International Telecommunication Union

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100
Annex F3
(01/2015)

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

International Telecommunication Union

ITU-T Z-SERIES RECOMMENDATIONS

LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

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

Recommendation ITU-T Z.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
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

* 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.

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

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.

Keywords **derived**, **domain**, **static**, **initially**, **controlled**, **monitored**, **shared**, **constraint**, **let**, **endlet**, **where**, **endwhere**, **choose**, **endchoose**, **extend**, **with**, **endextend**, **case**, **of**, **endcase**, **do**, **forall**, **enddo**, **if**, **then**, **else**, **elseif**, **endif**.

The domains *TIME*, *AGENT*, *X*, *BOOLEAN*, *NAT*, *REAL*, *TOKEN*, *DEFINITIONAS1*.

The functions *take*, *program*, *Self*, *undefined*, *true*, *false*, *empty*, *head*, *tail*, *last*, *length*, *toSet*, *parentAS1*, *parentAS1ofKind*, *rootNodeAS1*.

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

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

F3.1.2 Definitions from Annex F2

$\mathit{ENTITYDEFINITION}_1$: the union of all the entity definitions in AS1. It is therefore a subset of *DEFINITIONAS1*.

$\mathit{ENTITYDEFINITION}_1 =_{\mathrm{def}}$ *Agent-definition*
∪ *Agent-type-definition*
∪ *Channel-definition*
∪ *Composite-state-type-definition*
∪ *Data-type-definition*
∪ *Gate-definition*
∪ *Literal-signature*
∪ *Operation-signature*
∪ *Package-definition*
∪ *Procedure-definition*
∪ *Signal-definition*
∪ *State-node*
∪ *Syntype-definition*
∪ *Timer-definition*
∪ *Variable-definition*

Given an *Identifier*, the corresponding $\mathit{ENTITYDEFINITION}_1$ is retrieved using the function *idToNodeAS1*:

$\mathit{idToNodeAS1}(\mathit{id}: \mathit{Identifier}): [\mathit{ENTITYDEFINITION}_1] =_{\mathrm{def}}$
$\mathit{getEntityDefinition}_1(\mathit{id}, \mathit{idKind}_1(\mathit{id}))$

where

function $\mathit{getEntityDefinition}_1$ from Annex F2 gets the entity definition for an identifier:

getEntityDefinition$_1$: *Identifier*, ENTITYKIND$_1$→ENTITYDEFINITION$_1$

and function *idKind*$_1$ from Annex F2 is used determine the kind of the entity from the identifier:

idKind$_1$:*Identifier*→ENTITYKIND$_1$

Given a ENTITYDEFINITION$_1$, the corresponding *Identifier* is retrieved using the function *identifier*$_1$ from Annex F2:

identifier$_1$: ENTITYDEFINITION$_1$→ *Identifier*

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

isSuperType: ENTITYDEFINITION$_1$ × ENTITYDEFINITION$_1$ → BOOLEAN

F3.1.3 Status of Annex F3 (this annex)

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

As noted in clause F1.2.4 (d) (Annex F1), the *data semantics* is separated from the rest of the dynamic semantics, which allows the data model to be changed. The current document is based on the previous edition (01/2000) that described the **object** data types of SDL-2000. The document has been considerably reduced by the removal of **object** data types, user exception definitions, user exception raising and exception handling.

The previous edition (01/2000) included a clause "4 Example", where an example specification and its expansion into an abstract syntax tree were given, but the results of initialization and compilation of the example had "TBD" sentences, meaning work was still to be done. In this state the example is not useful for the illustrating application of the dynamic semantics, and it has been removed from this edition.

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.

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

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 $PLAINSIGNALINST = \varnothing$

shared $plainSignalType: PLAINSIGNALINST \rightarrow PLAINSIGNAL$

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 $si \in PLAINSIGNALINST$ **then** $si.plainSignalType$
  elseif $si \in TIMERINST$ **then** $si.$**s-**$TIMER$
  endif

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

shared $plainSignalSender: PLAINSIGNALINST \rightarrow PID$

$signalSender(si{:}SIGNALINST){:}\ PID =_{\text{def}}$
  if $si \in PLAINSIGNALINST$ **then** $si.plainSignalSender$
  elseif $si \in TIMERINST$ **then** $si.$**s-**PID
  endif

With each signal a (possibly empty) list of *signal values* is associated. Because the type information and concrete value for signal values is immaterial to the dynamic aspects considered here, values are abstractly represented in a uniform way as elements of the static domain *VALUE* (see clause F3.2.1.3):

shared $plainSignalValues: PLAINSIGNALINST \rightarrow VALUE^*$

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**

$TOARG =_{\text{def}} PID \cup Identifier$

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*

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.

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 $now \geq si.arrival$. Intuitively, that means that *s* remains "invisible" at *g* as long as it is in transit. Thus, in every given SAM state, the visible 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 visible part of *g* is denoted as *g.queue* and formally defined as follows.

$$queue(g\text{: }\textit{GATE})\text{: }\textit{SIGNALINST}^* =_{\text{def}} <\ si \ \mathbf{in}\ g.schedule\text{: }(now \geq si.arrival)\ >$$

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

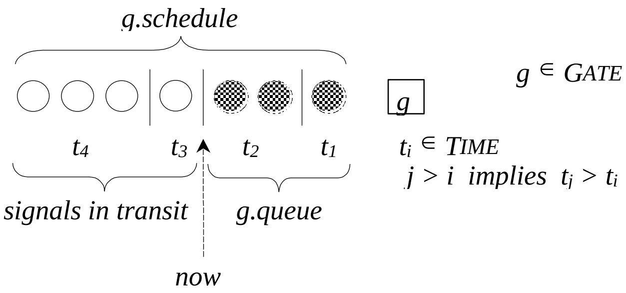

Figure F3-1 – Signal instances at a gate

Operations on schedules

To ensure that the order on signals is preserved when new signals are added to the schedule of a gate, there is a special insertion function *insert* on schedules.

insert(*si*:SIGNALINST, *t*:TIME, *siSeq*:SIGNALINST*): SIGNALINST* $=_{\text{def}}$
    if *siSeq* = *empty* **then**
      $<si>\frown siSeq$
    elseif $t < siSeq.head.arrival$
    then $<si>\frown siSeq$
    else $<siSeq.head>\frown insert(si, t, siSeq.tail)$
    endif

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

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

The macros INSERT and DELETE update the schedule of a gate *g* by assigning some new signal list to *g.schedule*.

INSERT(*si*:*SIGNALINST*, *t*:*TIME*, *g*:*GATE*) ≡
g.*schedule* := *insert*(*si*,*t*,*g*.*schedule*)
si.*arrival* := *t*+*si*.*delay*

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

The function *nextSignal* yields, for a sequence of signal instances and a signal instance, the next signal instance of the sequence, or the value *undefined*, if the next signal instance is not determined.

nextSignal(*si*: *SIGNALINST*, *siSeq*:*SIGNALINST**): [*SIGNALINST*] $=_{\text{def}}$
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*] $=_{\text{def}}$
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 elements of *LINK* are SAM agents, such that their behaviour is defined through LINK-PROGRAM.

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

LINKSEQ $=_{\text{def}}$ *LINK**

Intuitively, elements of *LINK* are considered as point-to-point connection primitives for the transport of signals. More specifically, each *l* 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* → *SIGNAL*-**set**
controlled *from*: *LINK* → [*GATE*] // need to have optional result here, because function is also called within *allConnections* with general *AGENT*
controlled *to*: *LINK* → *GATE*
controlled *noDelay*: *LINK* → [**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 $=_{\text{def}}$ *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*.

But, how does a link agent *l* know whether it is applicable to a signal *si*? Now, this decision does of course depend on the values of *si*.*toArg*, *si*.*viaArg*, *si*.*signalType* and *l*.*with*. In other words, *l* is a legal choice for the transportation of *si* only, if 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 *applicable*, defined in clause F3.2.1.1.4. The domain *TOARG* is defined in clause F3.2.1.1.1.

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 *applicable* formally states all conditions that must be satisfied.

applicable(*s*: *SIGNAL*, *toArg*: [*TOARG*], *viaArg*: *VIAARG*, *g*: *GATE*, *l*: [*LINK*]): *BOOLEAN* $=_{\text{def}}$
    ∃ *commPath* ∈*allConnections* (*g*):
      (∀ *ll* ∈ *commPath*: *s* ∈ *ll*.*with* ∧ *ll*.*owner* ≠ *undefined*) ∧
      if *commPath* = *empty* **then**
        l = *undefined* ∧ ((*g*.*direction* = *outDir*) ⇒
            (*toArg* = *undefined* ∧ *s* ∈ *g*.*gateAS1*.**s-***Out-signal-identifier***-set**)) ∧
        ((*g*.*direction* = *inDir*) ⇒ (*validDestinationGate*(*g*, *toArg*) ∧ // to self
            s ∈ *g*.*gateAS1*.**s-***In-signal-identifier***-set**)) ∧ *viaArg* = ∅
      else
        if *l* ≠ *undefined* **then** *commPath*.*head* = *l* **else** *true* **endif** ∧
        ¬ ∃*ll* ∈ *LINK*: (*ll*.*from* = *commPath*.*last*.*to* ∧ *s* ∈ *ll*.*with*) ∧ // the path is complete
        viaArg ⊆ *commPath*.*commPathIds* ∧ *validDestinationGate*(*commPath*.*last*.*to*, *toArg*)
      endif

validDestinationGate(*g*: *GATE*, *toArg*: [*TOARG*]): *BOOLEAN* $=_{\text{def}}$
    if *toArg* ∈ *Agent-identifier* **then**
      g.*myAgent*.*agentAS1*.$identifier_1$ = *toArg* **else** *true* **endif** ∧
    if *toArg* ∈ *PID* ∧ *toArg* ≠ *nullPid* **then**
      ∃*sa* ∈ *AGENT*: (*sa*.*owner* = *g*.*myAgent* ∧ *sa*.*selfPid* = *toArg*) **else** *true*

endif

$allConnections(g\text{: }GATE)\text{: }LINKSEQ$**-set** $=_{\text{def}}$
$\bigcup$ ({ { < l > ⁀ list | list ∈ allConnections(l.to) } | l ∈ LINK: l.from = g }) ∪
{ *empty* }

commPathIds(*lSeq*: *LINK**): *Identifier*-**set** $=_{\text{def}}$
{ $g.gateAS1.identifier_1$ | g ∈ *GATE*: ∃l ∈ *lSeq*: (g = l.from ∨ g = l.to) } ∪
{ $l.agentAS1.identifier_1$ | l ∈*LINK*: (l ∈ *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 process instance.

TIMER $=_{\text{def}}$ *Identifier*
TIMERINST $=_{\text{def}}$ *PID* × *TIMER* × *VALUE**

The information associated with timers is accessed using the functions defined on *SIGNAL*.

Active timers

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

active(*tmi*:*TIMERINST*): *BOOLEAN* $=_{\text{def}}$ *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 static function (*duration*) is used to represent default duration values as defined by an SDL-2010 specification under consideration.

static *duration*: *TIMER* → *DURATION*

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*

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*

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

static *system*: $\rightarrow$ *SDLAGENTSET*
  initially *AGENT* = { *system* }

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* = $\emptyset$

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

STATENODEKIND $=_{\text{def}}$ { *stateNode*, *statePartition*, *procedureNode*}
STATENODEREFINEMENTKIND $=_{\text{def}}$ { *compositeStateGraph*, *stateAggregationNode*}
STATEENTRYPOINT $=_{\text{def}}$ [*State-entry-point-name*]
STATEEXITPOINT $=_{\text{def}}$ *State-exit-point-name* $\cup$ **DEFAULT**
STATENODEWITHENTRYPOINT $=_{\text{def}}$ *STATENODE* $\times$ (*STATEENTRYPOINT* $\cup$ **HISTORY**)
STATENODEWITHEXITPOINT $=_{\text{def}}$ *STATENODE* $\times$ *STATEEXITPOINT*
STATENODEWITHCONNECTOR $=_{\text{def}}$ *STATENODE* $\times$ *Connector-name*

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

controlled *stateNodeKind*: *STATENODE* $\rightarrow$ *STATENODEKIND*
controlled *stateNodeRefinement*: *STATENODE* $\rightarrow$ [*STATENODEREFINEMENTKIND*]
controlled *stateName*: *STATENODE* $\rightarrow$ *State-name*
controlled *stateId*: *STATENODE* $\rightarrow$ *STATEID*
controlled *inheritedStateNode*: *STATENODE* $\rightarrow$ [*STATENODE*]
controlled *parentStateNode*: *STATENODE* $\rightarrow$ [*STATENODE*]
controlled *stateTransitions*: *STATENODE* $\rightarrow$ *SEMTRANSITION*-**set**
controlled *startTransitions*: *STATENODE* $\rightarrow$ *STARTTRANSITION*-**set**
controlled *freeActions*: *STATENODE* $\rightarrow$ *FREEACTION*-**set**
controlled *statePartitionSet*: *STATENODE* $\rightarrow$ *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** $=_{\text{def}}$
    $\{sn\} \cup \bigcup (\{collectCurrentSubStates(x) \mid x \in sn.currentSubStates \cup sn.inheritedStateNodes\})$

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* $=_{\text{def}}$
    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* $=_{\text{def}}$
    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* $=_{\text{def}}$
    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* $=_{\text{def}}$
    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* $=_{\text{def}}$
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*] $=_{\text{def}}$
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** $=_{\text{def}}$
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, i.e., a state node owning the procedure graph.

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, and on state nodes (see clause F3.2.1.2.1), 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 *owns* its substructure, consisting of agent set instances. Similarly, a composite state node *owns* the state nodes or state partitions forming the refinement.

controlled *agentAS1*: *AGENT* → *Agent-definition*
controlled *channelAS1*: *AGENT* → [*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 *selfPid*: *SDLAGENT* → *PID*
controlled *sender*: *SDLAGENT* → *PID*
controlled *parent*: *SDLAGENT* → [*PID*]
controlled *offspring*: *SDLAGENT* → *PID*

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

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 (i.e. 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* → *Connector-name*
controlled *scopeContinueLabel*: *SDLAGENT* × *STATEID* → *CONTINUELABEL*
controlled *scopeStepLabel*: *SDLAGENT* × *STATEID* → *STEPLABEL*

The functions *scopeName*, *scopeContinueLabel* and *scopeStepLabel* are used for *Compound-node* interpretation (see 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

Entering, leaving and exiting of state nodes in general 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* $=_{\text{def}}$
  let *myDef*: *Agent-type-definition* = *g*.*myAgent*.*agentAS1*.**s-***Agent-type-identifier*.*idToNodeAS1* **in**
    ∀*cd* ∈ *myDef*.**s-***Channel-definition*-**set**: ∀*cp* ∈ *cd*.**s-***Channel-path*-**set**:
      (*g*.*gateAS1* ≠ *cp*.**s-***Originating-gate*.*idToNodeAS1* ∧
       g.*gateAS1* ≠ *cp*.**s-***Destination-gate*.*idToNodeAS1*)
  endlet

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

$ingates(a: \textit{AGENT}): \textit{GATE}\textbf{-set} =_{def}$
if *a.isAgentSet* **then**
$\{ g \in \textit{GATE}: g.myAgent = a \wedge g.direction = inDir \wedge g.gateUnconnected\}$
else
a.owner.ingates
endif

$outgates(a: \textit{AGENT}): \textit{GATE}\textbf{-set} =_{def}$
if *a.isAgentSet* **then**
$\{ g \in \textit{GATE}: g.myAgent = a \wedge g.direction = outDir \wedge g.gateUnconnected\}$
else
a.owner.outgates
endif

The derived function *ingates* and *outgates* collect all input gates and 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 is introduced, consisting of the signal type, the start label for any firing conditions, a priority value, and the start label of the transition actions. Additionally, state exit points may be given. Depending on the kind of transition, some of these components may be unspecified. For instance, in case of an input transition, there is no firing transition and no priority.

$\textit{SEMTRANSITION} =_{def} \textit{SIGNAL} \times [\textit{LABEL}] \times [\textit{NAT}] \times \textit{LABEL} \times [\textit{STATEEXITPOINT}]$

$\textit{STARTTRANSITION} =_{def} \textit{LABEL} \times \textit{STATEENTRYPOINT}$

$\textit{FREEACTION} =_{def} \textit{Connector-name} \times \textit{LABEL}$

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

$priorityInputTransitions(tSet: \textit{SEMTRANSITION}\textbf{-set}): \textit{SEMTRANSITION}\textbf{-set} =_{def}$
$\{ t \in tSet: t.\textbf{s-}\textit{SIGNAL} \neq \textbf{NONE} \wedge t.\textbf{s-}\textit{LABEL} = undefined \wedge t.\textbf{s-}\textit{NAT} \neq undefined \}$

$inputTransitions(tSet: \textit{SEMTRANSITION}\textbf{-set}): \textit{SEMTRANSITION}\textbf{-set} =_{def}$
$\{ t \in tSet: t.\textbf{s-}\textit{SIGNAL} \neq \textbf{NONE} \wedge t.\textbf{s-}\textit{NAT} = undefined \}$

$continuousSignalTransitions(tSet: \textit{SEMTRANSITION}\textbf{-set}): \textit{SEMTRANSITION}\textbf{-set} =_{def}$
$\{ t \in tSet: t.\textbf{s-}\textit{SIGNAL} = \textbf{NONE} \wedge t.\textbf{s-}\textit{LABEL} \neq undefined \wedge t.\textbf{s-}\textit{NAT} \neq undefined \}$

$spontaneousTransitions(tSet: \textit{SEMTRANSITION}\textbf{-set}): \textit{SEMTRANSITION}\textbf{-set} =_{def}$
$\{ t \in tSet: t.\textbf{s-}\textit{SIGNAL} = \textbf{NONE} \wedge t.\textbf{s-}\textit{NAT} = undefined \wedge t.\textbf{s-}\textit{STATEEXITPOINT} = undefined \}$

$exitTransitions(tSet: \textit{SEMTRANSITION}\textbf{-set}): \textit{SEMTRANSITION}\textbf{-set} =_{def}$
$\{ t \in tSet: t.\textbf{s-}\textit{STATEEXITPOINT} \neq undefined \}$

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.

NOTE – The data type part does not include the REF Aggregation-kind for reference variables defined in SDL-2010, and therefore is inconsistent with SDL-2010. Further work needs to be done to update the data part for reference variables defined in SDL-2010.

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 a process 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 parameter values and variable names

controlled domain *STATEID*

DECLARATION=$_{\text{def}}$ *Procedure-formal-parameter* ∪ *Variable-definition*

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

initProcedureState: *STATE* × *STATEID*× *STATEID* × *DECLARATION*-**set** × *DECLARATION** × *VALUE** × *Variable-identifier** → *STATE*

The domain *DECLARATION* is used to create lists of variables for a state. Positional parameters are guaranteed to come first in this list.

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

VALUE =$_{\text{def}}$ *SDLINTEGER* ∪ *SDLBOOLEAN* ∪ *SDLREAL* ∪ *SDLCHARACTER* ∪ *SDLSTRING*
∪ *PID* ∪ *SDLLITERALS* ∪ *SDLSTRUCTURE* ∪ *SDLARRAY* ∪ *SDLPOWERSET*
∪ *SDLBAG* ∪ *SDLTIME* ∪ *SDLDURATION*

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, if an exception occurs in the operation the termination is not defined, 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 has to provide functions that model how assignments are performed, namely

assign: *Variable-identifier* × *VALUE* × *STATE* × *STATEID* → *STATEOREXCEPTION*

The function *eval* (see below) retrieves the value associated with a variable for a given state and state id. The function *assign* associates a new value with a given variable. There is an ASSIGN rule macro using this function, which is doing the real assignment.

ASSIGN(*variableName*: *Variable-identifier*, *value*: *VALUE*, *state*: *STATE*, *id*: *STATEID*) ≡
Self.*stateAgent*.*state*:= *assign*(*variableName*, *value*, *state*, *id*)

Assignments are the only way to change the state.

In order to get the current value of a variable, the data part provides the function *eval* to get it. It returns *undefined* if the variable is not set.

eval: *Variable-identifier* × *STATE* × *STATEID* → *VALUE*

The semantics of these functions is given by the data semantics part.

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 static domain *PROCEDURE* for procedures (definitions) and a function *dispatch* for procedure instances.

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

For modelling the dynamic dispatch, a dispatch function is provided by the data part.

dispatch: *PROCEDURE* × *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.

functional: *PROCEDURE* × *VALUE** → *BOOLEAN*
compute: *PROCEDURE* × *VALUE** → *VALUEOREXCEPTION*

Moreover, the following domains and functions referring to the Predefined data are used.
derived domain *SDLBOOLEAN*
derived domain *SDLINTEGER*
derived *semvalueBool*: *SDLBOOLEAN* → *BOOLEAN*
derived *semvalueInt*: *SDLINTEGER*→ *NAT*
derived *semvalueRealNum*: *SDLREAL*→ *NAT*
derived *semvalueRealDen*: *SDLREAL*→ *NAT*
derived *semvalueReal*: *SDLREAL* → *REAL*

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 new 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 a process 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* $=_{\text{def}}$ *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 $=_{\text{def}}$ *LABEL*
BEHAVIOUR $=_{\text{def}}$ *PRIMITIVE*-**set**
PRIMITIVE $=_{\text{def}}$ *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 $=_{\text{def}}$ *VAR* ∪ *OPERATIONAPPLICATION* ∪ *CALL* ∪ *RETURN* ∪ *TASK* ∪ *ASSIGNPARAMETERS* ∪ *EQUALITY* ∪ *DECISION* ∪ *OUTPUT* ∪ *CREATE* ∪ *SET* ∪ *RESET* ∪ *TIMERACTIVE* ∪ *STOP* ∪ *SYSTEMVALUE* ∪ *ANYVALUE* ∪ *SETRANGECHECKVALUE* ∪ *SCOPE* ∪ *SKIP* ∪ *BREAK* ∪ *CONTINUE* ∪ *ENTERSTATENODE* ∪ *LEAVESTATENODE*

Domains

During the execution phase and the evaluation of actions we use labels basically 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 $=_{\text{def}}$ *LABEL*

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

Functions

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

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

values(*lSeq*: *VALUELABEL**, *sa*: *SDLAGENT*): *VALUE** $=_{\text{def}}$
    if *lSeq* = *empty* **then** *empty*
    else < *value*(*lSeq.head*,*sa*) > ⁀ *values*(*lSeq.tail*,*sa*)
    endif

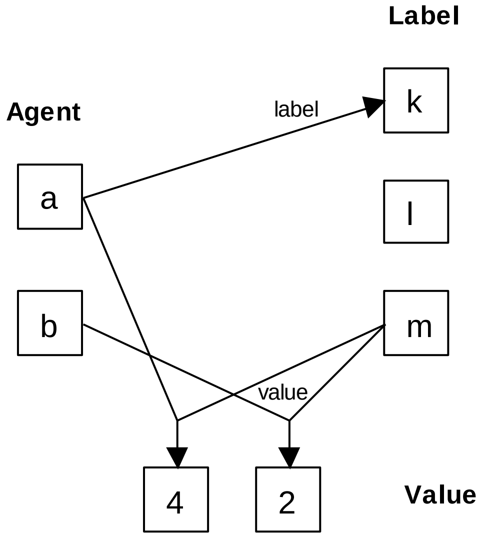

Figure F3-2 – Agents, labels and values

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* ∈ *VAR* **then** EVALVAR(*a*)
elseif *a* ∈ *OPERATIONAPPLICATION* **then** EVALOPERATIONAPPLICATION(*a*)
elseif *a* ∈ *CALL* **then** EVALCALL(*a*)
elseif *a* ∈ *RETURN* **then** EVALRETURN(*a*)
elseif *a* ∈ *TASK* **then** EVALTASK(*a*)
elseif *a* ∈ *ASSIGNPARAMETERS* **then** EVALASSIGNPARAMETERS(*a*)
elseif *a* ∈ *EQUALITY* **then** EVALEQUALITY(*a*)
elseif *a* ∈ *DECISION* **then** EVALDECISION(*a*)
elseif *a* ∈ *OUTPUT* **then** EVALOUTPUT(*a*)
elseif *a* ∈ *CREATE* **then** EVALCREATE(*a*)
elseif *a* ∈ *SET* **then** EVALSET(*a*)
elseif *a* ∈ *RESET* **then** EVALRESET(*a*)
elseif *a* ∈ *TIMERACTIVE* **then** EVALTIMERACTIVE(*a*)
elseif *a* ∈ *STOP* **then** EVALSTOP(*a*)
elseif *a* ∈ *SYSTEMVALUE* **then** EVALSYSTEMVALUE(*a*)
elseif *a* ∈ *ANYVALUE* **then** EVALANYVALUE(*a*)
elseif *a* ∈ *SETRANGECHECKVALUE* **then** EVALSETRANGECHECKVALUE(*a*)
elseif *a* ∈ *SCOPE* **then** EVALSCOPE(*a*)
elseif *a* ∈ *SKIP* **then** EVALSKIP(*a*)
elseif *a* ∈ *BREAK* **then** EVALBREAK(*a*)
elseif *a* ∈ *CONTINUE* **then** EVALCONTINUE(*a*)
elseif *a* ∈ *ENTERSTATENODE* **then** EVALENTERSTATENODE(*a*)
elseif *a* ∈ *LEAVESTATENODE* **then** EVALLEAVESTATENODE(*a*)
endif

F3.2.1.4.2 Primitive Var

Explanation

The Var primitive models the evaluation of a variable. It is used within the evaluation of expressions. An action of type *VAR* is a tuple consisting of a variable name and a so-called 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.

Representation

The domain *VAR* is defined as a Cartesian product of the domain *Variable-identifier* of variable names and domain *CONTINUELABEL* of labels.

VAR $=_{\mathrm{def}}$ *Variable-identifier* × *CONTINUELABEL*

Behaviour

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 function *eval* and is written at *Self*.*currentLabel*. In order to avoid conflicts with other agents, the function *value* takes a further argument of type *AGENT*, which identifies the owner of the value. Additionally, the label which determines the next rule to be fired is set to the given continue label.

```
EVALVAR(a:VAR) ≡
    if eval(a.s-Variable-identifier, Self.stateAgent.state, Self.currentStateId) = undefined then
        raise(UndefinedVariable)
    else
        value(Self.currentLabel, Self) := eval(a.s-Variable-identifier,
                    Self.stateAgent.state, Self.currentStateId)
        Self.currentLabel := a.s-CONTINUELABEL
    endif
```

Reference sections

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.

F3.2.1.4.3 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 procedure is executed by function *compute*: a non-functional operation, which is handled with function *dispatch* that determines (depending on the current values) the correct procedure identifier.

Representation

OPERATIONAPPLICATION $=_{\mathrm{def}}$ *PROCEDURE* × *VALUELABEL** × *CONTINUELABEL*

Behaviour

```
EVALOPERATIONAPPLICATION(a:OPERATIONAPPLICATION) ≡
    if functional(a.s-PROCEDURE, values(a.s-VALUELABEL-seq, Self)) then
        value(Self.currentLabel, Self):= compute(a.s-PROCEDURE, values(a.s-VALUELABEL-seq, Self))
        Self.currentLabel:= a.s-CONTINUELABEL
    else
```

let *pd*: *Procedure-definition* = *idToNodeAS1*(
dispatch(*a*.**s**-*PROCEDURE*, *values*(*a*.**s**-*VALUELABEL*-**seq**, *Self*))) **in**
CREATEPROCEDURE(*pd*, *Self*.*currentLabel*, *a*.**s**-*CONTINUELABEL*)
endlet
endif

Reference sections

For the definition of function *value* refer to clause F3.2.1.4.1. The definition of predicate *functional* and the definition of function *compute* can be found in clause F3.2.1.3.1.

F3.2.1.4.4 Primitive Call

Explanation

The call primitive models procedure calls, or method invocations. It is used within the evaluation of expressions and actions. 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, and a continue label. In-parameters are represented by value labels, in/out-parameters by variable identifiers. 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, and a continue label. In-parameters are represented by value labels, in/out-parameters by variable identifiers.

$CALLPARAM =_{def} VALUELABEL \cup Variable\text{-}identifier$

$CALL =_{def} Procedure\text{-}identifier \times CALLPARAM^{*} \times VALUELABEL \times CONTINUELABEL$

Behaviour

EVALCALL(*a*:*CALL*) ≡
let *pd*: *Procedure-definition* = *a*.**s**-*Procedure-identifier*.*idToNodeAS1* **in**
CREATEPROCEDURE(*pd*, *a*.**s**-*VALUELABEL*, *a*.**s**-*CONTINUELABEL*)
endlet

A procedure call is evaluated with macro CREATEPROCEDURE, which basically performs a procedure initialization and additionally creates a procedure state node.

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*

The parameter passing mechanism is realized by function *initProcedureState*. This function returns a state, which contains *Self*.*state* as a substate. Furthermore, for all local and in-parameters *initProcedureState* "creates" new locations. In-parameters are initialized with values stored in *resultLabel*. Formal inout-parameters are unified with the corresponding actual inout-parameters.

Reference sections

For the definition of macro CREATEPROCEDURE refer to clause F3.2.3.1.4. Information on procedure control blocks is given in clause F3.2.1.2.3.

F3.2.1.4.5 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 $=_{\text{def}}$ () × (*VALUELABEL* ∪ *STATEEXITPOINT*)

Behaviour

```
EVALRETURN(a: RETURN) ≡
        if a.s-implicit ∈ VALUELABEL then
            EVALEXITPROCEDURE(a.s-implicit )
        else
            EVALEXITCOMPOSITESTATE(a.s-implicit)
        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.

F3.2.1.4.6 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 name, a value label and a continue label. The variable name becomes as value within the state of the executing agent the value stored at value label.

Representation

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

TASK $=_{\text{def}}$ *Variable-identifier* × *VALUELABEL* × *BOOLEAN* × *CONTINUELABEL*

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 stored at value label.

EVALTASK(*a*:*TASK*) ≡
    ASSIGN(*a*.**s**-*Variable-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.

F3.2.1.4.7 Primitive AssignParameters

Explanation

The AssignParameters primitive is used for the assignments of parameters. An action of type *ASSIGNPARAMETERS* is defined as a tuple consisting of a variable identifier, a natural number, and a continue label.

Representation

An action of type *ASSIGNPARAMETERS* is defined as a tuple consisting of a variable identifier, a natural number, and a continue label.

ASSIGNPARAMETERS $=_{\text{def}}$ *Variable-identifier* × *NAT* × *CONTINUELABEL*

Behaviour

EVALASSIGNPARAMETERS(*a*:*ASSIGNPARAMETERS*) ≡
    let *v* = *Self*.*currentSignalInst*.*plainSignalValues*[*a*.**s**-*NAT*] **in**
        ASSIGN(*a*.**s**-*Variable-identifier*, *v*, *Self*.*stateAgent*.*state*, *Self*.*currentStateId*)
    endlet
    Self.*currentLabel* := *a*.**s**-*CONTINUELABEL*

Reference sections

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

F3.2.1.4.8 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 $=_{\text{def}}$ *VALUELABEL* × *VALUELABEL* × *CONTINUELABEL*

Behaviour

EVALEQUALITY (*a*:*EQUALITY*) ≡
    if *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.

F3.2.1.4.9 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 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.

$DECISION =_{\mathrm{def}} VALUELABEL \times ANSWER\text{-}\mathbf{set} \times [CONTINUELABEL]$

$ANSWER =_{\mathrm{def}} VALUELABEL \times CONTINUELABEL$

Behaviour

Macro EVALDECISION chooses an answer such that the decision-value given by the corresponding value label coincides with the answer-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*) = *value*(*an*.**s-***VALUELABEL*, *Self*)
Self.*currentLabel* := *an*.**s-***CONTINUELABEL*
endchoose
elseif *d*.**s-***CONTINUELABEL* ≠ *undefined* **then**
Self.*currentLabel* := *d*.**s-***CONTINUELABEL*
else *raise*(*NoMatchingAnswer*)
endif

Reference sections

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

F3.2.1.4.10 Primitive Output

Explanation

The Output primitive is used for expressing a signal output. An action of type *OUTPUT* consists of a signal, a sequence of value labels, an argument specifying the destination, an argument specifying a path, and a continue label.

Representation

An action of type *OUTPUT* consists of a signal type, a sequence of value labels, an argument specifying the destination, an argument specifying a path, and a continue label.

$OUTPUT =_{\mathrm{def}} SIGNAL \times VALUELABEL^{*} \times [VALUELABEL] \times VIAARG \times CONTINUELABEL$

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:OUTPUT) ≡
    SIGNALOUTPUT(a.s-SIGNAL, values(a.s-VALUELABEL-seq, Self),
        if a.s-VALUELABEL = undefined then undefined else value(a.s-VALUELABEL, Self) endif,
      a.s-VIAARG)
    Self.currentLabel := a.s-CONTINUELABEL
```

A signal output operation causes the creation of a new signal instance. The process 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 a process 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* (cf. clause F3.2.1.1.4). If the constraints cannot be met, the signal instance is discarded.

```
SIGNALOUTPUT(s:SIGNAL, vSeq:VALUE*, delay:DURATION, priority:NAT,
  toArg:[TOARG], viaArg:VIAARG) ≡
  let invReference =    (if toArg ∈ PID then
                        s.idToNodeAS1 ∉ toArg.s-Interface-definition.s-Signal-definition-set
                        else false endif)
  in
  if invReference then
    raise(InvalidReference)
  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.delay = delay
        si.priority = priority
        si.toArg := toArg
        si.viaArg := viaArg
        si.plainSignalSender := Self.selfPid
        INSERT(si, now, g)
      endextend
    endchoose
  endif
  endlet
```

Reference sections

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

F3.2.1.4.11 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 $=_{\text{def}}$ *Agent-identifier* × *VALUELABEL** × *CONTINUELABEL*

Behaviour

EVALCREATE(*a*:*CREATE*) ≡
  let *sas* = *take*({*sas* ∈ *SDLAGENTSET*: *sas*.*agentAS1* = *a*.**s-***Agent-identifier*.*idToNodeAS1* }) **in**
    if *sas*.*agentAS1*.**s-***Number-of-instances*.**s-***Maximum-number* ≠ *undefined* **then**
      let *n* = |{ *sa* ∈ *SDLAGENT*: *sa*.*owner* = *sas* }| **in**
        if *n* < *sas*.*agentAS1*.**s-***Number-of-instances*.**s-***Maximum-number* **then**
          CREATEAGENT(*sas*, *Self*, *sas*.*agentAS1*)
        else
          Self.*offspring* := *nullPid*
        endif
      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.

F3.2.1.4.12 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 $=_{\text{def}}$ *TIMELABEL* × *TIMER* × *VALUELABEL** × *CONTINUELABEL*

Domains

TIMELABEL $=_{\text{def}}$ *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.

F3.2.1.4.13 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.

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

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.

F3.2.1.4.14 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.

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

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.

F3.2.1.4.15 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.

F3.2.1.4.16 Primitive Stop

Explanation

The Stop primitive is used for initiating the stopping of an agent, which takes place in two phases. In the first phase, the state machine of the agent goes into a stopping state, meaning that it no longer selects and fires any transitions. The agent ceases to exist as soon as all contained agents have been removed.

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*) ≡
    Self.*agentMode2* := *stopping*

Reference sections

Clause F3.2.3.2.18.

F3.2.1.4.17 Primitive SystemValue

Explanation

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

Representation

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

Behaviour

EVALSYSTEMVALUE(*a*: *SYSTEMVALUE*) ≡
    value(*Self*.*currentLabel*, *Self*) :=
        case *a*.**s-***VALUEKIND* **of**
        | *kNow* => **mk-***SDLTIME*(*now*, *TimeType*)
        | *kSelf*=> *Self*.*selfPid*
        | *kParent*=> *Self*.*parent*
        | *kOffspring*=> *Self*.*offspring*
        | *kSender*=> *Self*.*sender*
        | *kActiveAgents*=> **mk-***SDLINTEGER*(|{ *sa* ∈ *SDLAGENT*: *sa*. *parent* = *Self* }|, *IntegerType*)
        endcase
    Self.*currentLabel* := *a*.**s-***CONTINUELABEL*

F3.2.1.4.18 Primitive AnyValue

Explanation

The AnyValue primitive computes the any expression.

Representation

$ANYVALUE =_{def} Sort\text{-}identifier \times CONTINUELABEL$

Behaviour

EVALANYVALUE(*a*: *ANYVALUE*) ≡
value(*Self*.*currentLabel*, *Self*) := *selectAnyValue*(*a*.**s-***Sort-identifier*)
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*: *Sort-identifier*): $VALUE =_{def}$
if *id*.*idToNodeAS1* ∈ *Interface-definition* **then** *nullPid*
else *take*({*v* | *v* ∈ *VALUE* ∧ *v*.*sort* =*id* })
endif

F3.2.1.4.19 Primitive SetRangeCheckLabel

Explanation

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

Representation

$SETRANGECHECKVALUE =_{def} VALUELABEL \times CONTINUELABEL$

static *rangeCheckValue*: → *LABEL*

The static function *rangeCheckValue* 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(*rangeCheckValue*, *Self*) := *value*(*a*.**s-***VALUELABEL*, *Self*)
Self.*currentLabel* := *a*.**s-***CONTINUELABEL*

F3.2.1.4.20 Primitive Scope

Explanation

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

Representation

$SCOPE =_{def} Connector\text{-}name \times Variable\text{-}definition\text{-}\mathbf{set} \times STARTLABEL \times STEPLABEL \times CONTINUELABEL$
$STEPLABEL =_{def} LABEL$

Behaviour

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

Reference sections

See also clause F3.2.3.1.8.

F3.2.1.4.21 Primitive Skip

Explanation

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

Representation

SKIP $=_{\text{def}}$ () × (*Connector-name* ∪ *CONTINUELABEL*)

Behaviour

EVALSKIP(*a*:*SKIP*) ≡
  if *a*.**s-implicit** ∈ *Connector-name* **then**
    Self.*stateNodeChecked* := *Self*.*currentParentStateNode*
    Self.*currentConnector* := **mk-***STATENODEWITHCONNECTOR*(*Self*.*currentParentStateNode*, *a*.**s-implicit**)
    Self.*agentMode2* := *selectingTransition*
    Self.*agentMode3* := *startSelection*
  else
    Self.*currentLabel* := *a*.**s-implicit**
  endif

Reference sections

Clause F3.2.3.2.8.

F3.2.1.4.22 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 $=_{\text{def}}$ () × (*Connector-name*)

Behaviour

EVALBREAK(*a*:*BREAK*) ≡
    if *scopeName*(*Self*, *Self*.*currentStateId*) = *a*.**s-***Connector-name* **then**
      Self.*currentLabel* := *scopeContinueLabel*(*Self*, *Self*.*currentStateId*)
    endif
    Self.*currentStateId* := *caller*(*Self*.*stateAgent*.*state*, *Self*.*currentStateId*)

F3.2.1.4.23 Primitive Continue

Explanation

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

Representation

CONTINUE $=_{\text{def}}$ () × (*Connector-name*)

Behaviour

EVALCONTINUE(*a*:*CONTINUE*) ≡
  if *scopeName*(*Self*, *Self*.*currentStateId*) = *a*.**s-***Connector-name* **then**
    Self.*currentLabel* := *scopeStepLabel*(*Self*, *Self*.*currentStateId*)
  else
    Self.*currentStateId* := *caller*(*Self*.*stateAgent*.*state*, *Self*.*currentStateId*)
  endif

F3.2.1.4.24 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 $=_{\text{def}}$ (*State-name* ∪ **HISTORY**) × *STATEENTRYPOINT* × *VALUELABEL**

Behaviour

```
EVALENTERSTATENODE(a:ENTERSTATENODE) ≡
    let enterName: (State-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
        Self.agentMode3 := enteringStateNode
        Self.agentMode4 := startPhase
        Self.currentLabel := undefined
        Self.continueLabel := undefined
    endlet
```

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.

F3.2.1.4.25 Primitive LeaveStateNode

Explanation

State nodes are left at the start of transitions.

Representation

LEAVESTATENODE $=_{\text{def}}$ *State-name* × *CONTINUELABEL*

Behaviour

```
EVALLEAVESTATENODE(a:LEAVESTATENODE) ≡
        choose sn: sn ∈ STATENODE ∧ sn.stateName = a.s-State-name ∧
            sn.stateNodeKind = stateNode ∧ sn.parentStateNode = Self.currentParentStateNode
            // assertion: sn = Self.previousStateNode
            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.

F3.2.1.5 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 following two functions 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.

```
getStateTransitions(s: State-node): SEMTRANSITION-set =def
    { mk-SEMTRANSITION(i.s-Signal-identifier,
        if i.s-Provided-expression = undefined then
            undefined
        else
            i.s-Provided-expression.startLabel
        endif,
        if i.s-PRIORITY = undefined then undefined else 1 endif,
        i.s-Transition.startLabel,
        undefined)
    | i ∈ s.s-Input-node-set } ∪
    { mk-SEMTRANSITION(NONE, sp.s-Provided-expression.startLabel,
        undefined, sp.s-Transition.startLabel, undefined)
    | sp ∈ s.s-Spontaneous-transition-set } ∪
    { mk-SEMTRANSITION(NONE, c.s-Continuous-expression.startLabel,
        c.s-Priority-name,     c.s-Transition.startLabel, undefined)
    | c ∈ s.s-Continuous-signal-set } ∪
    { mk-SEMTRANSITION(NONE, undefined, undefined, c.s-Transition.startLabel,
        if c.s-State-exit-point-name = undefined then DEFAULT else c.s-State-exit-point-name endif)
    | c ∈ s.s-Connect-node-set }

getStateStartTransitions(sn: State-start-node): STARTTRANSITION=def
    mk-STARTTRANSITION(sn.s-Transition.startLabel, sn.s-State-entry-point-name)

getNamedStartTransitions(sn: Named-start-node): STARTTRANSITION=def
    mk-STARTTRANSITION(sn.s-Transition.startLabel, sn.s-State-entry-point-name)

getProcStartTransitions(sn: Procedure-start-node): STARTTRANSITION=def
    mk-STARTTRANSITION(sn.s-Transition.startLabel, undefined)

getStartTransitions(s: (State-start-node ∪ Named-start-node ∪ Procedure-start-node)-set):
        STARTTRANSITION-set =def
```

{ **if** *sn* ∈ *State-start-node* then *getStateStartTransitions*(*sn*)
elseif *sn* ∈*Named-start-node* then *getNamedStartTransitions*(*sn*)
elseif *sn* ∈ *Procedure-start-node* then *getProcStartTransitions*(*sn*)
endif | *sn* ∈ *s* }

getFreeActions(*actions*: *Free-action*-**set**): *FREEACTION*-**set** $=_{\text{def}}$
{ **mk**-*FREEACTION*(*f*.**s**-*Connector-name*, *f*.**s**-*Transition*.*startLabel*) | *f* ∈ *actions* }

Here we present the function that compiles an 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, we first need a function to find unique labels for a syntactic entity. 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 $\forall\ d_1, d_2 \in$ *DEFINITIONAS1*: $\forall\ i_1, i_2 \in$ *NAT*:
$uniqueLabel(d_1, i_1) = uniqueLabel(d_2, i_2) \Leftrightarrow (d_1{=}d_2 \wedge i_1{=}i_2)$

Finally, to formalize the compilation, we also need an auxiliary function generating 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** $=_{\text{def}}$
if *s* = ∅ **then** *empty* **else**
let *el* = *c*.*take* **in**
< *el* > ⁀ *setToSeq*(*s* \ { *el* })
endlet
endif

The compilation is formalized in terms of the following two compilation functions, one for transition behaviour and one for expression behaviour.

compile: *DEFINITIONAS1* → *BEHAVIOUR*

compileExpr: *DEFINITIONAS1* × *LABEL* → *BEHAVIOUR*

The computed value of an expression *e* is always stored at *value*(*uniqueLabel*(*e*, 1), *Self*).

The two compilation functions are gradually introduced by defining a series of compilation patterns and the corresponding results; 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.

F3.2.2.1 States and triggers

The following parts are considered to form the definition of the function *compile* if put together with the following header. The contents of the case expression are all the compilation cases as given below.

compile(*a*: *DEFINITIONAS1*): *BEHAVIOUR* $=_{\text{def}}$
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*(*name*, *, *init*) =>
  if *init* ≠ *undefined* **then**
    compileExpr(*init*, *uniqueLabel*(*v*,1)) ∪
    {**mk-**PRIMITIVE(*uniqueLabel*(*v*,1), **mk-**TASK(*name*, *uniqueLabel*(*init*,1), *false*, *undefined*)) }
  else ∅
  endif

| *State-transition-graph*(*, *start*, *states*, *freeActions*) =>
    compile(*start*) ∪
  U{ *compile*(*s*) | *s* ∈ *states* } ∪
  U{ *compile*(*f*) | *f* ∈ *freeActions* }

| *Procedure-graph*(*start*, *states*, *freeActions*) =>
    compile(*start*) ∪
  U{ *compile*(*s*) | *s* ∈ *states* } ∪
  U{ *compile*(*f*) | *f* ∈ *freeActions* }

| *State-start-node*(*, *transition*) => *compile*(*transition*)

| *Procedure-start-node*(*transition*) => *compile*(*transition*)

| *Named-start-node*(*, *trans*) => *compile*(*trans*)

| *State-node*(*, *, *, *inputs*, *spontaneous*, *continuous*, *conns*, *) =>
    U{ *compile*(*i*) | *i* ∈ *inputs* } ∪
    U{ *compile*(*s*) | *s* ∈ *spontaneous* } ∪
  U{ *compile*(*c*) | *c* ∈ *continuous* } ∪
  U { *compile*(*c*) | *c* ∈ *conns* }

| *i* = *Input-node*(*, *, *vars*, *provided*, *transition*) =>
    if *provided* = *undefined* **then** ∅ **else** *compileExpr*(*provided*, *undefined*) **endif** ∪
    { **mk-**PRIMITIVE(*uniqueLabel*(*i*,*idx*),
      if *vars*[*idx*] ≠ *undefined* **then**
        mk-ASSIGNPARAMETERS(*vars*[*idx*], *idx*,
          uniqueLabel(*i*,*idx*))
      else mk-SKIP(*uniqueLabel*(*i*,*idx*))
      endif)
    | *idx* ∈ *toSet*(1..*vars*.*length* -1) } ∪
    { **mk-**PRIMITIVE(*uniqueLabel*(*i*, *vars*.*length*),
      if *vars*[*vars*.*length*] ≠ *undefined* **then**
        mk-ASSIGNPARAMETERS(*vars*[*vars*.*length*], *vars*.*length*, *transition*.*startLabel*)
      else mk-SKIP(*transition*.*startLabel*)
      endif)
    } ∪
    compile(*transition*)

| *Spontaneous-transition*(*provided*, *transition*) =>
    if *provided* = *undefined* **then** ∅ **else** *compileExpr*(*provided*, *undefined*) **endif** ∪
    compile(*transition*)

| *Continuous-signal*(*, *condition*, *, *transition*) =>
    compileExpr(*condition*, *undefined*) ∪
    compile(*transition*)

| *Connect-node*(*, *transition*) => *compile*(*transition*)

| *Free-action*(*, *transition*) => *compile*(*transition*)

```
| t=Transition(nodes, endnode) =>
    if t.parentAS1.parentAS1.s-State-name ≠ undefined then
        {mk-PRIMITIVE(uniqueLabel(a,1),
        mk-LEAVESTATENODE(t.parentAS1.parentAS1.s-State-name,
            startLabel(if nodes = empty then endnode else nodes.head endif))) }
    else ∅ endif ∪
    compileNodes ∪
    compile(endnode)
where

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

F3.2.2.2 Terminators

```
| Terminator(terminator) => compile(terminator)

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

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

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

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

| a=Action-return-node() =>
        {mk-PRIMITIVE(uniqueLabel(a,1), mk-RETURN
            (if parentAS1ofKind(a,Composite-state-type-definition).parentAS1 ∈
            Composite-state-type-definition then DEFAULT else undefined endif)) }

| v=Value-return-node(expr) =>
    compileExpr(expr, uniqueLabel(v,1)) ∪
        {mk-PRIMITIVE(uniqueLabel(v,1), mk-RETURN(uniqueLabel(expr,1))) }

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

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

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

| c= Continue-node(connector) =>
        {mk-PRIMITIVE(uniqueLabel(c,1), mk-CONTINUE(connector)) }
| d=Decision-node(question, answerset, elseanswer) =>
         (let aseq = answerset.setToSeq in
            compileExpr(question, aseq[1].startLabel) ∪
            { compileExpr(aseq[idx].s-implicit,
                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-implicit**, 1), *ans*.**s-***Transition*.*startLabel*)
| *ans* ∈ *answerset* },
if *elseanswer*=*undefined* **then** *undefined* **else** *elseanswer*.**s-***Transition* **endif**)) }
endlet) ∪
U{ *compile*(*ans*.**s-***Transition*) | *ans* ∈ *answerset* } ∪
compile(*elseanswer*.**s-***Transition*)

This concludes the definition of the *compile* function.

endcase // end of the compile function definition

F3.2.2.3 Actions

The following compilation parts define the function *compileExpr* with the following header.

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

All the contents of this function are given as patterns and what the result of the function for these patterns is. 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 process.

| *Graph-node*(*action*) => *compileExpr*(*action*, *next*)

| *a*=*Assignment*(*id*, *expr*) =>
compileExpr(*expr*, *uniqueLabel*(*a*,1)) ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*a*,1), **mk-***TASK*(*id*, *uniqueLabel*(*expr*,1), *false*, *next*))}

| *o*=*Output-node*(*sig*, *exprList*, *delay*, *priority*, *dest*, *via*) =>
if *dest* ∈ *Identifier* **then**
if *exprList* = *empty* **then** ∅
else *compileExpr*(*exprList*.*last*, *uniqueLabel*(*o*,1)) ∪
U{ *compileExpr*(*exprList*[*i*], *exprList*[*i*+1]. *startLabel*) | *i* ∈ 1..*exprList*.*length* - 1 }
endif ∪
compileExpr(*delay*, *uniqueLabel*(*o*,1)) ∪
compileExpr(*priority*, *uniqueLabel*(*o*,1)) ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*o*,1),
mk-*OUTPUT*(*sig*, <*uniqueLabel*(*e*,1) | *e* **in** *exprList* >,
uniqueLabel(*delay*,1), *uniqueLabel*(*priority*,1), *dest*, *via*, *next*)) }
else
if *exprList* = *empty* **then** ∅
else *compileExpr*(*exprList*.*last*, *dest*.*startLabel*) ∪
U{ *compileExpr*(*exprList*[*i*], *exprList*[*i*+1]. *startLabel*) | *i* ∈ 1..*exprList*.*length* - 1 }
endif ∪
compileExpr(*dest*, *uniqueLabel*(*o*,1)) ∪
compileExpr(*delay*, *uniqueLabel*(*o*,1)) ∪
compileExpr(*priority*, *uniqueLabel*(*o*,1)) ∪
{**mk-***PRIMITIVE*(*uniqueLabel*(*o*,1),
mk-*OUTPUT*(*sig*, <*uniqueLabel*(*e*,1) | *e* **in** *exprList* >,
uniqueLabel(*delay*,1), *uniqueLabel*(*priority*,1), *uniqueLabel*(*dest*,1), *via*, *next*)) }
endif

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

| *c*=*Call-node*(*, *procedureId*, *exprList*) =>
if *exprList* = *empty* **then** ∅
else *compileExpr*(*exprList*.*last*, *uniqueLabel*(*c*,1)) ∪
U{ *compileExpr*(*exprList*[*i*], *exprList*[*i*+1]. *startLabel*) | *i* ∈ 1..*exprList*.*length* - 1 }

```
        endif ∪
        (let paramDef = procedureId.idToNodeAS1.s-Procedure-formal-parameter-seq in
            {mk-PRIMITIVE(uniqueLabel(c,1),
                mk-CALL(procedureId,
                    <( if paramDef[idx] ∈ In-parameter
                     then uniqueLabel(exprList[idx], 1)
                     else exprList[idx]
                     endif )
                    | idx in (1..exprList.length ) >, uniqueLabel(c,1),
                    next)) }
        endlet)

    | c=Compound-node(name, variables, eh, initNodes, trans, stepNodes) =>
        {mk-PRIMITIVE(uniqueLabel(c,1),
            mk-SCOPE(name, variables,
                if initNodes = empty then trans.startLabel else initNodes.head.startLabel endif,
                if stepNodes = empty then trans.startLabel else stepNodes.head.startLabel endif,
                next)) } ∪
        compile(eh) ∪
        compileExpr(trans, undefined) ∪
        if stepNodes = empty then ∅
        else compileExpr( stepNodes.last, trans.startLabel) ∪
            ⋃{ compileExpr( stepNodes[i], stepNodes[i+1]. startLabel) | i ∈ 1..stepNodes.length - 1 }
        endif ∪
        if initNodes = empty then ∅
        else compileExpr( initNodes.last, trans.startLabel) ∪
            ⋃{ compileExpr( initNodes[i], initNodes[i+1]. startLabel) | i ∈ 1..initNodes.length - 1 }
        endif

    | s=Set-node(expr, timerId, exprList) =>
        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)) }

    | r=Reset-node(timerId, exprList) =>
        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) =>
        (let iseq = items.setToSeq in
            {mk-PRIMITIVE(uniqueLabel(r,1),
                mk-OPERATIONAPPLICATION(sdlTrue.idToNodeAS1, empty,
                    uniqueLabel(r, iseq.length+1))) } ∪
            { compileExpr(iseq[idx], uniqueLabel(r, idx)) | idx ∈ toSet(1..iseq.length) } ∪
            { mk-PRIMITIVE(uniqueLabel(r, idx),
                mk-OPERATIONAPPLICATION(sdlOr,
                    < uniqueLabel(r, idx+1), uniqueLabel(iseq[idx],1) >,
                    if idx=1 then next else iseq[idx-1].startLabel endif))
            | idx ∈ toSet(1..iseq.length) } ∪
            { mk-PRIMITIVE(uniqueLabel(r, 0), mk-BREAK(undefined)) }
        endlet)
```

The *Range-condition* above is computed as follows. First, a *true* value is evaluated. Then all items are sequentialized and evaluated from the last to the first; the results are cumulated using AND. Afterwards, the enclosing scope is left using a break.

| *o*=*Open-range*(*id*, *expr*) =>
compileExpr(*expr*, *uniqueLabel*(*o*, 1)) ∪
{ **mk-***Primitive*(*uniqueLabel*(*o*, 1),
mk-*OperationApplication*(*id*.*idToNodeAS1*,
< *rangeCheckValue*, *uniqueLabel*(*expr*, 1) >, *next*)) }

| *c*=*Closed-range*(*r1*, *r2*) =>
compileExpr(*r1*, *r2*.*startLabel*) ∪
compileExpr(*r2*, *uniqueLabel*(*c*, 1)) ∪
{ **mk-***Primitive*(*uniqueLabel*(*c*, 1),
mk-*OperationApplication*(*sdlAnd*, < *uniqueLabel*(*r1*, 1), *uniqueLabel*(*r2*, 1) >, *next*)) }

| *l*=*Literal*(*id*) =>
{**mk-***Primitive*(*uniqueLabel*(*l*,1),
mk-*OperationApplication*(*id*.*idToNodeAS1*, *empty*, *next*)) }

| *c*=*Conditional-expression*(*boolExpr*, *consExpr*, *altExpr*) =>
compileExpr(*boolExpr*, *uniqueLabel*(*c*, 2)) ∪
compileExpr(*consExpr*, *next*) ∪
compileExpr(*altExpr*, *next*) ∪
{**mk-***Primitive*(*uniqueLabel*(*c*,2),
mk-*OperationApplication*(*sdlTrue*.*idToNodeAS1*, *empty*, *uniqueLabel*(*c*, 1))) } ∪
{ **mk-***Primitive*(*uniqueLabel*(*c*, 1),
mk-*Decision*(*uniqueLabel*(*boolExpr*, 1),
{ **mk-***Answer*(*uniqueLabel*(*c*, 2), *consExpr*.*startLabel*) }, *altExpr*.*startLabel*)) }

| *e*=*Equality-expression*(*first*, *second*) =>
compileExpr(*first*, *second*.*startLabel*) ∪
compileExpr(*second*, *uniqueLabel*(*e*,1)) ∪
{**mk-***Primitive*(*uniqueLabel*(*e*,1),
mk-*Equality*(*uniqueLabel*(*first*,1), *uniqueLabel*(*second*,1), *next*)) }

| *o*=*Operation-application*(*id*, *exprList*) =>
if *exprList* = *empty* **then** ∅
else *compileExpr*(*exprList*.*last*, *uniqueLabel*(*o*,1)) ∪
⋃{ *compileExpr*(*exprList*[*i*], *exprList*[*i*+1]. *startLabel*) | *i* ∈ 1..*exprList*.*length* - 1 }
endif ∪
{**mk-***Primitive*(*uniqueLabel*(*o*,1),
mk-*OperationApplication*(*id*.*idToNodeAS1*,
< *uniqueLabel*(*e*, 1) | *e* **in** *exprList* >,
next)) }

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

| *v*=*Variable-access*(*id*) =>
{**mk-***Primitive*(*uniqueLabel*(*v*,1), **mk-***Var*(*id*, *next*)) }

| *n*=*Now-expression*() =>
{**mk-***Primitive*(*uniqueLabel*(*n*,1), **mk-***SystemValue*(*kNow*, *next*)) }

| *p*=*Parent-expression*() =>
{**mk-***Primitive*(*uniqueLabel*(*p*,1), **mk-***SystemValue*(*kParent*, *next*)) }

| *o*=*Offspring-expression*() =>
{**mk-***Primitive*(*uniqueLabel*(*o*,1), **mk-***SystemValue*(*kOffspring*, *next*)) }

| *s*=*Self-expression*() =>
{**mk-***Primitive*(*uniqueLabel*(*s*,1), **mk-***SystemValue*(*kSelf*, *next*)) }

| *s*=*Sender-expression*() =>
{**mk-***Primitive*(*uniqueLabel*(*s*,1), **mk-***SystemValue*(*kSender*, *next*)) }

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

| *a*=*Any-expression*(*id*) =>
{**mk-**_PRIMITIVE_(*uniqueLabel*(*a*,1), **mk-**_ANYVALUE_(*id*, *next*)) }

| *v*=*Value-returning-call-node*(*, *procedureId*, *exprList*) =>
if *exprList* = *empty* **then** ∅
else *compileExpr*(*exprList*.*last*, *uniqueLabel*(*v*,1)) ∪
⋃{ *compileExpr*(*exprList*[*i*], *exprList*[*i*+1]. *startLabel*) | *i* ∈ 1..*exprList*.*length* - 1 }
endif ∪
(**let** *paramDef* = *procedureId*.*idToNodeAS1*.**s-**_Procedure-formal-parameter_**-seq in**
{**mk-**_PRIMITIVE_(*uniqueLabel*(*v*,1),
mk-_CALL_(*procedureId*,
< (**if** *paramDef*[*idx*] ∈ *In-parameter*
then *uniqueLabel*(*exprList*[*idx*], 1)
else *exprList*[*idx*]
endif)
| *idx* **in** (1..*exprList*.*length*)>, *uniqueLabel*(*v*,1),
next)) }
endlet)

This concludes the definition of the expression compilation function.

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**
| *v*=*Variable-definition*(*, *, *init*) =>
if *init* = *undefined* **then** *undefined* **else** *init*.*startLabel* **endif**
| *s*=*State-start-node*(*,*, *trans*) => *startLabel*(*trans*)
| *p*=*Procedure-start-node*(*, *trans*) => *startLabel*(*trans*)
| *i*=*Input-node*(*, *, *, *, *, *trans*) => *startLabel*(*trans*)
| *s*=*Spontaneous-transition*(*, *, *trans*) => *startLabel*(*trans*)
| *c*=*Continuous-signal*(*, *, *, *trans*) => *startLabel*(*trans*)
| *c*=*Connect-node*(*, *, *trans*) => *startLabel*(*trans*)
| *f*=*Free-action*(*, *trans*) => *startLabel*(*trans*)
| *t*=*Transition*(*nodes*, *endnode*) =>
if *t*.*parentAS1*.*parentAS1* ∈ *State-node* **then** *uniqueLabel*(*t*,1) // insert the Leavestatenode
elseif *nodes* = *empty* **then** *startLabel*(*endnode*)
else *startLabel*(*nodes*.*head*)
endif
| *g*=*Graph-node*(*action*, *) => *startLabel*(*action*)
| *a*=*Assignment*(*, *expr*) => *startLabel*(*expr*)
| *o*= *Output-node*(*, *expr*, *dest*, *) =>
if *dest* ≠ *undefined* **then** *startLabel*(*dest*)
elseif *expr* = *empty* **then** *uniqueLabel*(*o*,1)
else *startLabel*(*expr*.*head*) **endif**
| *c*=*Create-request-node*(*, *exprList*) =>
if *exprList* = *empty* **then** *uniqueLabel*(*c*,1) **else** *exprList*.*head*.*startLabel* **endif**
| *c*=*Call-node*(*, *, *exprList*) =>
if *exprList* = *empty* **then** *uniqueLabel*(*c*,1) **else** *exprList*.*head*.*startLabel* **endif**
| *c*=*Compound-node*(*, *, *, *, *trans*, *) => *uniqueLabel*(*c*,1)
| *s*= *Set-node*(*when*, *, *) => *startLabel*(*when*)

| *r*=*Reset-node*(*, *exprList*) =>
    if *exprList* = *empty* **then** *uniqueLabel*(*r*,1) **else** *exprList*.*head*.*startLabel* **endif**
| *t*=*Terminator*(*terminator*, *) => *startLabel*(*terminator*)
| *n*=*Named-nextstate*(*, *undefined*) => *uniqueLabel*(*n*,1)
| *n*=*Named-nextstate*(*, *Nextstate-parameters*(*exprList*, *)) =>
    if *exprList* = *empty* **then** *uniqueLabel*(*n*,1) **else** *exprList*.*head*.*startLabel* **endif**
| *n*=*Dash-nextstate*(*) => *uniqueLabel*(*n*,1)
| *s*= *Stop-node*() => *uniqueLabel*(*s*,1)
| *a*=*Action-return-node*() => *uniqueLabel*(*a*,1)
| *v*=*Value-return-node*(*expr*) => *uniqueLabel*(*v*,1)
| *n*=*Named-return-node*(*expr*) => *uniqueLabel*(*n*,1)
| *j*= *Join-node*(*) => *uniqueLabel*(*j*,1)
| *b*= *Break-node*(*) => *uniqueLabel*(*b*,1)
| *c*= *Continue-node*(*) => *uniqueLabel*(*c*,1)
| *d*=*Decision-node*(*question*, *, *, *) => *startLabel*(*question*)
| a=*Decision-answer* (*r*, *) => *startLabel*(*r*)
| *n*=*Named-start-node*(*, *, *trans*) => *startLabel*(*trans*)
| *o*=*Open-range*(*, *expr*) => *startLabel*(*expr*)
| *c*=*Closed-range*(*, *) => *uniqueLabel*(*c*,1)
| *l*=*Literal*(*) => *uniqueLabel*(*l*,1)
| *c*=*Conditional-expression*(*, *, *) => *uniqueLabel*(*c*,1)
| *Equality-expression*(*first*, *) => *first*.*startLabel*
| *r*=*Range-check-expression*(*, *expr*) => *expr*.*startLabel*
| *v*=*Variable-access*(*id*) => *uniqueLabel*(*v*,1)
| *o*= *Operation-application*(*, *exprList*) =>
    if *exprList* = *empty* **then** *uniqueLabel*(*o*,1) **else** *exprList*.*head*.*startLabel* **endif**
| *v*=*Identifier*(*, *) => *uniqueLabel*(*v*,1)
| *n*=*Now-expression*() => *uniqueLabel*(*n*,1)
| *s*=*Self-expression*() => *uniqueLabel*(*s*,1)
| *p*=*Parent-expression*() => *uniqueLabel*(*p*,1)
| *o*=*Offspring-expression*() => *uniqueLabel*(*o*,1)
| *s*=*Sender-expression*() => *uniqueLabel*(*s*,1)
| *t*=*Timer-active-expression*(*, *exprList*) =>
    if *exprList* = *empty* **then** *uniqueLabel*(*t*,1) **else** *exprList*.*head*.*startLabel* **endif**
| *a*=*Any-expression*(*) => *uniqueLabel*(*a*,1)
| *v*=*Value-returning-call-node*(*, *, *exprList*) =>
    if *exprList* = *empty* **then** *uniqueLabel*(*v*,1) **else** *exprList*.*head*.*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.

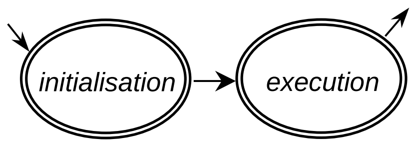

Figure F3-3 – Activity phases of SDL-2010 agents and agent sets (level 1)

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 S_0 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-Agent-type-identifier = undefined then
                atd.s-Agent-definition-set
            else let typedef: Agent-type-definition = atd.s-Agent-type-identifier.idToNodeAS1 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-Agent-type-identifier.idToNodeAS1 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-Agent-type-identifier.idToNodeAS1 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.

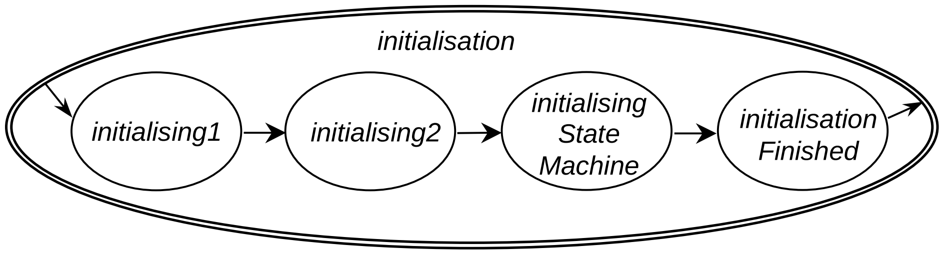

Figure F3-4 – Activity phases of SDL-2010 agents: initialization (level 2)

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*:*SDLAGENT*, *ad*:*Agent-definition*) ≡
    do forall *i*: *i* ∈ 1..*ad*.**s-***Number-of-instances*.**s-***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.*offspring* := *nullPid*
            sa.*selfPid* := **mk-***PID*(*sa*, *undefined*)
            if *pa* ≠ *undefined* **then**
                pa.*offspring* := **mk-***PID*(*sa*, *undefined*)
            endif
            let *ownerDef*: *Agent-type-definition* =

```
        ow.agentAS1.s-Agent-type-identifier. idToNodeAS1 in
    if ownerDef. s-Agent-kind ∈ {SYSTEM, BLOCK} then // containing agent set
        sa.stateAgent := sa
    elseif ownerDef.s-Agent-kind = PROCESS then // next level agent set
            sa.stateAgent := ow.owner.stateAgent
            else
        sa.stateAgent := sa
    endif
    endlet
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-Agent-type-identifier. idToNodeAS1 in
    if 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 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) ≡
```

REMOVEALLLINKS(*sa*)
sa.*program* := *undefined*
sa.*owner* := *undefined*

Removal of an agent is modelled by resetting the program (and the owner) to *undefined*, and by removing all owned link agents.

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.

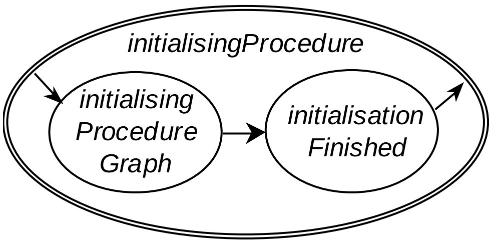

Figure F3-5 – Activity phases of SDL-2010 agents: firing of transitions (level 4)

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*, *vl*: [*VALUELABEL*], *cl*:[*CONTINUELABEL*]) ≡
    CREATEPROCEDUREGRAPH(*pd*, *vl*, *cl*)
    Self.*agentMode3* := *initialisingProcedure*
    Self.*agentMode4* := *initialisingProcedureGraph*

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-Agent-type-identifier = undefined then
                atd.s-Gate-definition-set
            else
                let typedef: Agent-type-definition = atd.s-Agent-type-identifier.idToNodeAS1 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.s-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.s-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-Agent-type-identifier = undefined then
                atd.s-Channel-definition-set
            else
                let typedef: Agent-type-definition = atd.s-Agent-type-identifier.idToNodeAS1 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.s-Originating-gate.idToNodeAS1 in
    let destDef: Gate-definition = cp.s-Destination-gate.idToNodeAS1 in
    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-Signal-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: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)
                si.viaArg := si.viaArg \
                    {Self.from.gateAS1.identifier_1,
                        Self.channelAS1.identifier_1}
            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 then
            sa.state := initAgentState(undefined, sid, undefined, atd.collectAllVariableDefinitions)
        else
            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-Agent-type-identifier = undefined then
                atd.s-Variable-definition-set
            else
                let typedef: Agent-type-definition = atd.s-Agent-type-identifier.idToNodeAS1 in
```

```
                atd.s-Variable-definition-set ∪
                typedef.collectAllVariableDefinitions
                endlet
            endif
    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 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)
    endextend

    where
        collectAllVariableDefinitions1(cstd: Composite-state-type-definition):
            Variable-definition-set =_def
            if cstd.s-Composite-state-type-identifier = undefined then
                cstd.s-Variable-definition-set
            else
                let typedef: Composite-state-type-definition =
                    cstd.s-Composite-state-type-identifier.idToNodeAS1 in
                cstd.s-Variable-definition-set ∪
                typedef .collectAllVariableDefinitions1
            endlet
            endif
    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) ≡
    extend StateId with sid
        sn.stateId := sid
        let outParams: Out-parameter* = < p in pd.collectAllProcedureFPars:
                                                    (p ∈ Out-parameter)> in
        sa.stateAgent.state := initProcedureState(sa.stateAgent.state, sid,
            sn.parentStateNode.stateId, pd.collectAllVariableDefinitions2,
            pd.collectAllProcedureFPars, empty,
            < p.s-Parameter.identifier_1 | p in outParams>)
        endlet
    endextend

    where
        collectAllVariableDefinitions2(pd: Procedure-definition): Variable-definition-set =_def
            if pd.s-Procedure-identifier = undefined then
                pd.s-Variable-definition-set
            else
                let procdef: Procedure-definition = pd.s-Procedure-identifier.idToNodeAS1 in
                pd.s-Variable-definition-set ∪
                procdef.collectAllVariableDefinitions2
                endlet
            endif

        collectAllProcedureFPars(pd:Procedure-definition): Procedure-formal-parameter* =_def
            if pd.s-Procedure-identifier = undefined then
                pd.s-Procedure-formal-parameter-seq
```

else
let *procdef*: *Procedure-definition* = *pd*.**s-***Procedure-identifier*.*idToNodeAS1* **in**
procdef.*collectAllProcedureFPars* ⁀
pd.**s-***Procedure-formal-parameter***-seq**
endlet
endif
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-***Connector-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*]) ≡
CREATETOPSTATEPARTITION(*smd*)

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, vl:[VALUELABEL], cl:CONTINUELABEL) ≡
    CREATEPROCEDURESTATENODE(pd, vl, 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*, but are not treated in this clause.

```
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-State-name
            sn.stateTransitions := snd.getStateTransitions
            sn.startTransitions := ∅        // updated if the state node is refined
            if snd.s-Composite-state-type-identifier ≠ undefined then
                Self.stateNodesToBeRefined := Self.stateNodesToBeRefined ∪ {sn}
                Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised ∪ {sn}
                let parent: Composite-state-type-definition =
                    snd.s-Composite-state-type-identifier.idToNodeAS1 in
                        sn.stateDefinitionAS1 := parent
                endlet
            endif
        endextend
    endchoose
```

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.

```
CREATEPROCEDURESTATENODE(pd:Procedure-definition, vl:[VALUELABEL], cl:CONTINUELABEL) ≡
    extend STATENODE with sn
        sn.procedureAS1 := pd
        sn.owner := Self
        sn.parentStateNode := Self.currentParentStateNode
        sn.stateNodeKind := procedureNode
        sn.stateName := mk-Name("")
        sn.stateTransitions := ∅
        sn.startTransitions := ∅          // updated if the state node is refined
        sn.resultLabel := vl
        Self.stateNodesToBeRefined := {sn}
        Self.stateNodesToBeCreated := ∅
        Self.statePartitionsToBeCreated := ∅
        Self.stateNodesToBeSpecialised := {sn}
        Self.currentProcedureStateNode := sn
        Self.callingProcedureNode := sn
        CREATEPROCEDUREVARIABLES(Self,sn,pd)
        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.

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
            let parent: Composite-state-type-definition =
                smd.s-Composite-state-type-identifier.idToNodeAS1 in
                sn.stateDefinitionAS1 := parent
            endlet
            sn.stateName := smd.s-State-name
            Self.stateNodesToBeRefined := {sn}
            Self.stateNodesToBeSpecialised := {sn}
        else
            sn.stateName := mk-Name("^pdummy^p")
            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
            do forall cd: cd ∈ spd.s-Connection-definition-set
                if cd ∈ Entry-connection-definition then
                    entryConnection(cd.s-Outer-entry-point.adaptEntryPoint, sn) :=
                        adaptEntryPoint(cd.s-Inner-entry-point)
                elseif cd ∈ Exit-connection-definition then
                    exitConnection(cd.s-Inner-exit-point, sn) := cd.s-Outer-exit-point
                endif
            enddo
            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: Composite-state-type-definition =
            sns.stateDefinitionAS1 in
            if cstd.s-Composite-state-type-identifier ≠ undefined then
                let parent: State-node = cstd.s-Composite-state-type-identifier.idToNodeAS1 in
                extend STATENODE with sn
                    sn.stateAS1 := 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
endlet
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*)
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*) ≡
let *sr* = *cstd*.**s-implicit in**
if *sr* ∈ *Composite-state-graph* **then**
CREATECOMPOSITESTATEGRAPH(*sn*,*sr*)
elseif *sr* ∈ *State-aggregation-node* **then**
CREATESTATEAGGREGATIONNODE(*sn*,*sr*)
endif
endlet

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*.

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**
DELIVERSIGNALS
endif
endlet

The behaviour of agent sets is formalized below.

DELIVERSIGNALS ≡
choose *g*: *g* ∈ *Self*.*ingates* ∧ *g*.*queue* ≠ *empty*
let *si* = *g*.*queue*.*head* **in**
DELETE(*si*,*g*)
if *si*.*toArg* ∈ *PID* ∧ *si*.*toArg* ≠ *undefined* **then**
choose *sa*: *sa* ∈ *SDLAGENT* ∧ *sa*.*owner* = *Self* ∧ *sa*.*selfPid* = *si*.*toArg*
INSERT(*si*, *si*.*arrival*, *sa*.*inport*)
endchoose
else

```
            choose sa: sa ∈ SDLAGENT ∧ sa.owner = Self
                INSERT(si, si.arrival, sa.inport)
            endchoose
        endif
    endlet
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.

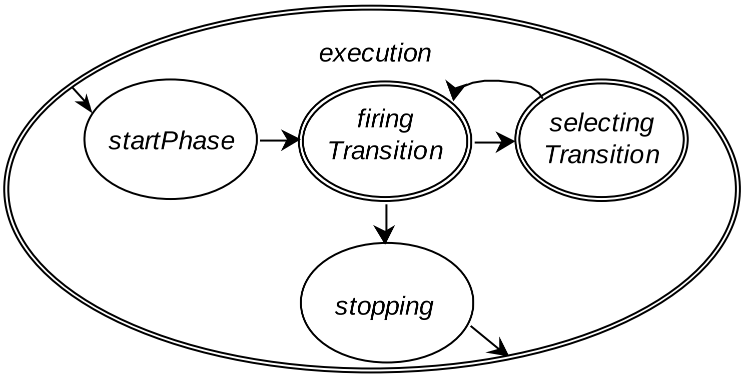

Figure F3-6 – Activity phases of SDL-2010 agents: execution (level 2)

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-Agent-type-identifier.idToNodeAS1 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.

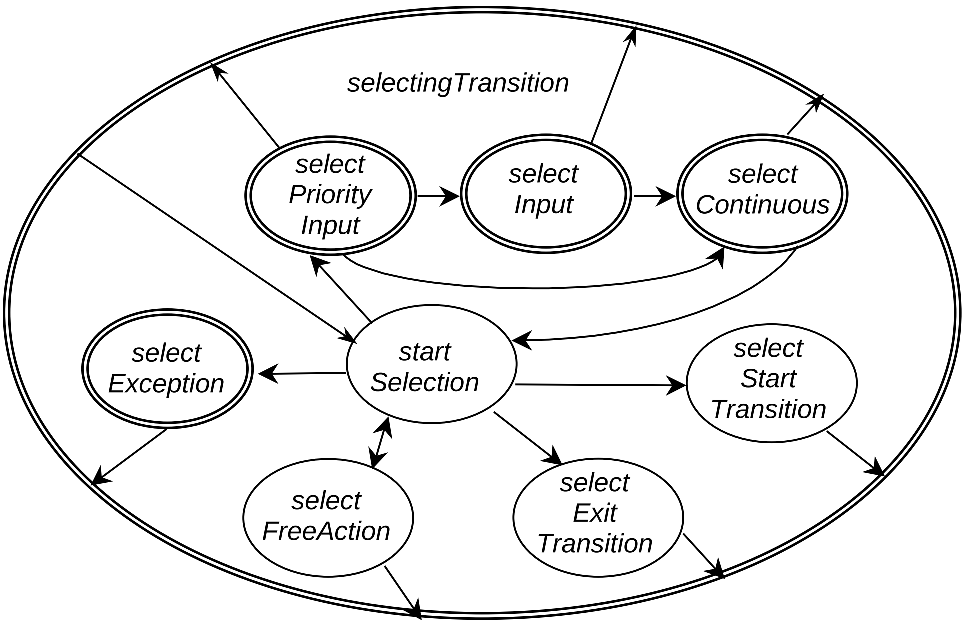

Figure F3-7 – Activity phases of SDL-2010 agents: selecting transition (level 3)

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 process is repeated/aborted. The evaluation of provided expressions and continuous expressions may alter the local state of the process, 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*.**s-***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 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-Connector-name = Self.currentConnector.s-Connector-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.

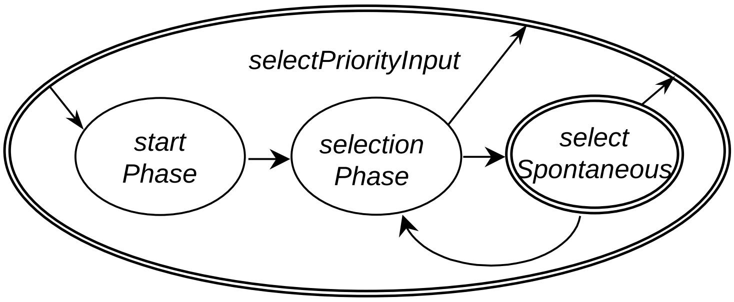

Figure F3-8 – Activity phases of SDL-2010 agents: selecting priority inputs (level 4)

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 Self.signalChecked.signalType ∈
                        sn.stateAS1.s-Save-signalset.s-Signal-identifier-set then
                            Self.SignalSaved := true
                    endif
                endlet
            endif
        endlet

    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.

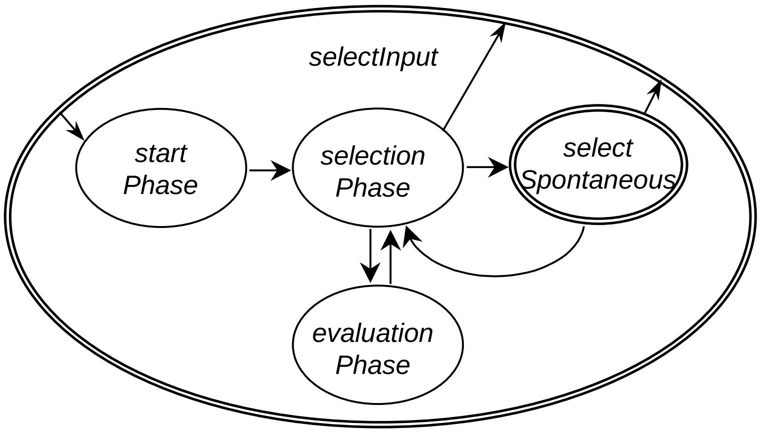

Figure F3-9 – Activity phases of SDL-2010 agents: selecting inputs (level 4)

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

This ASM macro 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, 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.s-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.s-LABEL
            Self.agentMode4 := evaluationPhase

        NEXTSTATENODETOBECHECKED1 ≡
            if Self.stateNodesToBeChecked ≠ ∅ ∧ ¬ Self.SignalSaved then
                SELECTNEXTSTATENODE1
            else
                if ¬ Self.SignalSaved then                // implicit transition
                    DELETE(Self.signalChecked,Self.inport)
                endif
                NEXTSIGNALTOBECHECKED1
                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)
                    // 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 := {t∈ sn.stateTransitions.inputTransitions:
                    t.s-SIGNAL = Self.signalChecked.signalType}
                if Self.signalChecked.signalType ∈
                    sn.stateAS1.s-Save-signalset.s-Signal-identifier-set then
                    Self.SignalSaved := true
                endif
            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
                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 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 ∧ b.s-LABEL = Self.currentLabel
            EVAL(b.s-ACTION)
        endchoose
    elseif semvalueBool(value(Self.transitionChecked.s-LABEL,Self)) then
            Self.currentSignalInst := Self.signalChecked
            Self.sender := Self.signalChecked.signalSender
            DELETE(Self.signalChecked,Self.inport)
            TRANSITIONFOUND(Self.transitionChecked)
    else
        Self.agentMode4 := selectionPhase
    endif
```

If an input 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 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.

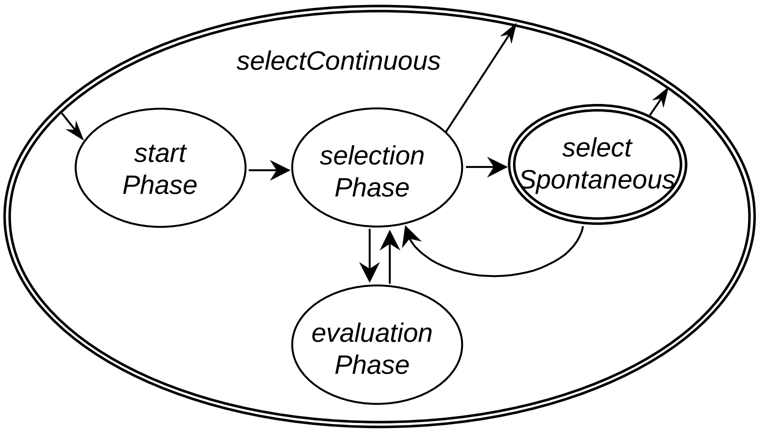

Figure F3-10 – Activity phases of SDL-2010 agents: selecting continuous signals (level 4)

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*.**s**-*LABEL* ≠ *undefined* **then**
          EVALUATEENABLINGCONDITION1(*t*)
        else
          TRANSITIONFOUND(*t*)
        endif
      else
        NEXTSTATENODETOBECHECKED2

```
        endif
    endlet
endif

where

    EVALUATEENABLINGCONDITION1(t:SEMTRANSITION) ≡
        Self.transitionChecked := t
        Self.currentStateId := Self.stateNodeChecked.parentStateNode.stateId
        Self.currentLabel := t.s-LABEL
        Self.agentMode4 := evaluationPhase

    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 checking, at any time during the selection process, a single spontaneous transition.

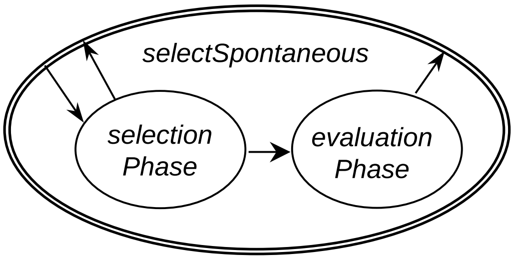

Figure F3-11 – Activity phases of SDL-2010 agents: selecting spontaneous transitions (level 5)

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**
        EVALUATEENABLINGCONDITION2(*t*)
      else
        Self.*sender* := *Self*.*selfPid*
        TRANSITIONFOUND(*t*)
      endif
    endchoose
  else
    Self.*agentMode4* := *selectionPhase*

endif

where

EVALUATEENABLINGCONDITION2(*t*:*SEMTRANSITION*) ≡
Self.*transitionChecked* := *t*
Self.*currentStateId* := *Self*.*stateNodeChecked*.*parentStateNode*.*stateId*
Self.*currentLabel* := *t*.**s-***LABEL*
Self.*agentMode5* := *evaluationPhase*
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.

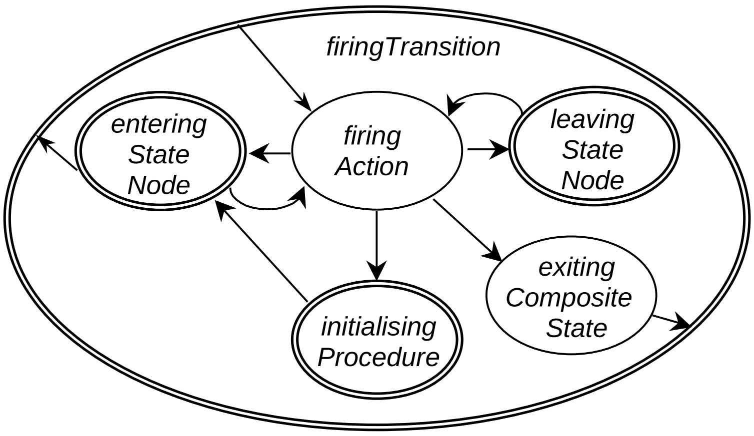

Figure F3-12 – Activity phases of SDL-2010 agents: firing transitions (level 3)

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.

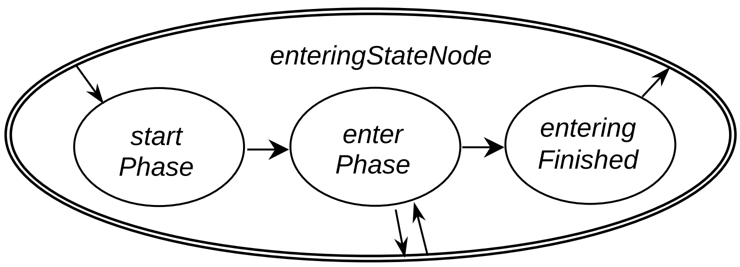

Figure F3-13 – Activity phases of SDL-2010 agents: entering state node (level 4)

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 *cstd*: *Composite-state-type-definition* =
snwen.**s-***STATENODE*. *stateDefinitionAS1* **in**
let *aggr*: *State-aggregation-node* = *cstd*.s-**implicit in**
if *aggr*.**s-***Entry-procedure-definition* ≠ *undefined* **then**
CREATEPROCEDURE(*aggr*.**s-***Entry-procedure-definition*, *undefined*,
undefined)

endif
endlet

REFINEMENTCOMPSTATENODE(*snwen*:*STATENODEWITHENTRYPOINT*) ≡
Self.*stateNodesToBeEntered* := *Self*.*stateNodesToBeEntered* \ {*snwen*}
let *cstd*: *Composite-state-type-definition* = *snwen*.**s-***STATENODE*.*stateDefinitionAS1* **in**
let *comp*: *Composite-state-graph* = *cstd*.s-**implicit in**
if *comp*.**s-***Entry-procedure-definition* ≠ *undefined* **then**
CREATEPROCEDURE(*comp*.**s-***Entry-procedure-definition*, *undefined*,
undefined)
endif
endlet
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.

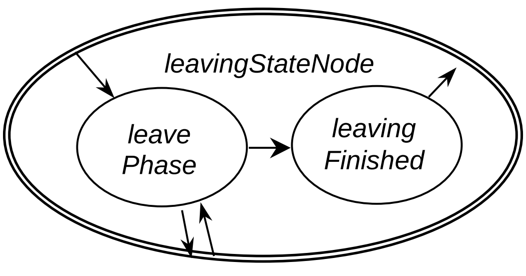

Figure F3-14 – Activity phases of SDL-2010 agents: leaving state node (level 4)

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 *cstd* : *Composite-state-type-definition* =
sn.*stateAS1*.**s-***Composite-state-type-identifier*.*idToNodeAS1* **in**
let *comp* : *Composite-state-graph* = *cstd*.**s-implicit** **in**
if *comp*.**s-***Exit-procedure-definition* ≠ *undefined* **then**
CREATEPROCEDURE(*comp*.**s-***Exit-procedure-definition*,*undefined*,
undefined)
endif
endlet
elseif *sn*.*stateNodeRefinement* = *stateAggregationNode* **then**
let *cstd*: *Composite-state-type-definition* =
sn.*stateAS1*.**s-***Composite-state-type-identifier*.*idToNodeAS1* **in**
let *aggr*: *State-aggregation-node* = *cstd*.**s-implicit** **in**
if *aggr*.**s-***Exit-procedure-definition* ≠ *undefined* **then**
CREATEPROCEDURE(*aggr*.**s-***Exit-procedure-definition*, *undefined*,
undefined)
endif
endlet
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 removed.

```
STOPPHASE ≡
    if ∀sas ∈ SDLAGENTSET: (sas.owner = Self ⇒ ¬ ∃sa ∈ SDLAGENT: sa.owner = sas) then
        REMOVEALLAGENTSETS(Self)
        REMOVEAGENT(Self)
    endif
```

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

An operator is functional if it is predefined. The built-in procedures for structures and literals are treated as predefined.

functional(*procedure*: *PROCEDURE*, *values*: *VALUE**): *BOOLEAN* =$_{\text{def}}$
(*procedure*.*identifier*$_1$.**s-***Qualifier*.*head* ∈ *Package-qualifier* ∧
procedure.*identifier*$_1$.**s-***Qualifier*.*head*.**s-***Package-name*.**s-***TOKEN* = "**Predefined**")
∨ *isSpecialStructOp*(*procedure*)
∨ *isSpecialLiteralOp*(*procedure*)

intype(*procedure*: *PROCEDURE*, *name*: *Name*): *BOOLEAN* =$_{\text{def}}$
procedure.*identifier*$_1$.**s-***Qualifier*.*last*.**s-***Data-type-name* = *name*

compute (*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUEOREXCEPTION* =$_{\text{def}}$
if *intype* (*procedure*, *IntegerType*.**s-***Name*) **then** *computeInteger*(*procedure*, *values*)
elseif *intype* (*procedure*, *BooleanType*.**s-***Name*) **then** *computeBoolean*(*procedure*, *values*)
elseif *intype* (*procedure*, *CharacterType*.**s-***Name*) **then** *computeChar*(*procedure*, *values*)
elseif *intype* (*procedure*, *RealType*.**s-***Name*) **then** *computeReal*(*procedure*, *values*)
elseif *intype* (*procedure*, *DurationType*.**s-***Name*) **then** *computeDuration*(*procedure*, *values*)
elseif *intype* (*procedure*, *TimeType*.**s-***Name*) **then** *computeTime*(*procedure*, *values*)
elseif *intype* (*procedure*, *StringType*.**s-***Name*) **then** *computeString*(*procedure*, *values*)
elseif *intype* (*procedure*, *ArrayType*.**s-***Name*) **then** *computeArray*(*procedure*, *values*)
elseif *intype* (*procedure*, *PowersetType*.**s-***Name*) **then** *computePowerset*(*procedure*, *values*)
elseif *intype* (*procedure*, *BagType*.**s-***Name*) **then** *computeBag*(*procedure*, *values*)
elseif *isSpecialStructOp*(*procedure*) **then** *computeStruct*(*procedure*, *values*)
elseif *isSpecialLiteralOp* (*procedure*) **then** *computeLiteral*(*procedure*, *values*)
else
raise(*OutOfRange*)
endif

The *TOKEN* domain consists of character strings. The function *emptyToken* is therefore an empty character string.

emptyToken: *TOKEN* =$_{\text{def}}$
""

The function *definingSort* computes the scope in which an operator was defined.

definingSort(*p*: *PROCEDURE*): *Identifier* =$_{\text{def}}$
p.*parentAS1*.*identifier*$_1$

The function *procName* computes the token of an operator.

procName(*p*: *PROCEDURE*): *TOKEN* =$_{\text{def}}$
p.**s-***Operation-name*.**s-***TOKEN*

F3.3.1.1 Well-known definitions

A set of functions refers to well-known *Data-type-definition* nodes from the package Predefined.

BooleanType: *Identifier* =$_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Boolean"))
CharacterType: *Identifier* =$_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Character"))
StringType: *Identifier* =$_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("String"))
IntegerType: *Identifier* =$_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Integer"))
RealType: *Identifier* =$_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Real"))

ArrayType: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Array"))
PowersetType: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Powerset"))
DurationType: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Duration"))
TimeType: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Time"))
BagType: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Bag"))
PidType: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Pid"))

Furthermore, there are a number of well-known identifiers for exceptions.

OutOfRange: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("OutOfRange"))
InvalidReference: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>,**mk-***Name*("InvalidReference"))
NoMatchingAnswer: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>,**mk-***Name*("NoMatchingAnswer "))
UndefinedVariable: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>,**mk-***Name*("UndefinedVariable"))
UndefinedField: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("UndefinedField"))
InvalidIndex: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("InvalidIndex"))
DivisionByZero: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("DivisionByZero"))
EmptyException: *Identifier* $=_{\text{def}}$
mk-*Identifier*(<**mk-***Package-qualifier*(**mk-***Name*("Predefined"))>, **mk-***Name*("Empty"))

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.

The further study of handling the *Aggregation-kind* **REF** requires exceptions "InvalidCall" and "InvalidSort" to be added to the above list (see clause 12.2.7 of [ITU-T Z.107] and clause 12.2.8.1 of [ITU-T Z.107]).

raise(*ex*:*Identifier*): *Identifier* $=_{\text{def}}$
UNDEFINEDBEHAVIOUR

There are also the following well-known operation signatures:

sdlAnd: *Static-operation-signature* $=_{\text{def}}$
mk-*Operation-signature*(**mk-***Name*("and"),
< (*BooleanType*), (*BooleanType*)>)

sdlOr: *Static-operation-signature* $=_{\text{def}}$
mk-*Operation-signature*(**mk-***Name*("or"),
< (*BooleanType*), (*BooleanType*)>)

sdlTrue: *Literal-signature* $=_{\text{def}}$
mk-*Literal-signature* (**mk-***Name*("true"), **mk-***Result*(*BooleanType*), *undefined*)

F3.3.1.2 Boolean

The function *computeBoolean* determines the value of an application of a Predefined Boolean operator.

SDLBOOLEAN $=_{def}$ *BOOLEAN* × *Identifier*

computeBoolean(*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUEOREXCEPTION* $=_{def}$
let *restype* = *definingSort*(*procedure*) **in**
case *procedure*.*procName* **of**
| "not" => **mk-***SDLBOOLEAN*(¬ *values*.*head*.*semvalueBool*, *restype*)
| "and" => **mk-***SDLBOOLEAN*(*values*.*head*.*semvalueBool* ∧ *values*.*tail*.*head*.*semvalueBool*, *restype*)
| "or" => **mk-***SDLBOOLEAN*(*values*.*head*.*semvalueBool* ∨ *values*.*tail*.*head*.*semvalueBool*, *restype*)
| "xor" => **mk-***SDLBOOLEAN*(¬ (*values*. *head*.*semvalueBool* ⇔ *values*.*tail*.*head*.*semvalueBool*), *restype*)
| "=>" => **mk-***SDLBOOLEAN*(*values*.*head*.*semvalueBool* ⇒ *values*.*tail*.*head*.*semvalueBool*, *restype*)
endcase
endlet

semvalueBool(*v*:*SDLBOOLEAN*): *BOOLEAN* $=_{def}$ *v*.**s-***BOOLEAN*

F3.3.1.3 Integer

SDLINTEGER $=_{def}$ *NAT* × *Identifier*

semvalueInt(*v*:*SDLINTEGER*): *NAT*$=_{def}$ *v*.**s-***NAT*

computeInteger(*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUEOREXCEPTION* $=_{def}$
let *restype* = *definingSort*(*procedure*) **in**
if *procedure* ∈ *Literal-signature* **then**
integerLiteral(0,*procedure*.*procName*, *restype*)
elseif *procedure*.*procName* = "-" ∧ *values*.*length* = 1 **then**
mk-*SDLINTEGER*(0 - *values*.*head*.*semvalueInt*, *restype*)
elseif *procedure*.*procName* ∈ {"+", "-", "*", "/", "mod", "rem", "<", ">", "<=", ">=", "power"}
then
let *val1* = *values*[1]. *semvalueInt*, *val2* = *values*[2]. *semvalueInt* **in**
case *procedure*.*procName* **of**
| "+" => **mk-***SDLINTEGER* (*val1*+*val2*, *restype*)
| "-"=> **mk-***SDLINTEGER* (*val1* – *val2*, *restype*)
| "*"=> **mk-***SDLINTEGER* (*val1* * *val2*, *restype*)
| "/"=>
if *val2* = 0 **then**
raise(*DivisionByZero*)
else
mk-*SDLINTEGER* (*intDiv*(*val1*,*val2*), *restype*)
endif
| "mod"=>
if *val2* = 0 **then**
raise(*DivisionByZero*)
else
mk-*SDLINTEGER* (*intMod*(*val1*,*val2*), *restype*)
endif
| "rem"=>
if *val2* = 0 **then**
raise(*DivisionByZero*)
else
mk-*SDLINTEGER* (*intRem*(*val1*,*val2*), *restype*)
endif
| "power"=> **mk-***SDLINTEGER* (*intPower*(*val1*,*val2*), *restype*)
| "<" => **mk-***SDLBOOLEAN*(*val1* < *val2*, *BooleanType*)
| "<=" => **mk-***SDLBOOLEAN*(*val1* ≤ *val2*, *BooleanType*)
| ">" => **mk-***SDLBOOLEAN*(*val1* > *val2*, *BooleanType*)
| ">="=> **mk-***SDLBOOLEAN*(*val1* ≥ *val2*, *BooleanType*)
endcase
endlet
else *raise*(*OutOfRange*)

endif
endlet

The function *numberValue* determines the *NAT* associated with a single character in the range "0" to "9".

numberValue(*c*:*TOKEN*): *NAT* $=_{\text{def}}$
case *c* **of**
| "0" => 0
| "1" => 1
| "2" => 2
| "3" => 3
| "4" => 4
| "5" => 5
| "6" => 6
| "7" => 7
| "8" => 8
| "9" => 9
endcase

The function *integerLiteral* returns the *SDLINTEGER* value for a real literal.

integerLiteral(*num*: *NAT*, *proc*: *TOKEN*, *type*: *Identifier*): *SDLINTEGER* $=_{\text{def}}$
if *proc* = *emptyToken* **then**
mk-*SDLINTEGER* (*num*, *type*)
else
integerLiteral(*num**10 + *numberValue*(*proc.head*), *proc.tail*, *type*)
endif

The function *intDiv* returns the result of integer-dividing its arguments.

intDiv(*a*: *NAT*, *b*: *NAT*):*NAT* $=_{\text{def}}$
if $a \geq 0 \wedge b > a$ **then** 0
elseif $a \geq 0 \wedge b \leq a \wedge b > 0$ **then** 1 + *intDiv*(*a* - *b*, *b*)
elseif $a \geq 0 \wedge b < 0$ **then** - *intDiv*(*a*, -*b*)
elseif $a < 0 \wedge b < 0$ **then** *intDiv* (-*a*, -*b*)
elseif $a < 0 \wedge b > 0$ **then** - *intDiv* (-*a*, *b*)
else *raise*(*DivisionByZero*)
endif

The function *intMod* returns the result of the integer-modulo operation.

intMod(*a*: *NAT*, *b*: *NAT*):*NAT* $=_{\text{def}}$
if $a \geq 0 \wedge b > 0$ **then** *intRem*(*a*,*b*)
elseif $b < 0$ **then** *intMod*(*a*, -*b*)
elseif $a < 0 \wedge b > 0 \wedge$ *intRem*(*a*,*b*) = 0 **then** *intRem*(*a*,*b*)
elseif $a < 0 \wedge b > 0 \wedge$ *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*: *NAT*, *b*: *NAT*):*NAT* $=_{\text{def}}$
a - *b* * *intDiv*(*a*,*b*)

The function *intPower* returns the result of the integer-power operation.

intPower(*a*: *NAT*, *b*: *NAT*):*NAT* $=_{\text{def}}$
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 $=_{\text{def}}$ *Name* × *Identifier*

computeChar(*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUEOREXCEPTION* $=_{\text{def}}$
  let *restype* = *definingSort*(*procedure*) **in**
  if *procedure* ∈ *Literal-signature* **then**
    mk-*SDLCHARACTER*(*procedure*.**s-***Literal-name*, *restype*)
  elseif *procedure*.*procName* = "num" **then**
    mk-*SDLINTEGER*(*charValue*(*values*.*head*.**s-***Name*), *IntegerType*)
  elseif *procedure*.*procName* = "chr" **then**
    mk-*SDLCHARACTER*(*values*.*head*.*semvalueInt*.*charChr*, *restype*)
  else *raise*(*OutOfRange*)
  endif
  endlet

The function *charValue* returns the numerical value of the character.

charValue(*ch*: *Name*): *NAT* $=_{\text{def}}$
  let *myDef*: *Value-data-type-definition* = *CharacterType*.*idToNodeAS1* **in**
  let *literals* = *myDef*.**s-***Literal-signature***-set in**
    take({*L*.**s-***Literal-natural* | *L* ∈ *literals*: *L*.**s-***Literal-name* = *ch*})
  endlet

The function *charChr* returns the character for a given Integer.

charChr(*a*: *NAT*): *Name* $=_{\text{def}}$
  if *a* > 128 **then** *charChr*(*a* - 128)
  elseif *a* < 0 **then** *charChr*(*a*+128)
  else
    let *char*: *Value-data-type-definition* = *CharacterType*.*idToNodeAS1* **in**
    let *literals* = *char*.**s-***Literal-signature***-set in**
      take({*L*.**s-***Literal-name* | *L* ∈ *literals*: *L*.*Literal-natural* = *a*})
  endif

F3.3.1.5 Real

The Predefined type Real is represented as a rational number, with numerator and denominator.

SDLREAL $=_{\text{def}}$ *NAT* × *NAT* × *Identifier*

semvalueRealNum(*v*: *SDLREAL*): *NAT* $=_{\text{def}}$ *v*.**s-***NAT*

semvalueRealDen(*v*: *SDLREAL*): *NAT* $=_{\text{def}}$ *v*.**s2-***NAT*

semvalueReal(*v*: *SDLREAL*): *REAL*$=_{\text{def}}$
  let *res*: *REAL* = *v*.*semvalueRealNum* / *v*.*semvalueRealDen* **in**
    res
  endlet

computeReal(*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUEOREXCEPTION* $=_{\text{def}}$
  let *restype* = *definingSort*(*procedure*) **in**
  if *procedure* ∈ *Literal-signature* **then**
    realLiteral(0,1,*procedure*.*procName*, *restype*)
  elseif *procedure*.*procName* = "-" ∧ *values*.*length* = 1 **then**
    mk-*SDLREAL*(0 - *values*.*head*.*semvalueRealNum*, *values*.*head*.*semvalueRealDen*, *restype*)
  elseif *procedure*.*procName* ∈ {"+", "-", "*", "/", "<", ">", "<=", ">="} **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 procedure.procName of
            | "+" => mk-SDLReal(num1*den2 + num2*den1, den1*den2, restype)
            | "-"=> mk-SDLReal(num1*den2 - num2*den1, den1*den2, restype)
            | "*"=> mk-SDLReal(num1*num2, den1*den2, restype)
            | "/"=>
                if num2 = 0 then
                    raise(DivisionByZero)
                else
                    mk-SDLReal(num1*num2, den1*den2, restype)
                endif
            | "<" => mk-SDLBoolean(num1*den2 < num2*den1, BooleanType)
            | "<=" => mk-SDLBoolean(num1*den2 ≤ num2*den1, BooleanType)
            | ">" => mk-SDLBoolean(num1*den2 ≥ num2*den1, BooleanType)
            | ">="=> mk-SDLBoolean(num1*den2 ≥ num2*den1, BooleanType)
            endcase
        endlet
    elseif procedure.procName = "float" then
        mk-SDLReal(semvalueInt(values.head), 1, restype)
    elseif procedure.procName = "fix" then
        mk-SDLInteger(computeFix(values.head.semvalueRealNum,
            values.head.semvalueRealDen), IntegerType)
    else raise(OutOfRange)
    endif
    endlet
```

The function *realLiteral* returns the *SDLReal* value for a real literal.

```
realLiteral(num: Nat, den: Nat, proc: Token, type: Identifier): SDLReal =def
    if proc = emptyToken then
        mk-SDLReal(num, den, type)
    elseif proc.head = "." then
        realLiteral(num*10,den*10, proc.tail, type )
    elseif den = 1 then
        realLiteral(num*10 + numberValue(proc.head), den, proc.tail, type)
    else
        realLiteral(num*10 + numberValue(proc.head), den, proc.tail, type)
    endif
```

The function *computeFix* returns the *Nat* value given numerator and denominator.

```
computeFix(num: Nat, den: Nat): Nat =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 *SDLReal*.

```
SDLDuration =def Duration × Identifier

computeDuration(procedure: Procedure, values: Value*): Value =def
    computeReal(procedure, values)
```

F3.3.1.7 Time

The domain *SDLTime* is based on the domain *SDLReal*.

```
SDLTime=def Time × Identifier
```

```
computeTime(procedure: PROCEDURE, values: VALUE*): VALUE =def
    let restype = definingSort(procedure) in
    if procedure ∈ Literal-signature then
        realLiteral(0,1,procedure.procName, restype)
    else
    case procedure.procName of
        | "time"=>
            let val: SDLREAL = values.head in
                mk-SDLREAL(val.s-NAT, val.s2-NAT, RealType)
            endlet
        | "<" => computeReal(procedure, values)
        | "<=" => computeReal(procedure, values)
        | ">" => computeReal(procedure, values)
        | ">=" => computeReal(procedure, values)
        | "+" => computeReal(procedure, values)
        | "-" =>
            if values.head ∈ SDLTIME ∧ values.tail.head ∈ SDLDURATION then
                computeReal(procedure, values)
            else
                let res: SDLREAL = computeReal(procedure,values) in
                    mk-SDLREAL(res.s-NAT, res.s2-NAT, RealType)
                endlet
            endif
    endcase
    endif
    endlet
```

F3.3.1.8 String

A string type is defined as a sequence of its element type.

```
SDLSTRING =def VALUE * × Identifier

computeString (procedure: PROCEDURE, values: VALUE*): VALUEOREXCEPTION =def
let restype = definingSort(procedure) in
    case procedure.procName of
    | "emptystring"=>mk-SDLSTRING(empty, restype)
    | "mkstring"=>    mk-SDLSTRING(<values.head>, restype)
    | "make"=>    mk-SDLSTRING(<values.head>, restype)
    | "length"=>   mk-SDLINTEGER (values.head. s-VALUE-seq.length, IntegerType)
    | "first"=> values.head. s-VALUE-seq.head
    | "last"=>  values.head. s-VALUE-seq.last
    | "//"=>     mk-SDLSTRING(values[1]. s-VALUE-seq ⁀ values[2].s-VALUE-seq, restype)
    | "extract"=>
        let string = values[1]. s-VALUE-seq in
        let intval: SDLINTEGER = values[2] in
        let index = intval.s-NAT in
            if index < 0 ∨ index > string.length then
                    raise(InvalidIndex)
            else
                string[index]
            endif
        endlet
    | "modify"=>
            let intval: SDLINTEGER = values[2] in
            let index = intval.s-NAT 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
    | "substring"=>
                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
    | "remove"=>
            let intval: SDLINTEGER = values[2] in
            let index = intval.s-NAT 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
    endcase
endlet
```

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 Array

An array is represented as a set of index/itemsort pairs, with an optional default value.

```
SDLARRAY =def VALUEPAIR-set × [VALUE] × Identifier

VALUEPAIR =def VALUE × VALUE

computeArray(procedure: PROCEDURE, values: VALUE*): VALUEOREXCEPTION =def
    let  restype = definingSort(procedure) in
    if procedure.procName = "Make" then
        if values.length = 0 then
            mk-SDLARRAY(∅, undefined, restype)
        else
            mk-SDLARRAY(∅,values.head, restype)
        endif
    elseif procedure.procName = "Modify" then
        let a = values[1], index = values[2], value = values[3] in
            mk-SDLARRAY(modifyArray(a.s-VALUEPAIR-set, index, value), a.s-VALUE, restype)
        endlet
    elseif procedure.procName = "Extract" then
        let v = take({ f.s2-VALUE | f ∈ values[1].s-VALUEPAIR-set: f.s-VALUE = values[2]}) in
            if v = undefined then
                if values[1].s-VALUE = undefined then
                    raise(InvalidIndex)
                else
                    values[1].s-VALUE
            else
                v
        endlet
    else raise(OutOfRange)
```

endif
endlet

modifyArray(*a*: *VALUEPAIR*-**set**, *index*: *VALUE*, *value*: *VALUE*): *VALUEPAIR*-**set** $=_{def}$
{ *item* | *item* ∈ *a*: *item*.**s-***VALUE* ≠ *index* } ∪ { **mk-***VALUEPAIR*(*index*,*value*)}

F3.3.1.10 Powerset

A Powerset is represented as a set.

SDLPOWERSET $=_{def}$ *VALUE*-**set** × *Identifier*

computePowerset (*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUEOREXCEPTION* $=_{def}$
let *restype* = *definingSort*(*procedure*) **in**
case *procedure*.*procName* **of**
| "empty"=> **mk-***SDLPOWERSET*(∅,*restype*)
| "in"=> **mk-***SDLBOOLEAN*(*values*[1] ∈ *values*[2].**s-***VALUE*-**set**, *BooleanType*)
| "incl"=> **mk-***SDLPOWERSET*(*values*[2].**s-***VALUE*-**set** ∪ {*values*[1] }, *restype*)
| "del"=> **mk-***SDLPOWERSET*(*values*[2].**s-***VALUE*-**set** \ {*values*[1] }, *restype*)
| "<"=> **mk-***SDLBOOLEAN*(*values*[1].**s-***VALUE*-**set** ⊂ *values*[2].**s-***VALUE*-**set**, *BooleanType*)
| "<="=> **mk-***SDLBOOLEAN*(*values*[1].**s-***VALUE*-**set** ⊆ *values*[2].**s-***VALUE*-**set**, *BooleanType*)
| ">"=> **mk-***SDLBOOLEAN*(*values*[2].**s-***VALUE*-**set** ⊂ *values*[1].**s-***VALUE*-**set**, *BooleanType*)
| ">="=> **mk-***SDLBOOLEAN*(*values*[2].**s-***VALUE*-**set** ⊆ *values*[1].**s-***VALUE*-**set**, *BooleanType*)
| "and"=> **mk-***SDLPOWERSET*(*values*[1].**s-***VALUE*-**set** ∩ *values*[2].**s-***VALUE*-**set**, *restype*)
| "or"=> **mk-***SDLPOWERSET*(*values*[1].**s-***VALUE*-**set** ∪ *values*[2].**s-***VALUE*-**set**, *restype*)
| "length"=> **mk-***SDLINTEGER*(| *values*[1].**s-***VALUE*-**set** |, *IntegerType*)
| "take"=> **if** *values*[1].**s-***VALUE*-**set** = ∅ then
raise(*EmptyException*)
else
values[1]. **s-***VALUE*-**set**.*take*
endif
endcase
endlet

F3.3.1.11 Bag

A Bag is represented as a set of value-frequency pairs.

SDLBAG $=_{def}$ *FREQUENCY*-**set** × *Identifier*

FREQUENCY $=_{def}$ *VALUE* × *NAT*

computeBag (*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUEOREXCEPTION* $=_{def}$
let *restype* = *definingSort*(*procedure*) **in**
case *procedure*.*procName* **of**
| "empty"=> **mk-***SDLBAG*(∅,*restype*)
| "in"=> **mk-***SDLBOOLEAN*(*bagcount*(*values*[1], *values*[2]) ≠ 0, *BooleanType*)
| "incl"=> **mk-***SDLBAG*(*bagincl*(*values*[1], *values*[2]), *restype*)
| "del"=> **mk-***SDLBAG*(*bagdel*(*values*[1], *values*[2]), *restype*)
| "<"=> **mk-***SDLBOOLEAN*(*baginbag*(*values*[1], *values*[2]), *BooleanType*)
| "<="=> **mk-***SDLBOOLEAN*(¬ *baginbag*(*values*[2], *values*[1]), *BooleanType*)
| ">"=> **mk-***SDLBOOLEAN*(*baginbag*(*values*[2], *values*[1]), *BooleanType*)
| ">="=> **mk-***SDLBOOLEAN*(¬ *baginbag*(*values*[1], *values*[2]), *BooleanType*)
| "and"=> **mk-***SDLBAG*(*bagand*(*values*[1], *values*[2]), *restype*)
| "or"=> **mk-***SDLBAG*(*bagor*(*values*[1], *values*[2]), *restype*)
| "length"=> **mk-***SDLINTEGER*(*baglength*(*values*[1].**s-***FREQUENCY*-**set**), *IntegerType*)
| "take"=> *values*[1].**s-***FREQUENCY*-**set**.*take*.**s-***VALUE*
endcase
endlet

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

bagincl(*item*: *VALUE*, *bag*: *SDLBAG*): *FREQUENCY*-**set** $=_{\text{def}}$
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** $=_{\text{def}}$
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* $=_{\text{def}}$
∀ *elem* ∈ *smaller*.**s-***FREQUENCY*-**set**: *bagcount*(*elem*.**s-***VALUE*, *larger*) < *elem*.**s-***NAT*

bagand(*a*: *SDLBAG*,*b*: *SDLBAG*): *FREQUENCY*-**set** $=_{\text{def}}$
{ **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* $=_{\text{def}}$ **if** *a*>*b* **then** *a* **else** *b* **endif**

bagor(*a*: *SDLBAG*,*b*: *SDLBAG*): *FREQUENCY*-**set** $=_{\text{def}}$
{ **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* $=_{\text{def}}$
if *a* = ∅ **then** 0
else let *x* = *a*.*take* **in**
x.**s-***NAT* + *baglength*(*a* \ {*x*})
endlet
endif

F3.3.2 Pid types

A *PID* value is represented by an agent and an interface.

PID $=_{\text{def}}$ *VALIDPID* ∪ *NULLPID*
NULLPID $=_{\text{def}}$ { **mk-***Null-literal-signature*(**mk-***Name*("null"), *Pidtype*, *undefined*) }
VALIDPID $=_{\text{def}}$ *SDLAGENT* × [*Interface-definition*]

static *nullPid*: *PID* $=_{\text{def}}$ *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*.

F3.3.3 Constructed types

F3.3.3.1 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 Make operator uses the order of the fields rather than the field names.

SDLSTRUCTURE $=_{\text{def}}$ *FIELD** × *Identifier*
FIELD $=_{\text{def}}$ *Name* × *VALUE*

isSpecialStructOp(*procedure*: *PROCEDURE*): *BOOLEAN* $=_{\text{def}}$
let *procsort* = *procedure*.*definingSort*, *pn* = *procedure*.*procName* **in**
(∃ *str* ∈ *SDLSTRUCTURE*: (*procsort* = *str*.**s-***Identifier*)) ∧
((*pn* = "Make")
∨ (*pn* = "Undefined")
∨ (∃ *fld* ∈ *procsort*.**s-***FIELD***-seq**: (*pn* = *fld*.**s-***Name* ⁀ "Modify"))
∨ (∃ *fld* ∈ *procsort*.**s-***FIELD***-seq**: (*pn* = *fld*.**s-***Name* ⁀ "Extract"))
∨ (∃ *fld* ∈ *procsort*.**s-***FIELD***-seq**: (*pn* = *fld*.**s-***Name* ⁀ "Present"))

The function *computeStruct* gives the value of applying the language-defined operators for structures.

computeStruct(*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUEOREXCEPTION* $=_{\text{def}}$
let *structsort* = *definingSort*(*procedure*), *pn* = *procedure*.*procName* **in**
if *pn* = "Undefined" **then**
structUndefined(*structsort*)
elseif *pn* = "Make " **then**
structMake(*structsort*, *empty*, *structsort*.**s-***FIELD***-seq**, *values*)
elseif (∃ fld ∈ *structsort*.**s-***FIELD***-seq**: (*pn* = fld.**s-***Name* ⁀ "Modify") **then**
let *fn* ⁀ "Modify" = *pn* **in**
structModify(*fn*, *structsort*, *values*.*head*, *empty*, *structsort*.**s-***FIELD***-seq**)
endlet
elseif (∃ fld ∈ *structsort*.**s-***FIELD***-seq**: (*pn* = fld.**s-***Name* ⁀ "Extract") **then**
let *fn* ⁀ "Extract" = *pn* **in**
structExtract(*fn*, *structsort*)
endlet
elseif (∃ fld ∈ *structsort*.**s-***FIELD***-seq**: (*pn* = fld.**s-***Name* ⁀ "Present")) **then**
let *fn* ⁀ "Present" = *pn* **in**
structFieldPresent(*fn*, *structsort*)
endlet
else *raise*(*OutOfRange*)
endif
endlet

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 value, 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. The new fields (*newflds*) and old fields (*oldflds*) are used in the internal recursion.

structMake(*st*: *SDLSTRUCTURE*, *newflds*: *FIELD**, *oldflds*: *FIELD**, *values*: *VALUE**): *VALUE* $=_{\text{def}}$
if *values*.*length* < *oldflds*.*length* **then** *structMake*(*st*, *newflds*, *oldflds*, *values* ⁀ *undefined*)
elseif *values*.*length* = 0 ∨ *oldflds*.*length* = 0 **then**
mk-*SDLSTRUCTURE*(*newflds*, *st*.**s-***Identifier*)
else
structMake(*st*, *newflds* ⁀ **mk-***FIELD*(*oldflds*.*head*.**s-***Name*, *values*.*head*),
oldflds.*tail*, *values*.*tail*)
endif

The function *structUndefined* returns the true if (and only if) all the fields are undefined.

structUndefined(*st*: *SDLSTRUCTURE*): *SDLBOOLEAN* $=_{\text{def}}$
mk-*SDLBOOLEAN*(*semvalueBool*(∀*value* ∈ *st*.**s-***FIELD*.**s-***VALUE*: (*value* = *undefined*)), *BooleanType*)

The function *structExtract* returns the field with a given name from a list of fields.

structExtract(*fieldname*:*Name*, *structtype*: *SDLSTRUCTURE*): *VALUE* $=_{\text{def}}$
let *valueset* = { *f*.**s-***VALUE* | *f* ∈ *structtype*.**s-***FIELD***-seq**: *f*.**s-***Name* = *fieldname*} **in**
if *valueset* = ∅ **then** *raise*(*UndefinedField*)

```
        else valueset.take
        endif
    endlet
```

The function *structModify* returns a new structure with one field changed. It should be called externally (internally it is recursive) with the field name, 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(fn: Name, 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 = fieldname then val else oldflds.head.s-VALUE endif),
            oldflds.tail)
    endif
```

The function *structFieldPresent* returns the true if the specified field has a value.

```
structFieldPresent(fn: Name, st: SDLSTRUCTURE): SDLBOOLEAN =def
    mk-SDLBOOLEAN(semvalueBool(fn.parentAS1.s-FIELD.s-VALUE ≠ undefined), BooleanType)
```

F3.3.3.2 Literals

Values of a literal sort are represented by the type in which the literal is defined, and the literal signatures:

```
SDLLITERALS =def Literal-signature × Identifier

isSpecialLiteralOp(procedure: PROCEDURE): BOOLEAN =def
    let procsort = procedure.definingSort, pn = procedure.procName in
    (∃ lit ∈ SDLLITERALS: (procsort = lit.s-Identifier )) ∧
    (    pn ∈ { "<", ">","<=",">=", "first", "last", "succ", "pred", "num" })
```

The function *computeLiteral* gives the value of applying the language-defined operators for structures.

```
computeLiteral(procedure:PROCEDURE, values:VALUE*): [VALUE ]=def
    let restype = definingSort(procedure) in
    let defi: Value-data-type-definition = restype.idToNodeAS1 in
    if procedure.procName ∈ { "<", ">","<=",">=" } then
        let v1 = values.head.s-Literal-signature.literalNum in
        let v2 = values.tail.head.s-Literal-signature.literalNum in
        case procedure.procName of
        | ">" => mk-SDLBOOLEAN(v1 > v2, BooleanType)
        | ">=" => mk-SDLBOOLEAN(v1 ≥ v2, BooleanType)
        | "<" => mk-SDLBOOLEAN(v1 < v2, BooleanType)
        | "<=" => mk-SDLBOOLEAN(v1 ≤ v2, BooleanType)
        endcase
        endlet
    elseif procedure.procName = "first" then
        literalMinimum (defi.s-Literal-signature-set)
    elseif procedure.procName = "last" then
        literalMaximum (defi.s-Literal-signature-set)
    elseif procedure. procName = "succ" then
        literalSucc(defi.s-Literal-signature-set, values.head)
    elseif procedure. procName = "pred" then
        literalPred(defi.s-Literal-signature-set, values.head)
```

elseif *procedure*. *procName* = "num" **then**
  mk-*SDLINTEGER*(*literalNum*(*values*.*head*).*semvalueInt*, *IntegerType*)
else
  undefined
endif
endlet

literalNum(*s*: *Literal-signature*): *NAT* $=_{\text{def}}$
  s.**s-***Literal-natural*

literalValue(*s*: *Literal-signature*): *VALUE* $=_{\text{def}}$
  mk-*SDLLITERALS*(*s*, *s*.**s-***Result*)

literalMinimum(*s*: *Literal-signature*-**set**): *VALUE* $=_{\text{def}}$
  take({*s1*.*literalValue*
    | *s1* ∈ *s*: ∀ *s2*∈*s*:*s2*.*literalNum*> *s1*.*literalNum*})

literalMaximum(*s*: *Literal-signature*-**set**): *VALUE* $=_{\text{def}}$
  take({*s1*.*literalValue*
    | *s1* ∈ *s*: ∀ *s2*∈*s*:*s2*.*literalNum* < *s1*.*literalNum*})

literalSucc(*s*: *Literal-signature*-**set**, *val*: *SDLLITERALS*): *VALUE* $=_{\text{def}}$
  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* ≤ *s*.*literalNum*) ∨ (*s1*.*literalNum* ≤ *s2*.*literalNum*))}
  endif

literalPred(*s*: *Literal-signature*-**set**, *val*: *SDLINTEGER*): *VALUE* $=_{\text{def}}$
  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* ≤ *s1*.*literalNum*) ∨ (*s*.*literalNum* ≤ *s2*.*literalNum*))}
  endif

F3.3.3.2 Choice

Further study is required for this subject.

F3.3.4 Variables with Aggregation-kind REF

Further study is required for this subject.

F3.3.5 State access

The *STATE* domain consists of substates (associations of values for a specific *STATEID*), and 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**

NAMEDVALUE $=_{\text{def}}$ *STATEID* × *Variable-identifier* × [*BOUNDVALUE*]

BOUNDVALUE $=_{\text{def}}$ *VALUE* ∪ *Variable-identifier*

SUPERSTATE $=_{\text{def}}$ *STATEID* × *STATEID*

initAgentState(*state*: [*STATE*], *newid*: *STATEID*, *id*: [*STATEID*], *declarations*: *DECLARATION*-**set**): *STATE* $=_{\text{def}}$
  let *newsub* = *initDeclarations*(*newid*, *declarations*) in
  if *state* = *undefined* **then**

```
        mk-STATE(newsub, ∅, ∅)
    else
    let newsuper = if id = undefined then ∅ else { mk-SUPERSTATE(id, newid)} endif in
        mk-STATE(state.s-NAMEDVALUE-set ∪ newsub, state.s-SUPERSTATE-set ∪ newsuper)
    endif
    endlet
```

```
initProcedureState(state: STATE, newid: STATEID, id: STATEID, vars: DECLARATION-set,
    declarations: DECLARATION*,
    values:VALUE*, variables: Variable-identifier*): STATE =def
    let newsub = assignValues(initDeclarations(newid, vars ∪ declarations.toSet),
                              newid,declarations,
                              values, variables) in
    let newsuper = mk-SUPERSTATE(id, newid) in
        mk-STATE(state.s-NAMEDVALUE-set ∪ newsub, state.s-SUPERSTATE-set ∪ { newsuper })
    endlet
```

```
initDeclarations(newid: STATEID, decls: DECLARATION-set): NAMEDVALUE-set =def
    { mk-NAMEDVALUE(newid, d.identifier1, d.s-Constant-expression)
        | d ∈ decls: d ∈ Variable-definition} ∪
    { mk-NAMEDVALUE(newid, d.identifier1,
        undefined)
        | d ∈ decls: d ∈ Procedure-formal-parameter}
```

The function *assignValues* puts a sequence of parameter values into a named values set for a given state id.

```
assignValues(namedvalues:NAMEDVALUE-set, id: STATEID, decls:DECLARATION*,
    values:VALUE*, variables:Variable-identifier*): NAMEDVALUE-set =def
    if values = empty then
        namedvalues
    else
    if decls.head ∈ In-parameter then
        assignValues(setValue(namedvalues, id, variables.head, values.head),
                     id, decls.tail, values.tail, variables.tail)
    else
        assignValues(namedvalues, id, decls.tail, values.tail, variables.tail)
    endif
```

The function *setValue* puts a single value into a named values set for a given state id.

```
setValue(namedvalues: NAMEDVALUE-set, id: STATEID, varname:Identifier, value:VALUE):
    NAMEDVALUE-set =def
    { binding | binding ∈ namedvalues:
        binding.s-Variable-identifier ≠ varname ∨ binding.s-STATEID ≠ id} ∪
    { mk-NAMEDVALUE(id, varname, value) }
```

The function *getValue* returns the association between *id* and *varname* in *namedvalues*.

```
getValue(namedvalues: NAMEDVALUE-set, id: STATEID, varname:Identifier): NAMEDVALUE-set =def
    { b ∈ namedvalues:
        b.s-STATEID = id ∧ b.s-Variable-identifier = varname}
```

The function *eval* returns the value associated with a state, a state id, and a name. If there is named value for the state and identified variable, there can be at most one. 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 must be a variable in the caller (the state id that caused this state id to exist), because static semantics ensures each variable exists. 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 the bound value is *undefined*, and *undefined* returned. If no named value is associated, the static semantics ensures the variable exists, so the identified variable must be associated with the caller (the state id that caused this state id to exist). In this case *eval* is called recursively to return the value (in the named values for the state) for the given variable and the caller state.

```
eval(varname:Identifier, state:STATE, id:STATEID): VALUEOREXCEPTION =def
    let callerid = caller(state, id) in
        let namedval = getValue(state.s-NAMEDVALUE-set, id, varname) in
            if namedval ≠ ∅ then
                if namedval.take.s-BOUNDVALUE ∈ VALUE then
                    namedval.take.s-BOUNDVALUE
                elseif namedval.take.s-BOUNDVALUE ∈ Variable-identifier then
                    eval(namedval.take.s-BOUNDVALUE, state, callerid)
                else // the BOUNDVALUE is undefined
                    raise(UndefinedVariable)
                endif
            else
                eval(varname, state, callerid)
            endif
        endlet
    endlet
```

The function *update* modifies a binding of a name to a value.

```
update(name:Identifier, value:VALUE, state:STATE, id:STATEID): STATE =def
    let val = getValue(state.s-NAMEDVALUE-set, id, name) in
        if val = ∅ then
            update(name, value, state, caller(state, id))
        elseif val.take ∈ NAMEDVALUE then
            mk-STATE(setValue(state.s-NAMEDVALUE-set, id, name, value),
                        state.s-SUPERSTATE-set)
        else
            update(val.take.s-Variable-identifier, value, state, id)
        endif
    endlet
```

The function *assign* modifies the variable with the given name in the state/id association to the given value.

```
assign (variablename:Variable-identifier, value:VALUE, state:STATE, id:STATEID): STATEOREXCEPTION =def
    if isValueVariable(variablename) then
        if isSyntypeVariable(variablename) ∧ ¬rangeCheck(variablename.variableSort, value ) then
            raise(OutOfRange)
        else update(variablename, value, state, id)
        endif
    else
        // pid variable, sort of variable is an Interface-definition
        if variablename.variableSort = value.interface ∨
            isSuperType(variablename.variableSort, value.interface) then
            update(variablename, value, state, id)
        else
            update(variablename, 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.s-STATEID | s ∈ state.s-SUPERSTATE-set: s.s2-STATEID = id})
```

The function *variableSort* returns the sort for a given variable identifier.

```
variableSort(variableid: Variable-identifier): Data-type-definition =def
```

variableid.*idToNodeAS1*.**s-***Sort-reference-identifier*.*idToNodeAS1*

The predicate *isValueVariable* holds if the *variablename* refers to a variable of a value type.

isValueVariable(*variableid*: *Variable-identifier*): *BOOLEAN* $=_{\text{def}}$
variableid.*variableSort* ∈ *Value-data-type-definition*

The predicate *isSyntypeVariable* holds if the *variablename* refers to a variable with a syntype.

isSyntypeVariable(*variableid*: *Variable-identifier*): *BOOLEAN* $=_{\text{def}}$
variableid.*idToNodeAS1*.**s-***Sort-reference-identifier* ∈ *Syntype-identifier*

interface(*val*: *VALUE*): *Interface-definition* $=_{\text{def}}$
if *val*.*sort* ∈ *Interface-definition* **then** *val*.*sort* **else** *undefined* **endif**

The function *sort* 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): Data-type-definition =def
if      val ∈ NULLPID then PidType.idToNodeAS1
elseif  val ∈ VALIDPID then
    if val.s-Interface-definition = undefined then PidType.idToNodeAS1
    else val.s-Interface-definition
    endif
else val.s-Identifier.idToNodeAS1
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
    | SDLBOOLEAN(*,t)       => t
    | SDLINTEGER(*, t)      => t
    | SDLCHARACTER(*, t)    => t
    | SDLREAL(*,*, t)       => t
    | SDLSTRING(*,t)        => t
    | SDLLITERALS(*,t)      => t
    | SDLSTRUCTURE(*,t)     => t
    | PID(*, t)         => t
    endcase
endif
```

F3.3.7 Operators and methods

The function *dispatch* determines the procedure to select given a set of actual parameters.

```
dispatch(procedure:PROCEDURE, values:VALUE*): Identifier =def
    if procedure ∈ Static-operation-signature then
        procedure.s-Identifier
    else
        let c = allDynamicCandidates(procedure) in
        let c1 = matchingCandidates(c, values) in
                bestMatch(c1)
        endlet
    endif
```

The function *allDynamicCandidates* returns the set of all signatures with the same name as the given signature.

allDynamicCandidates(*procedure*:*PROCEDURE*): *PROCEDURE*-**set** $=_{\text{def}}$
{ *p* | *p* ∈*Operation-signature*:
p.**s**-*Operation-name* = *procedure*.**s**-*Operation-name* }

The function *matchingCandidates* returns the set of all signatures that are compatible with the arguments.

matchingCandidates(*procedures*: *PROCEDURE*-**set**, *values*: *VALUE**): *PROCEDURE*-**set** $=_{\text{def}}$
{ *p* | *p* ∈ *procedures*: *isSignatureCompatible*(*p*.**s**-*Formal-argument*-**seq**, *dynamicTypes*(*values*)) }

The function *matchingCandidates* returns the most specialized signature.

bestMatch(*procedures*:*PROCEDURE*-**set**): *Identifier* $=_{\text{def}}$
take({ *p*.**s**-*Identifier* | *p* ∈ *procedures*:
∀ *q* ∈ *procedures*: *isSignatureCompatible*(*p*.**s**-*Formal-argument*-**seq**,
q.**s**-*Formal-argument*-**seq**) })

The predicate *isSignatureCompatible* holds if p is compatible with q.

isSignatureCompatible(*p*:*Formal-argument**, *q*:*Formal-argument**): *BOOLEAN* $=_{\text{def}}$
if *p* = *empty* **then**
true
else
isSortCompatible(*p*.*head*.**s**-*Argument*, *q*.*head*.**s**-*Argument*) ∧
isSignatureCompatible(*p*.*tail*, *q*.*tail*)
endif

isSortCompatible(*p*: *Sort-reference-identifier*, *r*: *Sort-reference-identifier*): *BOOLEAN* $=_{\text{def}}$
(*p* = *r*) ∨
isDirectlySortCompatible(*p*, *r*) ∨
(*r*.*idToNodeAS1* ∈ *Interface-definition* ∧
(∃ q ∈ *Sort-reference-identifier*: (*isSortCompatible*(*p*, *q*) ∧ *isSortCompatible*(*q*, *r*))))

isDirectlySortCompatible(*y*: *Sort-reference-identifier*, *z*: *Sort-reference-identifier*): *BOOLEAN* $=_{\text{def}}$
if *isSuperSort*(*z*, *y*) **then**
if *y*.*idToNodeAS1* ∈ *Value-data-type-definition* **then**
// true if y is <anchored sort> of the form **this** z
y.*idToNodeAS1*.**s**-*Data-type-identifier* = *z*
else // y is a pid sort (because not a value dat type) – and z is super sort of y
true
endif
else *false*
endif

isSuperSort(*z Sort-reference-identifier*, *y*: *Sort-reference-identifier*): *BOOLEAN* $=_{\text{def}}$
isSuperType(*z*, *y*) // see clause F2.2.1.6.4.

dynamicTypes(*values*:*VALUE**): *Formal-argument** $=_{\text{def}}$
<**mk**-*Formal-argument*(*dynamicType*(*v*)) | *v* **in** *values* >

F3.3.8 Syntypes

The predicate *rangeCheck* holds if the range check for a value of a *syntype* passes.

rangeCheck(*syntype*: *Syntype-definition*, *value*: *VALUE*): *BOOLEAN* $=_{\text{def}}$
∃ *cond* ∈ *syntype*.**s**-*Range-condition*.**s**-*Condition-item*-**set**:
conditionItemCheck(*cond*, *value*, *syntype*.**s**-*Parent-sort-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*: *Condition-item*, *value*: VALUE, *type*: *Identifier*): BOOLEAN $=_{\text{def}}$

if *cond* ∈ *Open-range* **then**

semvalueBool(*compute*(*cond*.**s-***Open-range*.**s-***Operation-identifier*,

< *cond*.**s-***Open-range*.**s-***Constant-expression* >))

elseif *cond* ∈ *Closed-range* **then**

choose *lessthaneq*: *lessthaneq* ∈ *type*.**s-***Static-operation-signature***-set** ∧ *lessthaneq*.*procName* = "<="

semvalueBool(*compute*(*lessthaneq*, < *cond*.**s-***Closed-range*.**s-***Constant-expression*, *value* >)) ∧

semvalueBool(*compute*(*lessthaneq*, < *value*, *cond*.**s-***Closed-range*.**s2-***Constant-expression* >))

endchoose

else //size constraint and *cond* ∈ *Size-constraint*

semvalueBool(*compute*(*cond*.**s-***Size-constraint*.**s-***Operation-identifier*, < *value* >))

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], [ITU-T Z.102] or [ITU-T Z.104]) where the grammar rule is defined.

Action-return-node
Agent-definition
Agent-identifier
Agent-kind
Agent-type-definition
Agent-type-identifier
Any-expression
Argument
Assignment
Break-node
Call-node
Channel-definition
Channel-path
Closed-range
Composite-state-graph
Composite-state-type-definition
Composite-state-type-identifier
Compound-node
Condition-item
Conditional-expression
Connect-node
Connection-definition
Connector-name
Constant-expression
Continue-node
Continuous-expression
Continuous-signal
Create-request-node
Dash-nextstate
Data-type-definition
Data-type-name
Decision-answer
Decision-node
Destination-gate
Entry-connection-definition
Entry-procedure-definition
Equality-expression
Exception-identifier
Exit-connection-definition
Exit-procedure-definition
Formal-argument
Free-action
Gate-definition
Graph-node
Identifier
In-parameter
In-signal-identifier
Initial-number
Inner-entry-point
Inner-exit-point
Input-node
Interface-definition
Join-node

Literal
Literal-name
Literal-natural
Literal-signature
Maximum-number
Name
Named-nextstate
Named-return-node
Named-start-node
Nextstate-parameters
Now-expression
Number-of-instances
Null-literal-signature
Offspring-expression
Open-range
Operation-application
Operation-identifier
Operation-name
Operation-signature
Originating-gate
Out-parameter
Out-signal-identifier
Outer-entry-point
Outer-exit-point
Output-node
Package-name
Package-qualifier
Parameter
Parent-expression
Parent-sort-identifier
Priority-name
Procedure-definition
Procedure-formal-parameter
Procedure-graph
Procedure-identifier
Procedure-start-node
Provided-expression
Qualifier
Range-check-expression
Range-condition
Reset-node
Result
Save-signalset
Self-expression
Sender-expression
Set-node
Signal-definition
Signal-identifier
Size-constraint
Sort
Sort-identifier
Sort-reference-identifier
Spontaneous-transition
State-aggregation-node
State-entry-point-name
State-exit-point-name
State-machine
State-name
State-node
State-partition
State-start-node
State-transition-graph
Static-operation-signature

Stop-node
Syntype-identifier
Syntype-definition
Terminator
Timer-active-expression
Transition
Value-data-type-definition
Value-return-node
Value-returning-call-node
Variable-access
Variable-definition
Variable-identifier

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series D	General tariff principles
Series E	Overall network operation, telephone service, service operation and human factors
Series F	Non-telephone telecommunication services
Series G	Transmission systems and media, digital systems and networks
Series H	Audiovisual and multimedia systems
Series I	Integrated services digital network
Series J	Cable networks and transmission of television, sound programme and other multimedia signals
Series K	Protection against interference
Series L	Construction, installation and protection of cables and other elements of outside plant
Series M	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	Terminals and subjective and objective assessment methods
Series Q	Switching and signalling
Series R	Telegraph transmission
Series S	Telegraph services terminal equipment
Series T	Terminals for telematic services
Series U	Telegraph switching
Series V	Data communication over the telephone network
Series X	Data networks, open system communications and security
Series Y	Global information infrastructure, Internet protocol aspects and next-generation networks
Series Z	**Languages and general software aspects for telecommunication systems**

Printed in Switzerland
Geneva, 2015