPE.s-Operation-signature.s1-Identifier // AnonPresent identifier
let litSigSet = anonPresId.refersto1.s-Literal-signature-set
    // set of literal signatures in AnonPresent definition
in
mk-SDLLITERAL(
    {litSig ∈ litSigSet : litSig.s-Name = f.s-Name ∧ f in choice.s-FIELD-seq ∧ f.s-VALUE ≠ undefined }.take,
    anonPresId
) // SDLLiteral
endlet // litSigSet
endlet // anonPresId
endlet // opSigPE
endlet // choicesort
```

F3.3.4 Variables and other items with Aggregation-kind REF

In AS1, *Aggregation-kind* is either **PART** or **REF** (where no aggregation kind is given in AS0 it is mapped to **PART** as the default in F2). The AS1 item *Aggregation-kind* is used in parameters (*Agent-formal-parameter*, *In-parameter*, *Inout-parameter*, *Out-parameter* and *Signal-parameter*), *Result* (of *Literal-signature* and *Procedure-definition*) and *Variable-definition*. The *Aggregation-kind* is relevant for assignments (to a variable in *Assignment*, or to formal parameters from an *Actual-parameters*), for *Value-return-node* (depending on the returning procedure *Result-aggregation*) and for *Equality-expression*.

F3.3.5 State access

The *STATE* domain consists of a set of *NAMEDVALUE* items, and a set of super states (associations between super state and substate). In case a certain variable is bound to an in/out parameter in a substate, it refers to the variable in the caller's state.

STATE =$_{\text{def}}$ *NAMEDVALUE*-**set** × *SUPERSTATE*-**set**

A *NAMEDVALUE* is an association a specific variable *Identifier* and a specific *STATEID* with an optional *BOUNDVALUE*. If the variable is undefined the *BOUNDVALUE* is omitted.

NAMEDVALUE =$_{\text{def}}$
{ (*s*, *id*, *bv*) ∈ *STATEID* × *Identifier* × [*BOUNDVALUE*]
: *id.refersto1* ∈ *VAR*
∧ (∀*n1*, *n2* ∈ *NAMEDVALUE* : *n1*.**s**-*STATEID* = *n2*.**s**- *STATEID* ⇒ *n1*.**s**- *Identifier* ≠ *n2*.**s**- *Identifier*)
// for a given state, each identifier should occur only once
}

A *BOUNDVALUE* is a *VALUE* or an *Identifier* that refers to a variable (that is: a variable definition or parameter) or a *FIELDREF* that refers to a field of a variable (that is: a field of a structure/choice variable definition or parameter) or a *Null-literal-signature* or the *nullPid* value.

BOUNDVALUE =$_{\text{def}}$
{ (*bv*) ∈ *VALUE* ∪ *Identifier* ∪ *FIELDREF* ∪ *Literal-signature* ∪ *nullPid*:
bv ∈ *Identifier* ⇒ *bv.refersto1* ∈ *VARDEF*
∧ *bv* ∈ *Literal-signature* ⇒
(*bv*.**s**-*Name* = "null"
∧ *bv*.**s**-*Result*.**s**-*Result-aggregation* = **REF**
) // Null-literal-signature
}

A *FIELDREF* is a *BOUNDVALUE* that refers to a field of a variable that has a structure/choice sort or to the field of a structure/choice parameter, where the field has the specified name.

FIELDREF =$_{\text{def}}$ // More study is required to allow field references other than the field of a variable
{ (*bid*, *bfn*) ∈ *Identifier* × *Name*
: *bid* ∈ *Identifier* ⇒ *bid.refersto1* ∈ *VARDEF*
∧ (*bid.refersto1*.**s**-*Sort-reference-identifier.isStructSort*
∨ *bid.refersto1*.**s**-*Sort-reference-identifier.isChoiceSort*
) // variable/parameter has a struct/choice sort
∧ (∃ *opSig* ∈ *bid.refersto1*.**s**-*Sort-reference-identifier.refersto1*.**s**-*Static-operation-signature*-**set**:
opSig.**s**-*Operation-signature*.**s**-*Name*.**s**-*TOKEN* = *bfn*+"Modify"
) // struct/choice sort has field with name bfn
}

A *SUPERSTATE* is a pair of *STATEID* state identifiers, where the first *STATEID* is the identifier of the super state of the state identified by the second *STATEID*.

SUPERSTATE =$_{\text{def}}$ *STATEID* × *STATEID*

The function *initAgentState* provides the initial value of the state (see clause F3.2.1.3.1 Functions provided by the data type part). The parameters are: *state* – state of the outermost process agent (undefined if the outermost process agent is being created); *newid* – state identifier of the new state; *id* – state identifier of the super state of the new state (undefined for the outermost agent); *vars* – variables of the agent.

initAgentState(*state*: [*STATE*], *newid*: *STATEID*, *id*: [*STATEID*], *vars*: *DECLARATION*-**set**): *STATE* =$_{\text{def}}$
mk-*STATE*(
(**if** *state* = *undefined* **then** ∅ **else** *state*.**s**-*NAMEDVALUE*-**set endif**
∪
initDeclarations(*newid*, *vars*)
), // NamedValue-set for mk-State
(**if** *state* = *undefined* **then** ∅ **else** *state*.**s**-*SUPERSTATE*-**set endif**
∪
if *id* = *undefined* **then** ∅ **else** { **mk**-*SUPERSTATE*(*id*, *newid*)} **endif**
) // SuperState-set for mk-State
) // mk-State

The function *initProcedureState* provides the initial value of the state (see clause F3.2.1.3.1 Functions provided by the data type part). The parameters are: *state* – state of the outermost process agent; *newid* – state identifier of the new state; *id* – state identifier of the super state of the new state; *declarations* – variables defined in the procedure; *fparams* – formal parameters of the procedure; *values* – list of actual parameter values for the procedure.

```
initProcedureState(state: STATE, newid: STATEID, id: STATEID, vars: DECLARATION-set,
fparams:PROCPARAM , ap:CALLPARAM*): STATEOREXCEPTION =def
if checkParamsRange(fparams, ap)  = false // at least one In-parameter out of range
then   raise(OutOfRange)
else // every In-parameter item with a syntype has an actual parameter value in range
   mk-STATE(
      (   state.s-NAMEDVALUE-set
          ∪
          assignParams(initDeclarations(newid, vars ∪ fparams.toSet), newid, fparams, ap)
      ), // NamedValue-set for mk-State
      state.s-SUPERSTATE-set ∪ { mk-SUPERSTATE(id, newid)  } // SuperState-set for mk-State
   ) // mk-State
```

The function *checkParamsRange* checks for each *In-parameter* that has a syntype, the actual parameter value is in range for the syntype.

```
checkParamsRange(fparams: PROCPARAM*, ap: CALLPARAM*): BOOLEAN =def
if      fparams =  empty then true // checked true for all params
elseif  fparams[1] ∈ In-parameter
   ∧    fparams[1].s-Parameter.s-Identifier.refersto1 ∈ Syntype-definition
   ∧    ap[1] ∈ VALUELABEL
   ∧    rangeCheck(fparams[1].s-Parameter.s-Identifier.refersto1, value(ap[1], Self))=false
then    false
else    checkParamsRange(fparams.tail, ap.tail) // check next parameter
endif
```

The function *initDeclarations* provides the initial *NAMEDVALUE* binding set for a declarations set within a state. The parameters are: *stid* – state identifier of the state; *declarations* set for the declarations set.

```
initDeclarations(stid: STATEID, declarations: DECLARATION-set): NAMEDVALUE-set =def
{   mk-NAMEDVALUE(stid, d.identifier1,
        if d.s-EXPRESSION.isConstant1 then d.s-EXPRESSION  else defaultInit(d.s-Identifier) endif
    ) // mk-NamedValue
|   d ∈ declarations ∧ d ∈ Variable-definition
}
∪
{   mk-NAMEDVALUE(stid, d.s-Parameter.identifier1, defaultInit(d.s-Parameter.s-Identifier)
    ) // mk-NamedValue
|   d ∈ declarations ∧ d ∈ PROCPARAM
}
```

The function *defaultInit* returns the default initialization for the sort reference identifier given as a parameter. If the sort reference identifier is for a value data type definition that has a default initialization, this is returned. Otherwise, if the value data type definition has no default initialization and inherits from another data type, the default initialization of this inherited data type is returned. If the value data type definition has no default initialization and does not inherit from another data type, undefined is returned. If the sort reference identifier is for a syntype definition that has a default initialization, this is returned. Otherwise, if the syntype definition has no default initialization, the default initialization of the parent data type is returned.

```
defaultInit(sortRefId: Identifier): EXPRESSION =def
if sortRefId.refersto1 ∈ Value-data-type-definition
then
```

```
    if sortRefId.refersto1.s-EXPRESSION ≠ undefined // Expression is default initialization
    then sortRefId.refersto1.s-EXPRESSION // Expression is default initialization
    else // no local default initialization
        if sortRefId.refersto1.s-Identifier ≠ undefined // inherits from another data type
        then defaultInit(sortRefId.refersto1.s-Identifier)
        else undefined
        endif // inherits from another data type
    endif // local default initialization or not
else // syntype
    if sortRefId.refersto1.s-EXPRESSION ≠ undefined // Expression is default initialization
    then sortRefId.refersto1.s-EXPRESSION // Expression is default initialization
    else defaultInit(sortRefId.refersto1.s-Identifier) // parent default initialization
    endif // local default initialization or not
endif // value data type or syntype
```

The function *assignParams* puts a sequence of actual parameter values into the named values set for a given state id for *In-parameter* and *Inout-parameter* items. The named value for an *In-parameter* has a bound value given as the actual parameter, the named value for an *Inout-parameter* has a bound value that is the given variable identifier, and the named value for an *Out-parameter* remains undefined from *initDeclarations*. There is no need to check that the actual parameters are in range because this is checked in function *initProcedureState*.

```
assignParams(namedvalues:NAMEDVALUE-set, id: STATEID,
fparams: PROCPARAM*, ap: CALLPARAM*): NAMEDVALUE-set =def
if ap = empty ∨ fparams = empty
then
    namedvalues // set unchanged
else
    assignParams(
    if fparams.head ∉ Out-parameter // In or Inout – set value
    then
        setValue(namedvalues, id, fparams.head.s-Parameter.identifier1,
            if ap.head ∈ VALUELABEL then value(ap.head, Self) else ap.head endif) // setValue
    else    namedvalues // Out-parameter, leave named value undefined
    endif, // namedvalues
    id, fparams.tail, values.tail) // assignParams
endif
```

The function *setValue* puts a single value into a named values set for a given state *stid*.

```
setValue(namedvalues: NAMEDVALUE-set, stid: STATEID, varid:Identifier, valueOrId:BOUNDVALUE):
NAMEDVALUE-set =def
{ binding | binding ∈ namedvalues: binding.s-STATEID ≠ stid ∨ binding.s-Identifier ≠ varid }
∪
{ mk-NAMEDVALUE(stid, varid, valueOrId) }
```

The function *getValueSet* returns the association between the state *stid* and *varid* in *namedvalues*. There should be at most one such association, but the function returns a set and an empty set indicates there was no association.

```
getValueSet(namedvalues: NAMEDVALUE-set, stid: STATEID, varid:Identifier): NAMEDVALUE-set =def
{ b ∈ namedvalues: b.s-STATEID = stid ∧ b.s-Identifier = varid}
```

The function *eval* is called to evaluate a variable access (see EVALVAR in F3.2.1.4.29) in a contexts where **REF** aggregation is not required, and returns the value associated with an identifier for a given a state and state id. There can be at most one named value for the state and identified item (variable or parameter). If this named value has a bound value that is a value, this is the result. Otherwise, if the bound value is a variable identifier, this bound variable is an item (variable or parameter). In this case *eval* is called recursively to return the value (in the named values for the state) for the bound variable and the caller (the state id that caused this state id to exist). Otherwise,

if the bound value is a field reference: first the variable containing the field is evaluated by calling *eval* with the *Identifier* of the bound reference; then the field is extracted from the structure or choice value of the variable. Otherwise the bound value is *undefined*, and *undefined* is returned.

```
eval(varid:Identifier, state:STATE, stid:STATEID): VALUEOREXCEPTION =def
    let callerid = caller(state, stid) in
    let namedval = getValueSet(state.s-NAMEDVALUE-set, stid, varid) in
    if namedval ≠ ∅ then
        case bv = namedval.take.s-BOUNDVALUE of
        | VALUE       then bv // varid bound to a value
        | Identifier  then // varid bound to variable id for variable definition or parameter
            eval(bv, state, stid) // evaluate referenced id
        | FIELDREF    then
            if varid.refersto1.s-Identifier.isStructSort // check if struct, otherwise choice
            then structExtract(bv.s-Name + "Extract", eval(bv.s-Identifier, state, stid)) // extract field of var
            else choiceExtract(bv.s-Name + "Extract", eval(bv.s-Identifier, state, stid)) // extract field of var
            endif
        | Literal-signature then // if a BoundValue is a Literal-signature, it is a Null-literal-signature
            raise(InvalidReference) // attempt to dereference a Null-literal-signature
        otherwise raise(UndefinedVariable) // the BOUNDVALUE is undefined
        endcase // bv boundvalue
    elseif callerid ≠ undefined // there is a calling state – false if the system state
        eval(varid, state, callerid) // not a local id – evaluate in calling state
    else // id not found
        raise(UndefinedVariable)
    endif
    endlet // namedval
    endlet // callerid
```

The function *update* modifies a binding of a variable identifier to a value or identifier or field reference or literal signature in the named values set for a given state *id*.

```
update(varid:Identifier, bvalue: BOUNDVALUE, state:STATE, id:STATEID): STATE =def
if getValueSet(state.s-NAMEDVALUE-set, id, varid)  = ∅ // not a local variable
then   update(varid, bvalue, state, caller(state, id)) // update in calling state
else mk-STATE(setValue(state.s-NAMEDVALUE-set, id, varid, bvalue), state.s-SUPERSTATE-set)
endif
```

The function *assign* modifies the variable with the given name in the state/id association to the given value.

```
assign (variableid:Identifier, value:VALUE, state:STATE, id:STATEID): STATEOREXCEPTION =def
if isValueVariable(variableid) then
    if isSyntypeVariable(variableid) ∧ ¬rangeCheck(variableid.variableSort, value )
    then   raise(OutOfRange)
    else update(variableid, mk-BOUNDVALUE(value), state, id)
    endif
else
    // pid variable, sort of variable is an Interface-definition
    if variableid.variableSort = value.interface ∨
        isSuperType(variableid.variableSort, value.interface)
    then   update(variableid, mk-BOUNDVALUE(value), state, id)
    else
        update(variableid, mk-BOUNDVALUE(nullPid), state, id)
    endif
endif
```

The function *caller* returns the state id that caused this state id to exist.

```
caller(state: STATE, id: STATEID): STATEID =def
take({ s.s1-STATEID | s ∈ state.s-SUPERSTATE-set: s.s2-STATEID = id})
```

The function *variableSort* returns the sort for a given variable identifier.

variableSort(*variableid*: *Identifier*):*Interface-definition* ∪ *Value-data-type-definition* ∪ *Syntype-definition*=$_{def}$
variableid.refersto1.**s**-*Identifier.refersto1* // Sort-reference-identifier

The predicate *isValueVariable* holds if the given variable *Identifier* refers to a variable of a value type.

isValueVariable(*variableid*: *Identifier*): *BOOLEAN* =$_{def}$
variableid.variableSort ∈ *Value-data-type-definition*

The predicate *isSyntypeVariable* holds if the given variable *Identifier* refers to a variable with a syntype.

isSyntypeVariable(*variableid*: *Identifier*): *BOOLEAN* =$_{def}$
variableid.variableSort ∈ *Syntype-definition*

The function *interface* for a value, delivers the *Interface-definition* for the value if the sort of the value is an *Interface-definition*, otherwise the result is *undefined*.

interface(*val*: *VALUE*): *Interface-definition* =$_{def}$
if *val.sort* ∈ *Interface-definition* **then** *val.sort* **else** *undefined* **endif**

The function *sort* for a value gives the sort of a value, which for most domains (such as *SDLBOOLEAN* or *SDLSTRUCTURE* that form part of the *VALUE* domain) is found from the *Identifier* element of the domain. The exception is the *PID* domain, which instead is either a *NULLPID* that has the value *nullPid*, and is a *PidType* value, or is a *VALIDPID* with an optional *Interface-definition*. In the case of a *VALIDPID* without an *Interface-definition*, the value is a *PidType* value; otherwise the data type definition is the *Interface-definition*.

sort(*val*: *VALUE*): *Interface-definition* ∪ *Value-data-type-definition* =$_{def}$
if *val* ∈ *NULLPID* **then** *PidType.refersto1*
elseif *val* ∈ *VALIDPID* **then**
  if *val*.**s**-*Interface-definition* = *undefined* **then** *PidType.refersto1* **else** *val*.**s**-*Interface-definition* **endif**
else *val*.**s**-*Identifier.refersto1*
endif

F3.3.6 Specialization

The function *dynamicType* determines the identity of the dynamic type of a value.

dynamicType(*v*: *VALUE*): *Identifier* =$_{def}$
if *v* = *nullPid* **then** *raise*(*OutOfRange*) **else**
  case *v* **of**
  | *PID*(*, *t*) **then** *t*
  | *SDLARRAY*(*, *, *, *t*) **then** *t*
  | *SDLBAG*(*, *t*) **then** *t*
  | *SDLBIT*(*, *t*) **then** *t*
  | *SDLBITSTRING*(*, *t*) **then** *t*
  | *SDLBOOLEAN*(*,*t*) **then** *t*
  | *SDLCHARACTER*(*, *t*) **then** *t*
  | *SDLCHARSTRING*(*, *t*) **then** *t*
  | *SDLCHOICE*(*, *t*) **then** *t*
  | *SDLDURATION*(*, *t*) **then** *t*
  | *SDLINTEGER*(*, *t*) **then** *t*
  | *SDLLITERAL*(*,*t*) **then** *t*
  | *SDLOCTETSTRING*(*, *t*) **then** *t*
  | *SDLPOWERSET*(*, *t*) **then** *t*
  | *SDLREAL*(*,*, *t*) **then** *t*
  | *SDLSTRING*(*,*t*) **then** *t*
  | *SDLSTRUCTURE*(*,*t*) **then** *t*

```
    | SDLTIME(*, t)          then t
    | SDLVECTOR(*, t)        then t
    endcase
endif
```

F3.3.7 Operators and methods

The function *dispatch* determines the procedure to select, given a set of actual parameters.

```
dispatch(oplitSig:OPLITSIGNATURE, values:VALUE*): Identifier =def
case oplitSig of
| Literal-signature then undefined // should not occur
| Static-operation-signature then oplitSig.s-Identifier // identifier of derived procedure
| Dynamic-operation-signature then
    bestMatch(matchingCandidates(allDynamicCandidates(oplitSig), values))
endcase
```

The function *allDynamicCandidates* returns the set of all signatures with the same name as the given signature.

```
allDynamicCandidates(oplitSig: Dynamic-operation-signature): OPLITSIGNATURE-set =def
{ p | p ∈ Operation-signature: p.s-Name = oplitSig.s-Operation-signature.s-Name }
```

The function *matchingCandidates* returns the set of all signatures for which the formal argument *Identifier* list is compatible with the argument values.

```
matchingCandidates(procedures: (Static-operation-signature ∪ Dynamic-operation-signature)-set,
values: VALUE*): OPLITSIGNATURE-set =def
{ p ∈ procedures: isSignatureCompatible(p.s-Operation-signature.s-Identifier-seq, dynamicTypes(values)) }
```

The function *bestMatch* returns the most specialized signature, that is the signature that is signature compatible with all the other procedures in the set.

```
bestMatch(oplitSigs:OPLITSIGNATURE-set): Identifier =def
take({ oplitSig.s-Identifier | oplitSig ∈ oplitSigs:
    (∀ q ∈ oplitSigs :
        isSignatureCompatible(
            oplitSig.s-Operation-signature.s-Identifier-seq,
            q.s-Operation-signature.s-Identifier-seq
        )
    ) // for all q
})
```

The predicate *isSignatureCompatible* holds if p is compatible with q.

```
isSignatureCompatible(p:Identifier*, q:Identifier*): BOOLEAN =def
if p = empty then
    true
else
    isSortCompatible(p.head, q.head) ∧
    isSignatureCompatible(p.tail, q.tail)
endif

isSortCompatible(p: Identifier, r: Identifier): BOOLEAN =def
    (p = r )
∨   isDirectlySortCompatible(p, r)
∨   (       r.refersto1 ∈ Interface-definition
    ∧   (∃ q ∈ Identifier: (isSortCompatible(p, q) ∧ isSortCompatible(q, r)))
    )

isDirectlySortCompatible(y: Identifier, z: Identifier): BOOLEAN =def
if isSuperSort(z, y)
then
```

```
    if y.refersto1 ∈ Value-data-type-definition
    then    // true if y is <anchored sort> of the form this z
        y.refersto1.s-Identifier = z
    else    // y is a pid sort (because not a value data type) – and z is super sort of y
        true
    endif
else    false
endif

isSuperSort(z: Identifier, y: Identifier): BOOLEAN =def
isSuperType(z, y) // see clause F2.2.1.6.4.

dynamicTypes(values:VALUE*): Identifier* =def
< dynamicType(v) | v in values >
```

F3.3.8 Syntypes

The predicate *rangeCheck* holds if the range check for a value of a *syntype* passes: that is, there exists a condition in the set of range conditions of the syntype that includes the given value. The condition check is made with respect to the parent type of the syntype.

```
rangeCheck(syntype: Syntype-definition, value: VALUE): BOOLEAN =def
∃ cond ∈ syntype.s-Range-condition: conditionItemCheck(cond, value, syntype.s-Identifier)
```

The predicate *conditionItemCheck* holds if the condition is true for the value of the given type. If the condition is a size constraint, rewriting the concrete grammar creates an anonymous operation identified by the *Operation-identifier* of the *Size-constraint* that embodies the ranges specified, so the *Open-range* or *Closed-range* items in the abstract grammar of *Size-constraint* are redundant. An alternative would be to construct an anonymous procedure here based on the *Open-range* or *Closed-range* items of *Size-constraint*, in which case the *Operation-identifier* of *Size-constraint* is redundant.

```
conditionItemCheck(cond: Open-range ∪ Closed-range ∪ Size-constraint, value: VALUE, type: Identifier):
BOOLEAN =def
if cond ∈ Open-range then
    semvalueBool(compute(cond.s-Open-range.s-Identifier, // Operation-identifier
        < cond.s-Open-range.s-EXPRESSION >)) // the expression of an Open-range is a constant expression
elseif cond ∈ Closed-range then
    choose lessthaneq:
        ( lessthaneq ∈ type.s-Static-operation-signature-set ∧ lessthaneq.opltName = """<=""" )
            semvalueBool(compute(lessthaneq, < cond.s-Closed-range.s1-EXPRESSION, value > ))
        ∧   semvalueBool(compute(lessthaneq, < value, cond.s-Closed-range.s2-EXPRESSION >))
            // both expressions of a Closed-range are constant expressions
    endchoose
else //size constraint and cond ∈ Size-constraint
    semvalueBool(compute(cond.s-Size-constraint.s-Identifier, < value >)) // Operation-identifier
endif
```

Appendix I
to Annex F3

List of abstract syntax grammar rules used

This list contains the Specification and Description Language abstract syntax grammar rules that are used in this annex (Annex F3). The complete list of abstract syntax grammar rules can be found in Annex A of Recommendation ITU-T Z.100, which also identifies the Recommendation ([ITU-T Z.101] or [ITU-T Z.102] or [ITU-T Z.104] or [ITU-T Z.107]) where the grammar rule is defined. The list below only includes constructors (defined with "::"). Non-constructors (defined with "=") are not used in the formalisation and instead the appropriate constructor (for example *Identifier* instead of *Agent-identifier*) or a union of constructors (for example, *Interface-definition* ∪ *Value-data-type-definition* instead of *Data-type-definition*). In the special case of the non-constructor *Expression* a domain *EXPRESSION* is defined. *Exception-identifier* is only defined in this annex (Annex F3) and is not defined or used in [ITU-T Z.101] or [ITU-T Z.102] or [ITU-T Z.104] or [ITU-T Z.107].

Abstract

Action-return-node

Active-agents-expression

Actual-parameters

Agent-definition

Agent-instance

Agent-instance-pid-value

Agent-kind

Agent-qualifier

Agent-type-definition

Agent-type-qualifier

Aggregation-kind

Any-decision

Any-expression

Assignment

Break-node

Call-node

Channel-definition

Channel-path

Closed-range

Composite-state-graph

Composite-state-type-definition

Compound-node

Compound-node-qualifier

Conditional-expression

Connect-node

Continue-node

Continuous-signal

Create-request-node

Dash-nextstate

Data-type-qualifier

Decision-answer

Decision-body

Decision-node

Decoding-expression

Dynamic-operation-signature

Else-answer

Encoded-expression

Encoding-expression

Encoding-path

Encoding-rules

Entry-connection-definition

Exception-identifier

Exit-connection-definition

Free-action

Gate-definition

Graph-node

Identifier

In-choice

In-parameter

Inout-parameter

Inner-entry-point

Inner-exit-point

Input-node

Interface-definition

Interface-qualifier

Join-node

Literal

Literal-signature

Name

Named-nextstate

Named-return-node

Named-start-node

Negative-equality-expression

Nextstate-parameters

Now-expression

Number-of-instances

Offspring-expression

Open-range

Operation-application

Operation-signature

Out-parameter

Outer-entry-point

Outer-exit-point

Output-node

Package-definition

Package-qualifier

Parameter

Parameter-aggregation

Parent-expression

Positive-equality-expression

Procedure-definition

Procedure-graph

Procedure-qualifier

Procedure-start-node

Range-check-expression

Range-condition

Reset-node

Result

Result-aggregation

Save-signalset

Self-expression

Sender-expression

Set-node

Signal-definition

Signal-destination

Signal-expression

Signal-parameter

Signallist-expression

Size-constraint

Spontaneous-transition

State-expression

State-aggregation-node

State-machine

State-node

State-partition

State-qualifier

State-start-node

State-timer

State-type-qualifier

State-transition-graph

Static-operation-signature

Stop-node

Syntype-definition

Terminator

Timer-active-expression

Timer-remaining-duration

Timer-definition

Transition

Type-check-expression

Type-coercion

Value-data-type-definition

Value-return-node

Value-returning-call-node

Variable-access

Variable-definition

The qualifiers *Agent-qualifier*, *Agent-type-qualifier*, *Compound-node-qualifier*, *Interface-qualifier*, *Procedure-qualifier*, *State-qualifier*, *State-type-qualifier* are constructors, each of which is defined as a *Name*.

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