INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100

Annex F3

(11/2000)

SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS

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

Specification and description language (SDL)

Annex F3: SDL formal definition: Dynamic semantics

ITU-T Recommendation Z.100 – Annex F3

(Formerly CCITT Recommendation)

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	Z.120–Z.129
PROGRAMMING LANGUAGES	
CHILL: The ITU-T programming 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.399
QUALITY OF TELECOMMUNICATION SOFTWARE	Z.400–Z.499
METHODS FOR VALIDATION AND TESTING	Z.500–Z.599

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

ITU-T Recommendation Z.100

Specification and description language (SDL)

ANNEX F3

Dynamic semantics

Summary

This Annex defines the dynamic semantics of SDL.

Source

Annex F3 to ITU-T Recommendation Z.100 was prepared by ITU-T Study Group 10 (2001-2004) and approved under the WTSA Resolution 1 procedure on 24 November 2000.

FOREWORD

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

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

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

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

NOTE

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

INTELLECTUAL PROPERTY RIGHTS

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

As of the date of approval of this Recommendation, ITU had not received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementors are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database.

CONTENTS

Page

ITU-T Recommendation Z.100

Specification and description language (SDL)

ANNEX F3

Dynamic semantics

1 General information

In order to define the formal semantics of SDL, the language definition is decomposed into several parts:

- grammar,
- well-formedness,
- transformation rules, and
- dynamic semantics.

Starting point for defining the formal semantics of SDL is a syntactically correct SDL specification, represented as an abstract syntax tree (AST).

The first three parts of the formal semantics are collectively referred to as *static semantics* in the context of SDL.

The *grammar* defines the set of syntactically correct SDL specifications. In ITU-T Z.100, a concrete textual, a concrete graphical, and an abstract grammar are defined formally using Backus-Naur Form (BNF) with some extensions for capturing the graphical part. The abstract grammar is obtained from the concrete grammars by removing irrelevant details such as separators and lexical rules.

The *well-formedness conditions* define which specifications, that are correct with respect to the grammar, are also correct with respect to context information, such as which names it is allowed to use at a given place, which kind of values it is allowed to assign to variables etc.

Furthermore, some language constructs appearing in the concrete grammars are replaced by other language elements in the abstract grammar using *transformation rules* to keep the set of semantic core concepts small. These transformations are described in the model paragraphs of Z.100. Formally, they are represented as rewrite rules.

The *dynamic semantics* is given only to syntactically correct SDL specifications that satisfy the well-formedness conditions. The dynamic semantics defines the set of computations associated with a specification.

1.1 Overview of the dynamic semantics

The *dynamic semantics* (clauses 2 and 3) consists of the following parts as illustrated by Figure F3-1:

- The *SDL Abstract Machine (SAM)* (see 2.1), which defines basic signal flow concepts of SDL such as signals, timers, exceptions, and gates, in terms of an ASM model (see 2.1.1). Furthermore, ASM agents are specialised to model agents in the context of SDL (see 2.1.2). Finally, several signal processing and behaviour primitives – conceptually, the abstract machine instructions of the SAM – are defined (see 2.1.3).
- The *compilation function* (see 2.2) mapping behaviour representations into the SDL Abstract Machine primitives. This function amounts to an abstract compiler taking the AST of the state machines as input and transforming it to abstract machine instructions.
- The *SAM Programs* (see 2.3) defining the set of computations. These programs consist of an initialisation phase and an execution phase. SAM programs have fixed parts that are the same for all SDL specifications, and variable parts that are generated from the abstract syntax representation of a given SDL specification.
- The *initialisation* (see 2.3.1) handling static structural properties of the specification. The initialisation recursively unfolds all the initial objects of the specification. In fact, the same process will be initiated at interpretation time also

when new SDL agents are created. From this point of view, the initialisation is merely the instantiation of the SDL system agent.

- The *execution* (see 2.3.2) is modelled by distinguishing two alternating phases, namely the selection and the firing of transitions.
- The *data semantics* (see clause 3), which is separated from the rest of the semantics by an interface (see 2.1.3). The use of an interface will allow to exchange the data model, if for some domain another data model is more appropriate than the SDL built-in model. Moreover, also the SDL built-in model can be changed this way without affecting the rest of the semantics.

As in the past, the new formal semantics is defined starting from the abstract syntax of SDL, which is documented in ITU-T Z.100. From this abstract syntax, a behaviour model that can be understood as abstract code generated from an SDL specification is derived. This approach differs substantially from the interpreter view taken in previous work, and will enable SDL-to-ASM compilers.

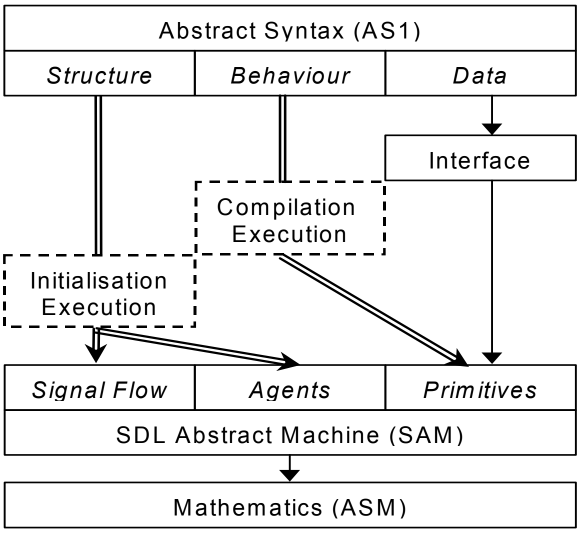


Figure F3-1/Z.100 – Overview of the dynamic semantics

The dynamic semantics associates, with each SDL specification, a particular distributed, real-time ASM. Intuitively, an ASM consists of a set of autonomous agents cooperatively performing concurrent machine runs. The behaviour of agents is determined by ASM programs, each consisting of a transition rule, which defines the set of possible computations (called “runs” in the context of ASM). Each agent has its own partial view on a global state, which is defined by a set of static and dynamic functions and domains. By having non-empty intersections of partial views, interaction among agents can be modelled. An introduction to the ASM model, and the notation used subsequently, is given in Annex F1.

1.2 Definitions from Annex F1

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

– the keywords **domain**, **static**, **initially**, **controlled**, **monitored**, **shared**, **constraint**, **let**, **where**, **choose**, **extend**;

– the domains *TIME*, *AGENT*, *PROGRAM*, *X*, *BOOLEAN*, *NAT*, *REAL*, *TOKEN*, *DefinitionAS1*, *DefinitionAS0*;

– the functions *take*, *currentTime*, *clock*, *program*, *Self*, *undefined*, *True*, *False*, *empty*, *head*, *tail*, *last*, *length*, *toSet*, *parentAS1*, *parentAS0*, *parentAS0ofKind*, *parentAS1ofKind*, *isAncestorAS0*, *isAncestorAS1*, *rootNodeAS1*, *rootNodeAS0*;

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

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

1.3 Definitions from Annex F2

Given an *Identifier*, the corresponding *DefinitionAS1* is retrieved using the function *idToNodeAS1*:

idToNodeAS1: *Identifier* → *DefinitionAS1*

Given a *DefinitionAS1*, the corresponding *Identifier* is retrieved using the function *nodeAS1ToId*:

nodeAS1ToId: *DefinitionAS1* → *Identifier*

2 Behaviour semantics

This clause defines the following parts of the dynamic semantics:

- the SDL Abstract Machine: see 2.1;
- the compilation function: see 2.2; and
- SAM programs: see 2.3.

An overview of the dynamic semantics is given in 1.1.

2.1 SDL Abstract Machine definition

The SDL Abstract Machine, or SAM, constitutes a generic behaviour model for SDL specifications. According to an abstract operational view, the possible computations of a given SDL 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: see 2.1.1;
- SDL agent related definitions: see 2.1.2;
- the interface to the data semantics: see 2.1.3; and
- behaviour primitives: see 2.1.4.

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

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* (see 2.1.1.5) and *exceptions* (see 2.1.1.6), which are modelled as special kinds of signals, are treated here.

2.1.1.1 Signals

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

$$PLAINSIGNAL =_{\mathrm{def}} \{sid \in Identifier\text{: } sid.idToNodeAS1 \in Signal\text{-}definition\} \cup \{\mathbf{NONE}\}$$

In an SDL specification, timers (see 2.1.1.5) and exceptions (see 2.1.1.6) are also considered as signals, they are contained in a common domain *SIGNAL*:

$$SIGNAL =_{\mathrm{def}} PLAINSIGNAL \cup EXCEPTION \cup TIMER$$

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 domain *SIGNALINST* contains all kinds of signal instances (*signals* for short).

shared domain *PLAINSIGNALINST*
initially *PLAINSIGNALINST* = ∅

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

Each element of *SIGNALINST* is uniquely related to an element of *SIGNAL*, as defined by the derived function *signalType*.

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

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

Furthermore, the functions *plainSignalSender* and *signalSender* can now be defined:

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

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

With each signal a (possibly empty) list of *signal values* is associated. Since 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 2.1.3):

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

signalValues(*si*:*SIGNALINST*): *VALUE** $=_{\text{def}}$
if *si* ∈ *PLAINSIGNALINST* **then** *si.plainSignalValues*
elseif *si* ∈ *TIMERINST* **then** *si*.**s-***VALUE***-seq**
elseif *si* ∈ *EXCEPTIONINST* **then** *si*.**s-***VALUE***-seq**
else *undefined*
endif

Additional functions on plain signals are *plainSignalSender*, *toArg*, and *viaArg* yielding the sender process, the destination and optional constraints on admissible communication paths. Furthermore, there is a (derived) function *signalSender* yielding the sender process of a signal. The precise meaning of these functions will be defined in subsequent clauses.

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*. Simultaneously arriving signals which match the same agent instance are appended, one at a time, in an order chosen non-deterministically. Signals are discarded whenever no matching receiver instance exists.

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

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

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

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

shared *toArg*: *PLAINSIGNALINST* → *TOARG*

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

2.1.1.2 Gates

Exchange of signals between SDL 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 2.1.2.4).

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

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

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

Global System Time

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

> **monitored** *now*: → *REAL*

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. That means it must be 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 must 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*

One can now represent the relation between signals and gates in a given SAM state by means of a dynamic function *schedule* defined on gates,

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

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

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

Waiting Signals

A signal instance *si* in *g*.*schedule* does not arrive "physically" at gate *g* before *now* ≥ *si*.*arrival*. Intuitively, that means that *s* remains "invisible" at *g* as long as it is in transit. Thus, in every given SAM state, 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* which 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*: *GATE*): *SIGNALINST** $=_{\text{def}}$ < *si* **in** *g*.*schedule* : (*now* ≥ *si*.*arrival*) >

See also Figure F3-2 below for an overview of the functions on schedules.

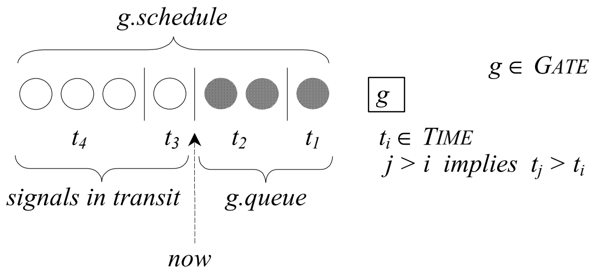


Figure F3-2/Z.100 – Signal instances at a gate

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 on schedules.

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

This defines the result of inserting some signal instance *si* with the intended arrival time *t* into a finite signal instance list *siSeq*, representing, e.g., 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* > ⁀ *delete*(*si*, *siSeq*.*tail*)
  endif

The rule 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*

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

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.

$$\begin{array}{l} \mathit{selectContinuousSignal}(\mathit{tSet}\text{: } \mathit{TRANSITION}\textbf{-set}, \mathit{nSet}\text{: } \mathit{NAT}\textbf{-set})\text{: } \mathit{TRANSITION} =_{\text{def}} \\ \quad \textbf{if } \forall t1 \in \mathit{tSet}\text{: } t1.\textbf{s-}\mathit{NAT} \in \mathit{nSet} \textbf{ then } \mathit{undefined} \\ \quad \textbf{else } \mathit{take}(\{t \in \mathit{tSet}\text{: } t.\textbf{s-}\mathit{NAT} \notin \mathit{nSet} \wedge \forall t1 \in \mathit{tSet}\text{: } (t1.\textbf{s-}\mathit{NAT} \notin \mathit{nSet} \Rightarrow t.\textbf{s-}\mathit{NAT} \leq t1.\textbf{s-}\mathit{NAT})\}) \\ \quad \textbf{endif} \end{array}$$

2.1.1.3 Channels

Channels, as declared in a given SDL 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 *LINKLINK*. The elements of *LINKLINK* are SAM agents, such that their behaviour is defined through LINK-PROGRAM.

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

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*.

controlled *from*: *LINK* → *GATE*
controlled *to*: *LINK* → *GATE*
controlled *noDelay*: *LINK* → **NODELAY**
controlled *with*: *LINK* → *SIGNAL*-**set**

Signal Delays

SDL considers channels as reliable and order-preserving communication links. A channel may however 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 must be 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.

static domain *DURATION*
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 values of *now* + *l.delay* increase monotonically (with respect to *now*).

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

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 needs 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 matching with the address information and the path constraints of *si*.

Abstractly, this decision can be expressed using a predicate *Applicable*, defined in 2.1.1.4. The domain *TOARG* is defined in 2.1.1.1.

FORWARDSIGNAL ≡
  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.nodeAS1.nodeAS1ToId*, *Self.nodeAS1.nodeAS1ToId*}
      endif
    endlet
  endif

2.1.1.4 Reachability

When signals are sent, it has to be determined whether there currently is an applicable communication path, i.e. 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* ∈ { *lSeq* ∈ *LINK**: (*lSeq* ≠ *empty* ∧ *Connected*(*lSeq.head*, *lSeq.tail*))}:
    (∀ *l* **in** *commPath*: *s* ∈ *l.with* ∧ *l.owner* ≠ *undefined* ∧
    commPath.head.from = *g* ∧
    if *l* ≠ *undefined* **then** *commPath.head* = *l* **endif** ∧
    ¬ ∃*l* ∈ *LINK*: (*l.from* = *commPath.last.to* ∧ *s* ∈ *l.with*) ∧   // the path is complete
    viaArg ⊆ *commPath.commPathIds* ∧
    if *toArg* ∈ *Agent-identifier* **then**
      commPath.last.to.myAgent.nodeAS1.nodeAS1ToId = *toArg* **endif** ∧
    if *toArg* ∈ *PID* ∧ *toArg* ≠ *null* **then**
      ∃*sa* ∈ *AGENT*: (*sa.owner* = *commPath.last.to.myAgent* ∧ *sa* = *toArg*.**s-***AGENT*) **endif**)

  where
    Connected(*l*:*LINK*,*lSeq*:*LINK**): *BOOLEAN* $=_{\text{def}}$
      if *lSeq* = *empty* **then** *True*
      elseif *l.to* = *lSeq.head.from* **then** *Connected*(*lSeq.head*, *lSeq.tail*)
      else *False*
      endif

    commPathIds(*lSeq*: *LINK**): *Identifier*-**set** $=_{\text{def}}$
      { *g.nodeAS1.nodeAS1ToId* | *g* ∈ *GATE*: ∃*l* **in** *lSeq*: (*g* = *l.from* ∨ *g* = *l.to*) } ∪
      { *l.nodeAS1.nodeAS1ToId* | *l* ∈ *LINK*: *l* **in** *lSeq* ∧ *l.nodeAS1* ≠ *undefined* }
  endwhere

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 $=_{\text{def}}$ {*tid* ∈ *Identifier*: *tid.idToNodeAS1* ∈ *Timer-definition*}
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*:

$Active(tmi{:}\mathit{TIMERINST})\text{: }\mathit{BOOLEAN} =_{\text{def}} tmi \in Self.inport.schedule$

Timer Operations

The macros below model the SDL actions *Set-node* and *Reset-node* on timers as executed by a corresponding SDL agent. A static function *duration* is used to represent default duration values as defined by an SDL specification under consideration.

static *duration*: $\mathit{TIMER} \rightarrow \mathit{DURATION}$

SETTIMER(*tm*:*TIMER*, *vSeq* :*VALUE**, *t*:*TIME*) ≡
    let *tmi* = **mk-***TIMERINST*(*Self.self*, *tm*, *vSeq*) **in**
        if *t* = *undefined* **then**
            Self.inport.schedule := *insert*(*tmi*, *now* + *tm.duration*, *delete*(*tmi*, *Self.inport.schedule*))
            si.arrival := *now* + *tm.duration*
        else
            Self.inport.schedule := *insert*(*tmi*, *t*, *delete*(*tmi*, *Self.inport.schedule*))
            si.arrival := *t*
        endif
    endlet

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

2.1.1.6 Exceptions

Like timers, exceptions are also identified with exception signals.

$\mathit{EXCEPTION} =_{\text{def}} \{eid \in Identifier\text{: } eid.idToNodeAS1 \in Exception\text{-}definition\}$
$\mathit{EXCEPTIONINST} =_{\text{def}} \mathit{PID} \times \mathit{EXCEPTION} \times \mathit{VALUE}^*$

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

Exception Handlers

Exception handlers are modelled by state nodes. Every agent keeps track of its active exception handlers with the function *activeHandler*. We also define a domain for the levels where exception handlers may reside. In order to model recursion an agent can store its active exception handlers at a state node.

EXCEPTIONSCOPE $=_{\text{def}}$ {*esEntireGraph*, *esCompositeState*, *esCompositeStateGraph*, *esCurrentState*, *esStimulusOrStart*, *esExceptionState*, *esHandleClause*, *esAction*}

exceptionScopeSeq $=_{\text{def}}$ < *esAction*, *esHandleClause*, *esExceptionState*, *esStimulusOrStart*, *esCurrentState*, *esCompositeStateGraph*, *esCompositeState*, *esEntireGraph* >

Moreover, to every agent a current exception is associated.

Exception Operations

The macros below model setting and resetting of handlers that occurs when entering or leaving their associated scope.

SETEXCEPTIONHANDLER(*handlerName*:*Exception-handler-name*, *scope*:*EXCEPTIONSCOPE*) ≡
    activeHandler(*Self*,*scope*) := *handlerName*

RESETEXCEPTIONHANDLER(*scope*:*EXCEPTIONSCOPE*) ≡
activeHandler(*Self*,*scope*) := *undefined*

Moreover, a macro is provided for raising an exception. The appropriate handler is selected in this case.

RAISEEXCEPTION(*eid*:*EXCEPTION*, *vSeq*:*VALUE**) ≡
Self.*currentExceptionInst* := **mk-***EXCEPTIONINST*(*Self*.*self*, *eid*, *vSeq*)
Self.*agentMode2* := *selectingTransition*
Self.*agentMode3* := *startSelection*

Information on exception handling can be found in 2.3.2.6.

2.1.2 SDL 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 2.1.1.3), SDL agent instances, and SDL 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 agent set instance of the domain *SDLAGENTSET*.

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

2.1.2.1 State machine

The structure of the agent's state machine is directly modelled, and built up during the agent initialisation. 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* = ∅

STATENODEKIND $=_{\text{def}}$ { *stateNode*, *statePartition*, *procedureNode*, *handlerNode*}
STATENODEREFINEMENTKIND $=_{\text{def}}$ { *compositeStateGraph*, *stateAggregationNode*}
STATEENTRYPOINT $=_{\text{def}}$ *State-entry-point-name* ∪ {**DEFAULT**}
STATEEXITPOINT $=_{\text{def}}$ *State-exit-point-name* ∪ {**DEFAULT**}
STATENODEWITHENTRYPOINT $=_{\text{def}}$ *STATENODE* × *STATEENTRYPOINT*
STATENODEWITHEXITPOINT $=_{\text{def}}$ *STATENODE* × *STATEEXITPOINT*
STATENODEWITHCONNECTOR $=_{\text{def}}$ *STATENODE* × *Connector-name*

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

controlled *stateNodeKind*: *STATENODE* → *STATENODEKIND*
controlled *stateNodeRefinement*: *STATENODE* → *STATENODEREFINEMENTKIND*
controlled *stateName*: *STATENODE* → *State-name*
controlled *stateId*: *STATENODE* → *STATEID*
controlled *inheritedStateNode*: *STATENODE* → *STATENODE*
controlled *parentStateNode*: *STATENODE* → *STATENODE*
controlled *stateTransitions*: *STATENODE* → *TRANSITION*-**set**
controlled *startTransitions*: *STATENODE* → *TRANSITION*-**set**
controlled *freeActions*: *STATENODE* → *FREEACTION*-**set**
controlled *statePartitionSet*: *STATENODE* → *STATENODE*-**set**

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**

This function defines, for each state node, the *current* sub states. If the state node is refined into a composite state graph, this will be at most one sub state. In case of a state aggregation node, this will be a subset of the state partition set.

collectCurrentSubStates(*snSet*: *STATENODE*-**set**): *STATENODE*-**set** $=_{\text{def}}$
    let *sn* = *take*(*snSet*) **in**
        if *sn* = *undefined* **then** ∅
        else {*sn*} ∪ *collectCurrentSubStates*(*snSet* \ {*sn*} ∪ *sn*.*currentSubStates*)
        endif
    endlet

This function collects, for a given state node set, all current sub states.

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

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

DirectlyInheritsFrom(*sn1*: *STATENODE*, *sn2*: *STATENODE*): *BOOLEAN* $=_{\text{def}}$
    if *sn2*.*parentStateNode* ∈ *sn1*.*parentStateNode*.*inheritedStateNodes* ∧
        sn1.*stateName* = *sn2*.*stateName* ∧
        (¬ ∃*sn3* ∈ *STATENODE*:
            sn3.*parentStateNode* ∈ *sn1*.*parentStateNode*.*inheritedStateNodes* ∧
            sn2.*parentStateNode* ∈ *sn3*.*parentStateNode*.*inheritedStateNodes* ∧
            sn3.*stateName* = *sn2*.*stateName*) **then** *True*
    else *False*
    endif

This predicate determines whether two state nodes are inherited by a single inheritance step.

InheritsFrom(*sn1*: *STATENODE*, *sn2*: *STATENODE*): *BOOLEAN* $=_{\text{def}}$
    sn2.*parentStateNode* ∈ *sn1*.*parentStateNode*.*inheritedStateNodes* ∧
    sn1.*stateName* = *sn2*.*stateName*

This predicate determines whether two state nodes are inherited.

DirectlyRefinedBy(sn1: *STATENODE*, *sn2*: *STATENODE*): *BOOLEAN* $=_{\text{def}}$
    sn2.*parentStateNode* = *sn1*

This predicate determines whether a state node is refined by another state node by a single refinement step.

InheritsFromOrRefinedBy(sn1: *STATENODE*, *sn2*: *STATENODE*): *BOOLEAN* $=_{\text{def}}$
    ∃*n* ∈ *NAT*: *InheritsFromOrRefinedByStep*(*sn1, sn2, n*)

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

InheritsFromOrRefinedByStep(sn1: *STATENODE*, *sn2*: *STATENODE*, *n*:∈ *NAT*): *BOOLEAN* $=_{\text{def}}$
    if *n* = 0 **then** *False*
    elseif *n* = 1 **then** *DirectlyRefinedBy*(*sn1, sn2*) ∨ *DirectlyInheritsFrom*(*sn1, sn2*)
    else ∃*sn3* ∈ *STATENODE*:
        (*InheritsFromOrRefinedByStep*(*sn1, sn3, 1*) ∧
        InheritsFromOrRefinedByStep(*sn3, sn2 ,n-1*))
    endif

This 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 s*n* = *undefined* **then** *undefined*
  elseif ∃*sn1* ∈ *snSet*: *DirectlyInheritsFrom*(*sn1, sn*) ∨ *sn* = *sn1*.*inheritedStateNode* **then**
   selectNextStateNode(*snSet* \ {*sn*})
  else *sn*
  endif
 endlet

This 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

This 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

This function defines, for a given state node, the set of inherited 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 function returns, for a given state node, the most specialised (rightmost) state node. It is 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*)})

This 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

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

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

This 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*: *STATENODE* → *STATENODE*

This 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*: $\mathit{STATEENTRYPOINT} \times \mathit{STATENODE} \rightarrow \mathit{STATEENTRYPOINT}$
controlled *exitConnection*: $\mathit{STATEEXITPOINT} \times \mathit{STATENODE} \rightarrow \mathit{STATEEXITPOINT}$

Finally, functions to model the entry and exit connections of state nodes are introduced.

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 will be explained in 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
    selectException,                // 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 will be relevant. In cases no conflict arises, agent modes may be applied on more than one level of this hierarchy.

2.1.2.3 Agent Control Block

The state information of an SDL agent instance is collected in an *agent control block*. The agent control block is partially initialised when an SDL agent (set) instance is created, and completed/modified during its initialisation 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 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 *nodeAS1*: *AGENT* ∪ *GATE* ∪ *STATENODE* → *DefinitionAS1*

A unary function *nodeAS1* defined on agents, gates and state nodes identifies the corresponding AST definition. This definition is needed, e.g., during the initialisation phase and also during dynamic creation of agents.

IsAgentSet(*ag*: *AGENT*): *BOOLEAN* $=_{\text{def}}$ *ag*.*nodeAS1* ∈ *Agent-definition*

To distinguish SDL agent sets from other agents, the predicate *IsAgentSet* is defined.

controlled *self*: *SDLAGENT* → *PID*
controlled *sender*: *SDLAGENT* → *PID*
controlled *parent*: *SDLAGENT* → *PID*
controlled *offspring*: *SDLAGENT* → *PID*

The above functions model the corresponding functions as introduced in ITU-T Z.100.

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 will be 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 agent are normally associated with this agent. However, in case of nested process agents, they are associated with the outermost process agent. The function *stateAgent* yields, for a given SDL agent, the SDL agent to which the variable values are associated.

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

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

controlled *isActive*: *SDLAGENT* → *SDLAGENT*

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

monitored *Spontaneous*: *AGENT* → *BOOLEAN*

The SDL concept of *spontaneous transition* is abstractly modelled by means of a monitored predicate *Spontaneous* associated with a particular SDL 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 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 *currentExceptionInst*: *SDLAGENT* → *EXCEPTIONINST*

The current exception instance is used within exception handling.

controlled *scopeName*: *SDLAGENT* × *STATEID* → *Connector-name*
controlled *scopeContinueLabel*: *SDLAGENT* × *STATEID* → *CONTINUELABEL*
controlled *scopeStepLabel*: *SDLAGENT* × *STATEID* → *STEPLABEL*

These functions are used to execute compound nodes.

InitStateMachine/InitProcedureGraph Control Block

When the state machine of an agent is initialised, a hierarchical inheritance state graph is created. Since this in general 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 will be used during the initialisation 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 *ehNodesToBeCreated*: *SDLAGENT* → *Exception-handler-node*-**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 state nodes, state partitions and exception handler nodes to be created. Also, the refinement and specialisation 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 to arrive while the selection is active; however, these signals will not be 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* → *TRANSITION*-**set**
controlled *transitionChecked*: *SDLAGENT* → *TRANSITION*
controlled *signalChecked*: *SDLAGENT* → *SIGNALINST*

controlled *SignalSaved*: *SDLAGENT* → *BOOLEAN*
controlled *continuousPriorities*: *SDLAGENT* → *NAT*-**set**
controlled *exceptionScopesToBeChecked*: *SDLAGENT* → *EXCEPTIONSCOPE**
controlled *ehParentStateNodeChecked*: *SDLAGENT* → *STATENODE*
controlled *exceptionHandlerChecked*: *SDLAGENT* → *Exception-handler-name*

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, e.g., 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 sub states. Every sub state is given an identification at the time of its creation, e.g. 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. Since a modification of an object or the assignment of the value may effect non-local variables, modifications returned by the data type part always modify the state of the outermost process agent.

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

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

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

This 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*: $\mathit{SDLAGENT} \cup \mathit{STATENODE} \rightarrow \mathit{STATENODE}$

The current parent state node is needed to define the correct ownership between state nodes, and to identify states to be left and to be entered.

controlled *previousStateNode*: $\mathit{SDLAGENT} \cup \mathit{STATENODE} \rightarrow \mathit{STATENODE}$

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

controlled *currentProcedureStateNode*: $\mathit{SDLAGENT} \cup \mathit{STATENODE} \rightarrow \mathit{STATENODE}$

Refers to the current procedure state node.

controlled *activeHandler*: $(\mathit{AGENT} \cup \mathit{STATENODE}) \times \mathit{EXCEPTIONSCOPE} \rightarrow \mathit{Exception\text{-}handler\text{-}name}$

2.1.2.4 Agent Connections

SDL agents are organised in agent sets. All members of an agent set share the same sets of input gates and output gates as defined for the agent set.

$$
\begin{aligned}
&\mathit{GateUnconnected}(g{:}\mathit{GATE}){:}\mathit{BOOLEAN} =_{\mathrm{def}} \\
&\quad \forall cd \in g.\mathit{myAgent}.\mathit{nodeAS1}.\mathbf{s}\text{-}\mathit{Channel\text{-}definition}\text{-}\mathbf{set}\text{: } \forall cp \in cd.\mathbf{s}\text{-}\mathit{Channel\text{-}path}\text{-}\mathbf{set}\text{:} \\
&\qquad (g.\mathit{nodeAS1} \neq cp.\mathbf{s}\text{-}\mathit{Originating\text{-}gate}.\mathit{idToNodeAS1} \wedge \\
&\qquad\ g.\mathit{nodeAS1} \neq cp.\mathbf{s}\text{-}\mathit{Destination\text{-}gate}.\mathit{idToNodeAS1})
\end{aligned}
$$

A gate is called *unconnected* if it is not linked to an inner gate by a channel path:

$$
\begin{aligned}
&\mathit{ingates}(a\text{: }\mathit{AGENT})\text{: }\mathit{GATE}\text{-}\mathbf{set} =_{\mathrm{def}} \\
&\quad \mathbf{if}\ a.\mathit{IsAgentSet}\ \mathbf{then} \\
&\qquad \{\, g \in \mathit{GATE}\text{: } g.\mathit{myAgent} = a \wedge g.\mathit{direction} = \mathit{inDir} \wedge g.\mathit{GateUnconnected}\} \\
&\quad \mathbf{else} \\
&\qquad a.\mathit{owner}.\mathit{ingates} \\
&\quad \mathbf{endif}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{outgates}(a{:}\mathit{AGENT})\text{: }\mathit{GATE}\text{-}\mathbf{set} =_{\mathrm{def}} \\
&\quad \mathbf{if}\ a.\mathit{IsAgentSet}\ \mathbf{then} \\
&\qquad \{\, g \in \mathit{GATE}\text{: } g.\mathit{myAgent} = a \wedge g.\mathit{direction} = \mathit{outDir} \wedge g.\mathit{GateUnconnected}\} \\
&\quad \mathbf{else} \\
&\qquad a.\mathit{owner}.\mathit{outgates} \\
&\quad \mathbf{endif}
\end{aligned}
$$

Two derived function *ingates* and *outgates* collecting all input gates and all output gates of an agent are defined. 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.

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 undefined. For instance, in case of an input transition, there is no firing transition and no priority.

$$\mathit{TRANSITION} =_{\mathrm{def}} \mathit{SIGNAL} \times \mathit{LABEL} \times \mathit{NAT} \times \mathit{LABEL} \times \mathit{STATEEXITPOINT}$$

$$\mathit{STARTTRANSITION} =_{\mathrm{def}} \mathit{LABEL} \times \mathit{STATEENTRYPOINT}$$

$$\mathit{FREEACTION} =_{\mathrm{def}} \mathit{Connector\text{-}name} \times \mathit{LABEL}$$

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

> *priorityInputTransitions*(*tSet*:*TRANSITION*-**set**): *TRANSITION*-**set** $=_{\text{def}}$
> { $t \in$ *tSet*: *t*.**s**-*SIGNAL* $\neq$ *undefined* $\wedge$ *t*.**s**-*LABEL* = *undefined* $\wedge$ *t*.**s**-*NAT* $\neq$ *undefined* }

> *inputTransitions*(*tSet*:*TRANSITION*-**set**): *TRANSITION*-**set** $=_{\text{def}}$
> { $t \in$ *tSet*: *t*.**s**-*SIGNAL* $\neq$ *undefined* $\wedge$ *t*.**s**-*NAT* = *undefined* }

> *continuousSignalTransitions*(*tSet*:*TRANSITION*-**set**): *TRANSITION*-**set** $=_{\text{def}}$
> { $t \in$ *tSet*: *t*.**s**-*SIGNAL* = *undefined* $\wedge$ *t*.**s**-*LABEL* $\neq$ *undefined* $\wedge$ *t*.**s**-*NAT* $\neq$ *undefined* }

> *spontaneousTransitions*(*tSet*:*TRANSITION*-**set**): *TRANSITION*-**set** $=_{\text{def}}$
> { $t \in$ *tSet*: *t*.**s**-*SIGNAL* = **NONE**}

> *exitTransitions*(*tSet*:*TRANSITION*-**set**): *TRANSITION*-**set** $=_{\text{def}}$
> { $t \in$ *tSet*: *t*.**s**-*STATEEXITPOINT* $\neq$ *undefined* }

2.1.3 Interface to the Data Type Part

The semantics of the data type part of SDL will be handled separately from the concurrency related aspects of the language. To make this splitting possible, an interface for the semantics definition has to be defined.

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.

> **derived domain** *STATE*

This function will be 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 this function via the functions *initAgentState* and *initProcedureState*. In order to handle recursion, a state might contain sub states. Every sub state is given an identification at the time of its creation, e.g. 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. Since a modification of an object or the assignment of the value may effect non-local variables, modifications returned by the data type part always modify the outermost process agent.

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* $\cup$ *Variable-definition*

> *PARAMETERKIND* $=_{\text{def}}$ {*in*, *out*, *inout*}

> *initAgentState*: *STATE* $\times$ *STATEID* $\times$ *STATEID* $\times$ *DECLARATION*-**set** $\rightarrow$ *STATE*

> *initProcedureState*: *STATE* $\times$ *STATEID* $\times$ *STATEID* $\times$ *DECLARATION*-**set** $\times$ (*VALUE* $\cup$ *Identifier*)* $\rightarrow$ *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 will also be a domain for values, called *VALUE*. Moreover, there is a domain for variable names that is used by both parts of the semantics. The domain *VARIABLE* denotes variables independent from their assigned values.

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

derived domain *VARIABLE*

Some operations invoked in the data part may terminate with an exception. In that case, they do not return a new state, but an exception.

STATEOREXCEPTION = *STATE* ∪ *EXCEPTION*
VALUEOREXCEPTION = *VALUE* ∪ *EXCEPTION*

The data type part has to provide function how assignments are performed, namely

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

assignClones: *Variable-identifier* × *VALUE* → *BOOLEAN*

assignCopies: *Variable-identifier* × *VALUE* → *BOOLEAN*

objectsAssign: *OBJECTVALUE** × *Variable-identifier* × *VALUE* → *OBJECTVALUE**

The function *eval* retrieves the value associated with a variable for a given state and state id. *assign* associates a new value with a given variable. Since an assignment may involve inocation of an operator copy or assign, there are two predicates *assignClones* and *assignCopies* that determine whether these operators must be invoked to perform the assignment. In some cases, an assignment will modify the bindings of objects to values. In these cases, *objectsAssign* returns the new values for the objects. There is a rule macro using this function, which is doing the real assignment.

```
ASSIGN(variableName, value, state, id) ≡
    if assignClones(variableName, value) then
        tmpval := CALLCLONE(variableName,value)
        state := assign(variableName, tmpval, state, id)
        SETOBJECTS(Self, objectsAssign(getObjects(Self, variablename, value)))
    else
        if assignCopies(variablename, value) then
            tmpval := CALLCOPY(variablename, value)
            state := assign(variableName, tmpval, state, id)
            SETOBJECTS(Self, objectsAssign(getObjects(Self, variablename, value)))
        else
            state := assign(variableName, value, state, id)
            SETOBJECTS(Self, objectsAssign(getObjects(Self, variablename, value)))
```

Assignments are the only way to change the state. The object values may change as a side effect of an operator computation, as well. The function getObjects returns the object values for a given agent; the macro SETOBJECTS modifies the object values.

In order to get the current value of a variable, the data part provides a function 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 for procedures (definitions) and a dynamic domain for procedure instances.

domain *PROCEDUREBODY* $=_{\text{def}}$ *BEHAVIOUR*

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

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

static *dispatch*: *PROCEDURE* × *VALUE** → *Identifier*.

Finally, there are two functions to model the predefined functions that do not have a procedure body. There is one function to check if the procedure is functional (predefined), and one function computing the result in this case. The function *computeSideEffects* returns any changes that need to be made to the object values as a result of the computation.

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

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

computeSideEffects: *PROCEDURE* × *VALUE** × *STATE* × *STATEID* → *OBJECTVALUE**

Moreover, the following domains and functions referring to the predefined data are used.

derived domain *SDLBOOLEAN*

derived domain *SDLINTEGER*

derived *semvalue*: *SDLBOOLEAN* → *BOOLEAN*

2.1.3.2 Functions Used by the Data Type Part

Values of object types have a unique identification, which is represented by a domain *OBJECTIDENTIFIER*.

domain *OBJECTIDENTIFIER*
initially *OBJECTIDENTIFIER* = { *null* }

The function *mkObjectId* will generate a new *OBJECTIDENTIFIER* on each evaluation:

mkObjectId: → *OBJECTIDENTIFIER*

This is the complete dynamic semantics interface. 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 will be no semantic problems in the concurrency part, as all state changes are automatically synchronised 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.

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 sections* for further explanations complement the description of behaviour primitives.

static *behaviour*: → *BEHAVIOUR*

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 $=_{\text{def}}$ *STARTLABEL* × *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.

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 sections. For example, there exists a domain *VAR* which contains actions for the evaluation of variables.

$ACTION =_{def} VAR \cup OPERATIONAPPLICATION \cup CALL \cup RETURN \cup TASK \cup ASSIGNPARAMETERS \cup EQUALITY \cup DECISION \cup OUTPUT \cup CREATE \cup SET \cup RESET \cup TIMERACTIVE \cup SETHANDLER \cup RAISE \cup STOP \cup SYSTEMVALUE \cup ANYVALUE \cup SETRANGECHECKVALUE \cup SCOPE \cup SKIP \cup BREAK \cup CONTINUE \cup ENTERSTATENODE \cup LEAVESTATENODE$

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.

$CONTINUELABEL =_{def} LABEL$

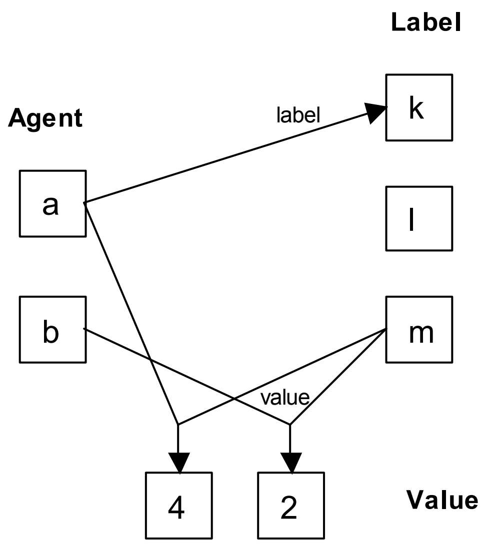
$VALUELABEL =_{def} LABEL$

Functions

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

controlled *value*: $VALUELABEL \times SDLAGENT \rightarrow VALUE$

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



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 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* ∈ *SETHANDLER* **then** EVALSETHANDLER(*a*)
    elseif *a* ∈ *RAISE* **then** EVALRAISE(*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

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 cartesian product of the domain *Variable-identifier* of variable names and domain *CONTINUELABEL* of labels.

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

Behaviour

The value of a variable in the current state of the executing agent is determined by function *eval* and 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*) ≡
  value(*Self.currentLabel*, *Self*) := *eval*(*a*.**s-***Variable-identifier*, *Self.stateAgent.state*)
  Self.currentLabel := *a*.**s-***CONTINUELABEL*

Reference clauses

For definition of function *value*, refer to 2.1.4.1. Definition of function *eval* can be found in 2.1.3.1. Function *currentLabel* is defined in 2.1.2.3.

2.1.4.3 Primitive Operation Application

Explanation

The operation application 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 will be executed by function *compute*. A non-functional operation which is handled with function *dispatch* which determines depending on the current values the correct procedure identifier.

Representation

OPERATIONAPPLICATION $=_{\text{def}}$ *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* = *idToNodeAS1*(*dispatch*(*a*.**s-***PROCEDURE*, *values*(*a*.**s-***VALUELABEL***-seq**, *Self*)) **in**
      CREATEPROCEDURE(*pd*, *a*.**s-***CONTINUELABEL*)
    endlet
  endif

Reference clauses

For definition of function *value*, refer to 2.1.4.1. Definition of predicate *functional* and function *compute* can be found in 2.1.3.1.

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 will be 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.

$$CALL =_{def} Procedure\text{-}identifier \times (VALUELABEL \cup Variable\text{-}identifier)^* \times CONTINUELABEL$$

Behaviour

EVALCALL(*a*:*CALL*) ≡
    CREATEPROCEDURE(*a*.**s-***Procedure-identifier*.*idToNodeAS1*,*a*.**s-***CONTINUELABEL*)

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

Furthermore, the current exception information is saved to the new state node. The current exception information has is initialised

SAVEEXCEPTIONSCOPES(*sn*:*STATENODE*) ≡
    do forall *scope*: *scope* ∈ *EXCEPTIONSCOPE*
        activeHandler(*sn*, *scope*) := *activeHandler*(*Self*, *scope*)
        RESETEXCEPTIONHANDLER(*scope*)
    enddo

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* := *Self*.*cl*
    sn.*currentParentStateNode* := *Self*.*currentParentStateNode*
    sn.*previousStateNode* := *Self*.*previousStateNode*

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

Reference clauses

For definition of macro CREATEPROCEDURE**,** refer to 2.3.1.4. Information associated with exceptions can be found in 2.1.1.6. Information on procedure control blocks are given in 2.1.2.3.

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 ∈ VALUELABEL then
            EVALEXITPROCEDURE(a.s-VALUELABEL)
        else
            EVALEXITCOMPOSITESTATE(a.s-STATEEXITPOINT)
        endif

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

RESTOREEXCEPTIONSCOPES(sn:STATENODE) ≡
        do forall scope : scope ∈ EXCEPTIONSCOPE
            activeHandler(Self, scope) := activeHandler(sn, scope)
        enddo
```

Reference clauses

Information associated with exceptions can be found in 2.1.1.6. Information on procedure control blocks is given in 2.1.2.3.

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 realised 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 clauses

Definition of macro ASSIGN can be found in 2.1.3.1

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.

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

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 clauses

Definition of macro ASSIGN can be found in 2.1.3.1

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

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

Behaviour

EVALEQUALITY (*a*:*EQUALITY*) ≡
    if *value*(*a*.**s-***VALUELABEL*, *Self*) = *value*(*a*.**s2-***VALUELABEL*, *Self*) **then**
        value(*a*.**s-***CONTINUELABEL*, *Self*) := *True*
    else
        value(*a*.**s-***CONTINUELABEL*, *Self*) := *False*
    endif
    Self.currentLabel := *a*.**s-***CONTINUELABEL*

Reference clauses

No references.

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.

$DECISION =_{\text{def}} VALUELABEL \times ANSWER\textbf{-set} \times CONTINUELABEL$

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

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(*a*.**s-***VALUELABEL*, *Self*) = *value*(*an*.**s-***VALUELABEL*, *Self*)
      Self.*currentLabel* := *an*.**s-***CONTINUELABEL*
    endchoose
  elseif *an*.**s-***CONTINUELABEL* ≠ *undefined* **then**
    Self.*currentLabel* := *d*.**s-***CONTINUELABEL*
  else RAISEEXCEPTION(*OutOfRange*, *empty*)
  endif

Reference clauses

For definition of function *value*, refer to 2.1.4.1.

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.

$OUTPUT =_{\text{def}} SIGNAL \times VALUELABEL^* \times VALUELABEL \times VIAARG \times CONTINUELABEL$

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*), *value*(*a*.**s-***VALUELABEL*, *Self*),
      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 which is connected to a path being *compatible* with *TOARG*, *VIAARG*. In the rule below, this choice is stated in abstract terms using the predicate *Applicable* (see 2.1.1.4). If the constraints can not be met, the signal instance is discarded.

SIGNALOUTPUT(*s*:*SIGNAL*, *vSeq*:*VALUE**, *toArg*:*TOARG*, *viaArg*:*VIAARG*) ≡
    if *toArg* ∈ *PID* ∧ *s* ∉ *toArg*.**s-***Interface-definition* **then**
        RAISEEXCEPTION(*InvalidReference*, *empty*)
    else
        choose *g*: *g* ∈ *Self.outgates* ∧ *Applicable*(*s*, *toArg*, *viaArg*, *g*, *undefined*)
            extend *PLAINSIGNALINST* **with** *si*
                si.plainSignalType:= *s*
                si.plainSignalValues := *vSeq*
                si.toArg := *toArg*
                si.viaArg := *viaArg*
                si.plainSignalSender := *Self.self*
                INSERT(*si*, *now*, *g*)
            endextend
        endchoose
    endif

Reference clauses

For definition of function *value*, refer to 2.1.4.1. Definitions of functions associated with signals can be found in 2.1.1.1.

2.1.4.11 Primitive Create

Explanation

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

Representation

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

$CREATE =_{def} Agent\text{-}identifier \times VALUELABEL^* \times CONTINUELABEL$

Behaviour

EVALCREATE(*a*:*CREATE*) ≡
        let *sas* = *take*({*sas* ∈ *SDLAGENTSET*: *sas.nodeAS1* = *a*.**s-***Agent-identifier.idToNodeAS1* ∧
            sas ∈ *Self.getAgentSetsWithinScope*}) **in**
            if *sas.nodeAS1*.**s-***Number-of-instances*.**s-***Maximum-number* ≠ *undefined* **then**
                let *n* = |{ *sa* ∈ *SDLAGENT*: *sa.owner* = *sas* }| **in**
                    if *n* < *sas.nodeAS1*.**s-***Number-of-instances*.**s-***Maximum-number* **then**
                        CREATEAGENT(*sas*, *Self.self*, *sas.nodeAS1*.**s-***Agent-type-definition*)
                    else
                        Self.offspring := *null*
                    endif
                endlet
            else
                CREATEAGENT(*sas*, *Self.self*, *sas.nodeAS1*.**s-***Agent-type-definition*)
            endif

endlet
Self.currentLabel := *a*.**s-**CONTINUELABEL

where
getAgentSetsWithinScope(*sa*: SDLAGENT): SDLAGENTSET-**set** $=_{def}$
if *sa.owner.nodeAS1*.**s-***Agent-kind* ∈ {**SYSTEM**, **BLOCK**}
then {*sa.owner*}
else {*sa.owner*} ∪ *sa.owner.owner.getAgentSetsWithinScope*
endif
endwhere

Reference clauses

For the definition of the macro CREATEAGENT, see 2.3.1.3.

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

Domains

TIMELABEL $=_{def}$ *VALUELABEL*

Behaviour

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

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

Reference clauses

Definition of macro SETTIMER can be found in 2.1.1.5.

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 $=_{def}$ *TIMER* × *VALUELABEL** × *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 clauses

Definition of macro RESETTIMER can be found in 2.1.1.5.

2.1.4.14 Primitive TimerActive

Explanation

The timer active 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 timer active is defined as tuple consisting of a timer, a sequence of value labels, and a continue label.

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

Behaviour

Macro EVALTIMERACTIVE specifies the evaluation of a timer active expression.

EVALTIMERACTIVE(*t*:*TIMERACTIVE*) ≡
let *tmi* = **mk-***TIMERINST*(*Self*.*self*, *t*.**s-***TIMER*, *values*(*t*.**s-***VALUELABEL***-seq**, *Self*)) **in**
value(*Self*.*currentLabel*, *Self*) := *semvalue*(*Active*(*tmi*))
Self.*currentLabel* := *t*.**s-***CONTINUELABEL*
endlet

Reference clauses

Definition of function *Active* can be found in 2.1.1.5.

2.1.4.15 Primitive SetHandler

Explanation

The setHandler primitive is used for expressing the setting of an exception handler. An action of type *SETHANDLER* consists of an exception handler name, an exception scope, and a continue label. The primitive defines the setting of an exception handler with macro SETEXCEPTIONHANDLER.

Representation

An action of type *SETHANDLER* consists of a handler label, an exception scope, and a continue label. If *Exception-handler-name* is *undefined*, then the exception handler is reset.

$SETHANDLER =_{def} Exception\text{-}handler\text{-}name \times EXCEPTIONSCOPE \times CONTINUELABEL$

Behaviour

Macro EVALSETHANDLER defines the setting of exception handler with macro SETEXCEPTIONHANDLER.

EVALSETHANDLER(*a*:*SETHANDLER*) ≡
SETEXCEPTIONHANDLER(*a*.**s-***Exception-handler-name*, *a*.**s-***EXCEPTIONSCOPE*)
Self.*currentLabel* := *a*.**s-***CONTINUELABEL*

Reference clauses

Information associated with exceptions can be found in 2.1.1.6.

2.1.4.16 Primitive Raise

Explanation

The raise primitive is used for expressing the raising of exceptions. An action of type *RAISE* is defined as tuple consisting of an exception identifier and a sequence of value labels. Macro EVALRAISE defines the raising of an exception with macro RAISEEXCEPTION.

Representation

An action of type *RAISE* is defined as tuple consisting of an exception and a value label.

$RAISE =_{\text{def}} Exception\text{-}identifier \times VALUELABEL^*$

Behaviour

Macro EVALRAISE defines the raising of an exception with macro RAISEEXCEPTION.

EVALRAISE(*a*:*RAISE*) ≡
RAISEEXCEPTION(*a*.**s-***Exception-identifier*, *values*(*a*.**s-***VALUELABEL***-seq**, *Self*))

Reference clauses

Information associated with exceptions can be found in 2.1.1.6.

2.1.4.17 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

$STOP =_{\text{def}} \{stop\}$

Behaviour

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

EVALSTOP(*a*:*STOP*) ≡
Self.*agentMode2* := *stopping*

Reference clauses

See 2.3.2.19.

2.1.4.18 Primitive SystemValue

Explanation

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

Representation

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

Behaviour

EVALSYSTEMVALUE(*a*: *SYSTEMVALUE*) ≡
value(*Self*.*currentLabel*, *Self*) :=
case *a*.**s-***VALUEKIND* **of**
| *kNow*: *now.semvalue*
| *kSelf*: *Self.self.semvalue*
| *kParent*: *Self.parent.semvalue*
| *kOffspring*: *Self.offspring.semvalue*
| *kSender*: *Self.sender.semvalue*
otherwise *undefined*
endcase
Self.*currentLabel* := *a*.**s-***CONTINUELABEL*

2.1.4.19 Primitive AnyValue

Explanation

The AnyValue Primitive computes the any expression.

Representation

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

Behaviour

EVALANYVALUE(*a*: *ANYVALUE*) ≡
value(*Self*.*currentLabel*, *Self*) := *compute*(**ANY**, *a*.**s-***Sort-identifier*)
Self.*currentLabel* := *a*.**s-***CONTINUELABEL*

2.1.4.20 Primitive SetRangeCheckLabel

Explanation

The *SETRANGECHECKVALUE* Primitive is used to set the value to be used in a range check.

Representation

SETRANGECHECKVALUE $=_{\text{def}}$ *VALUELABEL* × *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*

2.1.4.21 Primitive Scope

Explanation

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

Representation

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

Behaviour

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

Reference clauses

See also 2.3.1.8.

2.1.4.22 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* ∈ *Connector-name* **then**
    Self.*stateNodeChecked* := *Self*.*parentStateNode*
    Self.*currentConnector* := *a*.**s-***Connector-name*
    Self.*agentMode2* := *selectingTransition*
    Self.*agentMode3* := *startSelection*
  else
    Self.*currentLabel* := *a*.**s-***CONTINUELABEL*
  endif

Reference clauses

See 2.3.2.9.

2.1.4.23 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*)

2.1.4.24 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

2.1.4.25 Primitive EnterStateNode

Explanation

State nodes are entered when an SDL 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**, **DASH** }) × *STATEENTRYPOINT* × *VALUELABEL**

Behaviour

EVALENTERSTATENODE(*a*:*ENTERSTATENODE*) ≡
    choose *sn*: *sn* ∈ *STATENODE* ∧ *sn*.*stateName* = *a*.**s-***State-name* ∧
      sn.*stateNodeKind* = *stateNode* ∧ *sn*.*parentStateNode* = *Self*.*currentParentStateNode*
      Self.*stateNodesToBeEntered* :=
        {**mk-***STATENODEWITHENTRYPOINT*(*sn*, *a*.**s-***STATEENTRYPOINT*)}
    endchoose
    Self.*agentMode3* := *enteringStateNode*
    Self.*agentMode4* := *startPhase*
    Self.*currentLabel* := *undefined*
    Self.*continueLabel* := *undefined*

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 clauses

See also 2.3.2.16.

2.1.4.26 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 clauses

See also 2.3.2.17 for information on how state nodes are left.

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 contents 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.

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*): *TRANSITION*-**set** $=_{def}$
{ **mk-***TRANSITION*(*i*.**s-***Signal-identifier*, *i*.**s-***Provided-expression*.*startLabel*,
if *i*.**s-PRIORITY** = *undefined* **then** *undefined* **else** 1 **endif**,
i.**s-***Transition*.*startLabel*, *undefined*)
| *i* ∈ *s*.**s-***Input-node*-**set** } ∪
{ **mk-***TRANSITION*(**NONE**, *sp*.**s-***Provided-expression*.*startLabel*,
undefined, *sp*.**s-***Transition*.*startLabel*, *undefined*)
| *sp* ∈ *s*.**s-***Spontaneous-transition*-**set** } ∪
{ **mk-***TRANSITION*(*undefined*, *c*.**s-***Continuous-expression*.*startLabel*,
c.**s-***Priority-name*, *c*.**s-***Transition*.*startLabel*, *undefined*)
| *c* ∈ *s*.**s-***Continuous-signal*-**set** } ∪
{ **mk-***TRANSITION*(*undefined*, *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** }

getHandleNodes(*eh*: *Exception-handler-node*): *TRANSITION*-**set** $=_{def}$
{ **mk-***TRANSITION*(*h*.**s-***Exception-identifier*, *undefined*, *undefined*,
h.**s-***Transition*.*startLabel*, *undefined*)
| *h* ∈ *eh*.**s-***Handle-node*-**set** }

getStartTransitions(*s*: (*State-start-node* ∪ *Named-start-node* ∪ *Procedure-start-node*)-**set**):
STARTTRANSITION-**set** $=_{def}$
{ **mk-***STARTTRANSITION*(*sn*.**s-***Transition*.*startLabel*, *c*.**s-***State-entry-point-name*)
| *sn* ∈ *s* ∧ *sn* ∈ *State-start-node* } ∪
{ **mk-***STARTTRANSITION*(*sn*.**s-***Transition*.*startLabel*, *c*.**s-***State-entry-point-name*)
| *sn* ∈ *s* ∧ *sn* ∈ *Named-start-node* } ∪
{ **mk-***STARTTRANSITION*(*sn*.**s-***Transition*.*startLabel*, *undefined*)
| *sn* ∈ *s* ∧ *sn* ∈ *Procedure-start-node* }

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

Here we present the function that compiles an SDL 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 agent to transition rules in the resulting SAM model. The effect of state transitions of SDL 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 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 formalise 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 formalised 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 2.2.4.

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 is 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, handlers) =>
        compile(start) ∪
        ⋃{ compile(s) | s ∈ states } ∪
        ⋃{ compile(f) | f ∈ freeActions } ∪
        ⋃{ compile(h) | h ∈ handlers }

| Exception-handler-node( *, *, handleNodes, elseNode) =>
        ⋃{ compile(h) | h ∈ handleNodes } ∪
        compile(elseNode)

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

| Else-handle-node(onexcep, transition) =>
        if onexcep = undefined then ∅ else compileExpr(onexcep, transition.startLabel) endif ∪
        compile(transition)

| Procedure-graph( onexcep, start, states, freeActions, handlers) =>
        if onexcep = undefined then ∅ else compileExpr(onexcep, start.startLabel) endif ∪
        compile(start) ∪
        ⋃{ compile(s) | s ∈ states } ∪
        ⋃{ compile(f) | f ∈ freeActions } ∪
        ⋃{ compile(h) | h ∈ handlers }

| State-start-node(onexcep, *, transition) =>
        if onexcep = undefined then ∅ else compileExpr(onexcep, transition.startLabel) endif ∪
        compile(transition)

| Procedure-start-node(onexcep, transition) =>
        if onexcep = undefined then ∅ else compileExpr(onexcep, transition.startLabel) endif ∪
        compile(transition)

| Named-start-node(*, onexcept, trans) =>
        if onexcep = undefined then ∅ else compileExpr(onexcep, transition.startLabel) endif ∪
        compile(transition)

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

| Save-signalset(signalset) =>
        if signalset = undefined then ∅ else ⋃{ compile(s) | s ∈ signalset }  endif

| Input-node(*, *, vars, provided, onexcep, transition) =>
        compileExpr(provided, undefined) ∪
        if onexcep = undefined then ∅ else compileExpr(onexcep, transition.startLabel) endif ∪
        { mk-PRIMITIVE(uniqueLabel(h,idx),
            if vars[idx] ≠ undefined then
                mk-ASSIGNPARAMETERS(varx[idx], idx,
                    if idx=vars.length then transition.startLabel else uniqueLabel(h,idx) endif)
            else mk-SKIP(if idx=vars.length then transition.startLabel else uniqueLabel(h,idx) endif)
            endif)
```

| *idx* ∈ *toSet*(1..*vars*.*length*) } ∪
compile(*transition*)

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

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

| *Connect-node*(*, *onexcep*, *trans*) =>
if *onexcep* = *undefined* **then** ∅ **else** *compileExpr*(*onexcep*, *transition*.*startLabel*) **endif** ∪
compile(*transition*)

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

| *t*=*Transition*(*nodes*, *endnode*) =>
{**mk-***PRIMITIVE*(*uniqueLabel*(*a*,1),
mk-*LEAVESTATENODE*(*t*.*parentAS1*.*parentAS1*.**s-***State-node*, *nodes*.*startLabel*)) } ∪
compileNodes ∪
compile(*endnode*)
where
compileNodes: *BEHAVIOUR* $=_{\mathrm{def}}$
if *nodes* = *empty* **then** ∅
else *compileExpr*(*nodes*.*last*, *endnode*. *startLabel*) ∪
⋃{ *compileExpr*(*nodes*[*i*], *nodes*[*i*+1]. *startLabel*) | *i* ∈ 1..*nodes*.*length*-1 }
endif
endwhere

2.2.2 Terminators

| *Terminator*(*terminator*, *onexcep*) =>
if *onexcep* = *undefined* **then** ∅ **else** *compileExpr*(*onexcep*, *terminator*.*startLabel*) **endif** ∪
compileExpr(*terminator*, *next*)

| *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)) ∪
⋃{ *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*(*undefined*) =>
{**mk-***PRIMITIVE*(*uniqueLabel*(*n*,1), **mk-***ENTERSTATENODE*(**DASH**, *undefined*, *empty*)) }

| *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*(*v*,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*(*j*,1), **mk-***CONTINUE*(*connector*)) }

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

| *d*=*Decision-node*(*question*, *onexcep*, *answerset*, *elseanswer*) =>
if *onexcep* = *undefined* **then** ∅ **else** *compileExpr*(*onexcep*, *question*.*startLabel*) **endif** ∪
(**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.. *iseq*.*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) ∪
⋃{ *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

2.2.3 Actions

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

compileExpr(*a*: *DefinitionAS1*, *next*: *LABEL*): *BEHAVIOUR* $=_{\mathrm{def}}$
case *a* **of**

All the contents of this function is 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.

| *o*=*On-exception*(*n*) =>
{**mk-***PRIMITIVE*(*uniqueLabel*(*o*,1), **mk-***SETHANDLER*(*n*, *scope*, *next*)) }
where
scope: *EXCEPTIONSCOPE* $=_{\mathrm{def}}$
case *o*.*parentAS1* **of**
| *Handle-node* => *esHandleClause*
| *Else-handle-node* => *esHandleClause*
| *State-start-node* => *esStimulusOrStart*
| *Procedure-start-node* => *esStimulusOrStart*
| *Input-node* => *esStimulusOrStart*
| *Spontaneous-transition* => *esStimulusOrStart*
| *Graph-node* => *esAction*
| *Terminator* => *esAction*

```
            | Decision-node => esAction
            | Connect-node => esStimulusOrStart
            | Named-start-node => esStimulusOrStart
            otherwise undefined
            endcase
endwhere

| Graph-node(action, onexcep) =>
        if onexcep = undefined then ∅ else compileExpr(onexcep, action.startLabel) endif ∪
        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) }

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

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

| c=Create-request-node(agentId, exprList) =>
        if exprList = empty then ∅
        else compileExpr(exprList.last, uniqueLabel(c,1)) ∪
            ⋃{ 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)) ∪
            ⋃{ 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 >,
                    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 ∈  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 ∈  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 sequentialised and evaluated from the last to the first; their result 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(c,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(c,1)) ∪
            ⋃{ compileExpr(exprList[i],  exprList[i+1]. startLabel) | i ∈ 1.. exprList.length-1 }
        endif ∪
        {mk-PRIMITIVE(uniqueLabel(c,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(c,1), mk-VAR(id, next)) }

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

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

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

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

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

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

This concludes the definition of the expression compilation function.

endcase // end of the compileExpr function definition

2.2.4 Start Labels

This clause introduces the function *startLabel* which is responsible to define 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**
| *h*=*Handle-node* (*, *, *, *trans*) => *startLabel*(*trans*)
| *e*=*Else-handle-node*(*, *trans*) => *startLabel*(*trans*)
| *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
| *o*=*On-exception*(*handler*) => *uniqueLabel*(*o*,1)
| *g*=*Graph-node*(*action*, *) => *startLabel*(*action*)
| *a*=*Assignment*(*, *expr*) => *startLabel*(*expr*)
| *a*=*Assignment-attempt*(*, *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*) => *uniqueLabel*(*c*,1)
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)
| *r*=*Raise-node*(*, *) => *uniqueLabel*(*r*,1)
| *d*=*Decision-node*(*question*, *, *, *) => *startLabel*(*question*)
| *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

2.3 SDL Abstract Machine Programs

Building on the SDL Abstract Machine and the compilation, SAM programs defining, for each SDL specification, the set of legal system runs are presented.

2.3.1 System Initialisation

Starting from any pre-initial state of S_0, the initialisation rules describe a recursive *unfolding* of the specified system instance according to its initial hierarchical structure. For each SDL agent instance, a corresponding ASM agent is created and initialised. Furthermore, ASM agents are created to model links and SDL agent sets.

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. (See Figure F3-3.)

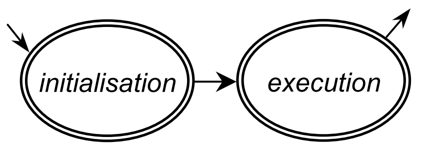


Figure F3-3/Z.100 – Activity phases of SDL agents and agent sets (level 1)

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
    behaviour = *rootNodeAS1*.*compile* ∧
    if *rootNodeAS1*.**s-***Agent-definition* ≠ *undefined* **then**
        system.*nodeAS1* = *rootNodeAS1*.**s-***Agent-definition* ∧
        system.*owner* = *undefined* ∧
        system.*agentMode1* = *initialisation* ∧
        system.*program* = AGENT-SET-PROGRAM
    else
        system.*program* = *undefined*
    endif

For a given SDL specification, the initial constraint distinguishes two cases. The first case applies when an agent definition is part of the SDL 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 agent sets in general. Further functions and domains are initialised when this program is executed, or are derived functions or derived domains. In the second case, no system agent is defined in the SDL specification, therefore, no behaviour is assigned via *program*.

2.3.1.2 Agent Set Creation, Initialisation, and Removal

ASM agents modelling SDL agent sets are created during system initialisation and possibly dynamically, during system execution. They can be understood as containers that reflect certain structural aspects of SDL systems, in particular agent hierarchy and the connection structure. These structural aspects are crucial to the intelligibility of SDL 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** $=_{\text{def}}$
                if *atd*.**s-***Agent-type-identifier* = *undefined* **then**
                    atd.**s-***Agent-definition*-**set**
                else
                    atd.**s-***Agent-definition*-**set** ∪
                    atd.**s-***Agent-type-identifier*.*idToNodeAS1*.*collectAllAgentDefinitions*
                endif
    endwhere

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

CREATEAGENTSET(*ow*:*SDLAGENT*, *ad*:*Agent-definition*) ≡
    extend *AGENT* **with** *sas*
        sas.*nodeAS1* := *ad*
        sas.*owner* := *ow*
        CREATEALLGATES (*ow*, *ad*.**s-***Agent-type-identifier*.*idToNodeAS1*)
        sas.*program* := AGENT-SET-PROGRAM
        sas.*agentMode1* := *initialisation*
    endextend

Creation of an SDL agent set is modelled by creating an ASM agent and initialising its control block. In particular, the node *Agent-definition* of the AST is assigned to the function *nodeAS1*, 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 agent instance. Further functions and domains are initialised when AGENT-SET-PROGRAM is executed, or are derived functions or derived domains. The initial agent instances of the considered SDL 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 agent set, too.

The creation of SDL 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 initialisation step, the agent set is switched to the execution mode.

```
INITAGENTSET ≡
    if Self.nodeAS1.s-Agent-type-identifier.idToNodeAS1.s-Agent-kind = SYSTEM then
        CREATEALLGATES(Self, Self.nodeAS1.s-Agent-type-identifier.idToNodeAS1)
    endif
    CREATEALLAGENTS(Self, null, Self.nodeAS1)
    Self.agentMode1 := execution
```

The initialisation 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 initialisation, the initial agent instances – in the case of *system* a single agent instance – are created. After this initialisation, the ASM agent is switched to the execution mode.

In case of the SDL agent set *system*, the gates of the system instance are created. The reasons why this is done during initialisation (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*.

2.3.1.3 Agent Creation, Initialisation, and Removal

The creation of SDL agent instances happens during system initialisation, 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 initialisation, which is defined subsequently.

The initialisation of an agent is decomposed into a sequence of phases, as shown in the state diagram in Figure F3-4. In each of these phases, certain parts of the agent’s structure are created. After agent initialisation, the agent execution is started.

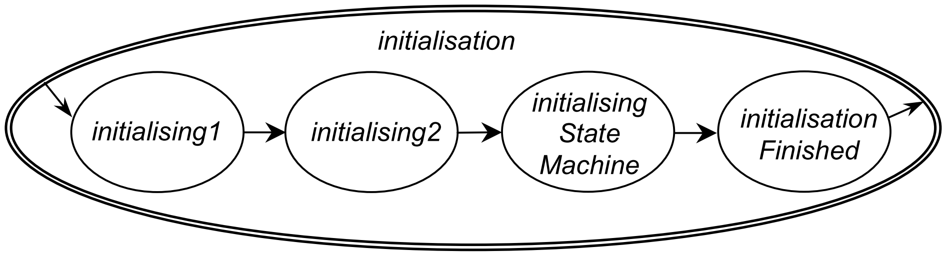


Figure F3-4/Z.100 – Activity phases of SDL agents: initialisation (level 2)

CREATEALLAGENTS(*ow*:*SDLAGENT*, *pa*:*PID*, *ad*:*Agent-definition*) ≡
do forall *i*: *i* ∈ 1 .. *ad*.**s-***Number-of-instances*.**s-***Initial-number*
CREATEAGENT(*ow*, *pa*, *ad*.**s-***Agent-type-identifier*.*idToNodeAS1*)
enddo

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

CREATEAGENT(*ow*:*SDLAGENTSET*, *pa*:*PID*, *atd*:*Agent-type-definition*) ≡
extend *AGENT* **with** *sa*
INITAGENTCONTROLBLOCK(*sa*, *ow*, *pa*, *atd*)
CREATEINPUTPORT(*sa*)
sa.*agentMode1* := *initialisation*
sa.*agentMode2* := *initialising1*
sa.*program* := AGENT-PROGRAM
endextend

where
INITAGENTCONTROLBLOCK(*sa*: *SDLAGENT*, *ow*:*SDLAGENTSET*, *pa*:*PID*,
atd:*Agent-type-definition*) ≡
sa.*nodeAS1* := *atd*
sa.*owner* := *ow*
sa.*isActive* := *undefined*
sa.*currentStartNodes* := ∅
sa.*currentConnector* := *undefined*
sa.*callingProcedureNode* := *undefined*
sa.*currentSignalInst* := *undefined*
sa.*currentExceptionInst* := *undefined*
sa.*parent* := *pa*
sa.*sender* := *null*
sa.*offspring* := *null*
sa.*self* := **mk-***PID*(*sa*, *atd*.**s-***Interface-definition*)
if *pa* ≠ *null* **then**
pa.*offspring* := **mk-***PID*(*sa*, *atd*.**s-***Interface-definition*)
endif
if *ow*.*nodeAS1*.**s-***Agent-kind* ∈ {**SYSTEM**, **BLOCK**} **then** // containing agent set
sa.*stateAgent* := *sa*
elseif *ow*.*owner*.*owner*.*nodeAS1*.**s-***Agent-kind* = **PROCESS** **then** // next level agent set
sa.*stateAgent* := *ow*.*owner*.*stateAgent*
else
sa.*stateAgent* := *sa*
endif
endwhere

To create an agent, the controlled domain *AGENT* is extended. The control block of this new agent is initialised. An input port for receiving signals from other agents is created and attached to the new agent. 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 initialisation, the agent is switched to the execution mode. Additionally, the agent synchronises in case it belongs to a set of nested agents, in order to obtain an interleaving execution amongst these agents.

```
INITAGENT ≡
    if Self.agentMode2 = initialising1 then
        CREATEAGENTVARIABLES(Self, Self.nodeAS1)
        CREATEALLAGENTSETS(Self, Self.nodeAS1)
        CREATESTATEMACHINE(Self.nodeAS1.s-State-machine-definition)
        Self.agentMode2 := initialising2
    elseif Self.agentMode2 = initialising2 then
        CREATEALLCHANNELS(Self, Self.nodeAS1)
        CREATEALLLINKS(Self)
        Self.agentMode2 := initialisingStateMachine
    elseif Self.agentMode2 = initialisingStateMachine then
        INITSTATEMACHINE
    elseif Self.agentMode2 = initialisationFinished then
        Self.agentMode1 := execution
        Self.agentMode2 := startPhase
    endif
```

The initialisation 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 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 initialised, 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.

2.3.1.4 Procedure Creation and Initialisation

The creation of SDL 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.

The initialisation of a procedure is decomposed into a sequence of phases, as shown in the state diagram in Figure F3-5. In each of these phases, certain parts of the procedure’s structure are created. After procedure initialisation, the agent execution is continued.

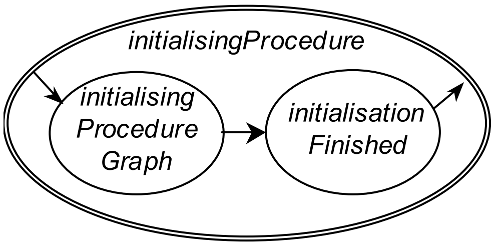


Figure F3-5/Z.100 – Activity phases of SDL agents: firing of transitions (level 4)

CREATEPROCEDURE(*pd*:*Procedure-definition*,*cl*:*CONTINUELABEL*) ≡
  CREATEPROCEDUREGRAPH(*pd*, *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*, **DEFAULT**)}
    Self.*agentMode3* := *enteringStateNode*
    Self.*agentMode4* := *startPhase*
    Self.*currentLabel* := *undefined*
  endif

The initialisation 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 initialised, 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.

2.3.1.5 Gate Creation

Exchange of signals between SDL 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
          atd.**s-***Gate-definition* **-set** ∪
          atd.**s-***Agent-type-identifier*.*idToNodeAS1*.*collectAllGateDefinitions*
        endif
  endwhere

SDL agent sets are created when the surrounding SDL agent is initialised 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.nodeAS1 := *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.nodeAS1 := *gd*
g.schedule := *empty*
g.direction := *outDir*
endextend
endif

For each SDL gate, one or two elements of the controlled domain *GATE* (also called "gates") are added, depending on whether the gate is uni- 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 initialised to be empty.

CREATEINPUTPORT(*ow*:*AGENT*) ≡
extend *GATE* **with** *g*
g.myAgent := *ow*
g.nodeAS1 := *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 initialised. In addition, the created gate explicitly becomes the input port of the owning agent.

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** $=_{\text{def}}$
if *atd*.**s-***Agent-type-identifier* = *undefined* **then**
atd.**s-***Channel-definition***-set**
else
atd.**s-***Channel-definition***-set** ∪
atd.**s-***Agent-type-identifier.idToNodeAS1.collectAllChannelDefinitions*
endif
endwhere

Channels are created by agents during the second phase of their initialisation. 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*) ≡
choose *fromGate*: *fromGate* ∈ *GATE* ∧ *fromGate.nodeAS1*=*cp*.**s-***Originating-gate.idToNodeAS1* ∧
(*OuterGate*(*ow*, *fromGate*, *inDir*) ∨ *InnerGate*(*ow*, *fromGate*, *outDir*))
choose *toGate*: *toGate* ∈ *GATE* ∧ *toGate.nodeAS1* = *cp*.**s-***Destination-gate.idToNodeAS1* ∧
(*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* $=_{\text{def}}$
g.myAgent = *ow.owner* ∧ *g.direction* = *dir*
InnerGate(*ow*: *AGENT*, *g*: *GATE*, *dir*: *DIRECTION*): *BOOLEAN* $=_{\text{def}}$
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.

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.

CREATEALLLINKS(*ow*:*AGENT*) ≡
do forall *g*: *g* ∈ *ow.ingates*
CREATELINK(*ow*, *g*, *ow.inport*, **NODELAY**, *g.nodeAS1*.**s-***In-signal-identifier***-set**, *undefined*)
enddo

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.nodeAS1 := *cd*
l.owner := *ow*
l.from := *fromGate*
l.to := *toGate*
l.noDelay := *nd*
l.with := *w*
l.program := LINK-PROGRAM
endextend

LINK-PROGRAM:

FORWARDSIGNAL

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.

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 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
        atd.s-Variable-definition-set ∪
        atd.s-Agent-type-identifier.idToNodeAS1.collectAllVariableDefinitions
      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,
      sn.parentStateNode.stateId, cstd.collectAllVariableDefinitions)
  endextend

  where
    collectAllVariableDefinitions(cstd: Composite-state-type-definition):
      Variable-definition-set =def
      if cstd.s-Composite-state-type-definition = undefined then
        cstd.s-Variable-definition-set
      else
        cstd.s-Variable-definition-set ∪
        cstd.s-Composite-state-type-definition.idToNodeAS1.collectAllVariableDefinitions
      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
    sa.stateAgent.state := initProcedureState(sa.stateAgent.state, sid,
      sn.parentStateNode.stateId, pd.collectAllVariableDefinitions,
      pd.collectAllProcedureFPars)
  endextend
```

where
collectAllVariableDefinitions(*pd*: *Procedure-definition*): *Variable-definition*-**set** $=_{\text{def}}$
if *pd*.**s**-*Procedure-identifier* = *undefined* **then**
pd.**s**-*Variable-definition*-**set**
else
pd.**s**-*Variable-definition*-**set** ∪
pd.**s**-*Procedure-identifier*.*idToNodeAS1*.*collectAllVariableDefinitions*
endif

collectAllProcedureFPars(*pd*:*Procedure-definition*): *Procedure-formal-parameter** $=_{\text{def}}$
if *pd*.**s**-*Procedure-identifier* = *undefined* **then**
pd.**s**-*Procedure-formal-parameter**
else
pd.**s**-*Procedure-identifier*.*idToNodeAS1*.*collectAllProcedureFPars* ⁀
pd.**s**-*Procedure-formal-parameter**
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.

2.3.1.9 State Machine Creation and Initialisation

The behaviour of an SDL 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-definition*) ≡
CREATETOPSTATEPARTITION(*smd*)

When an SDL 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 agent has a behaviour, the root node is refined (and possibly specialised) subsequently. If the agent is passive, no refinement is made. The unfolding of the graph is treated by the macro INITSTATEMACHINE.

If an SDL agent has a 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*.*ehNodesToBeCreated* ≠ ∅ **then**
CREATEEXCEPTIONHANDLERENODE
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.

2.3.1.10 Procedure Graph Creation and Initialisation

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(*pg*:*Procedure-graph*,*cl*:*CONTINUELABEL*) ≡
CREATEPROCEDURESTATENODE(*pg*,*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.ehNodesToBeCreated* ≠ ∅ **then**
CREATEEXCEPTIONHANDLERENODE
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.

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.nodeAS1 := *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
Self.stateNodesToBeRefined := *Self.stateNodesToBeRefined* ∪ {*sn*}
Self.stateNodesToBeSpecialised := *Self.stateNodesToBeSpecialised* ∪ {*sn*}
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 book-keeping information is initialised. 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*, *cl*:*CONTINUELABEL*) ≡
extend *STATENODE* **with** *sn*
sn.*nodeAS1* := *pd*.**s-***Procedure-graph*
sn.*owner* := *Self*
sn.*parentStateNode* := *Self*.*currentParentStateNode*
sn.*stateNodeKind* := *procedureNode*
sn.*stateName* := *undefined*
sn.*stateTransitions* := ∅
sn.*startTransitions* := ∅ // updated if the state node is refined
Self.*stateNodesToBeRefined* := {*sn*}
Self.*stateNodesToBeCreated* := ∅
Self.*statePartitionsToBeCreated* := ∅
Self.*stateNodesToBeSpecialised* := {*sn*}
Self.*currentProcedureStateNode* := *sn*
Self.*callingProcedureNode* := *Self*.*currentParentStateNode*.*previousProcedureNode*
CREATEPROCEDUREVARIABLES(*Self*,*sn*,*pd*)
SAVEPROCEDURECONTROLBLOCK(*sn*,*cl*)
SAVEEXCEPTIONSCOPES(*sn*)
endextend

where
previousProcedureNode(*sn*:*STATENODE*): *STATENODE* $=_{\mathrm{def}}$
if *sn*.*stateNodeKind* = *procedureNode* **then** *sn*
else
let *psn* = *sn*.*parentStateNode* **in**
if *psn* = *undefined* **then** *undefined*
elseif *psn*.*stateNodeKind* = *procedureNode* **then** *psn*
else *previousProcedureNode*(*psn*)
endif
endlet
endif
endwhere

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.

CREATEEXCEPTIONHANDLERENODE ≡
choose *ehnd*: *ehnd* ∈ *Self*.*ehNodesToBeCreated*
Self.*ehNodesToBeCreated* := *Self*.*ehNodesToBeCreated* \ {*ehnd*}
extend *STATENODE* **with** *ehn*
ehn.*nodeAS1* := *ehnd*
ehn.*owner* := *Self*
ehn.*parentStateNode* := *Self*.*currentParentStateNode*
ehn.*stateNodeKind* := *handlerNode*
ehn.*stateName* := *ehnd*.**s-***Exception-handler-name*
ehn.*stateTransitions* := *ehnd*.*getHandleNodes*
endextend
endchoose

Exception handler nodes are created as part of a state transition graph and a procedure graph.

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-definition) ≡
  extend STATENODE with sn
    sn.nodeAS1 := smd
    sn.owner := Self
    Self.topStateNode := sn
    sn.parentStateNode := undefined
    sn.stateNodeKind := statePartition
    sn.stateName := smd.s-State-name
    sn.stateTransitions := ∅
    sn.startTransitions := ∅                // updated if the state partition is refined
    if smd ≠ undefined then
      Self.stateNodesToBeRefined := {sn}
      Self.stateNodesToBeSpecialised := {sn}
    else
      Self.stateNodesToBeRefined := ∅
      Self.stateNodesToBeSpecialised := ∅
    endif
    Self.stateNodesToBeCreated := ∅
    Self.statePartitionsToBeCreated := ∅
    Self.stateNodesToBeSpecialised := ∅
  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 book-keeping information is initialised. In particular, the root node is classified as a state partition. If the agent has a 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.nodeAS1 := 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, sn) := 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
```

(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 book-keeping information is initialised. Modelling a state partition, the created state node may have a substructure, and is therefore included in the set of state nodes to be refined.

```
CREATEINHERITEDSTATE ≡
    choose sns: sns ∈ Self.stateNodesToBeSpecialised
        Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised \ {sns}
        let cstd = sns.nodeAS1.s-Composite-state-type-identifier.idToNodeAS1 in
            if cstd.s-Composite-state-type-identifier ≠ undefined then
                extend STATENODE with sn
                    sn.nodeAS1 := cstd.s-Composite-state-type-identifier.idToNodeAS1
                    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
            else
                sns.inheritedStateNode := undefined
            endif
        endlet
    endchoose
```

Specialisation of composite state types is modelled by adding another dimension to the hierarchical state graph, yielding a "hierarchical *inheritance* state graph". Formally, specialisation 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 specialisation, then a root node that is inherited by the refined state node, and has the composite state type being specialised, is created. By adding the root node to the set of state nodes to be refined, a "hierarchical inheritance state graph" modelling the specialisation is subsequently attached to this root node.

2.3.1.13 Composite State Creation

All (regular) state nodes, state partitions, and procedure nodes are candidates for refinement and, if refined, for specialisation. Refinements are defined by a composite state type, which includes another composite state type in case of specialisation. 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.nodeAS1 ∈ Procedure-graph then
            CREATEPROCEDUREVARIABLES(Self, snr, snr.nodeAS1)
            CREATEPROCEDUREGRAPHNODES(snr, snr.nodeAS1)
        elseif snr.nodeAS1.s-Composite-state-type-identifier ≠ undefined then
            CREATECOMPOSITESTATEVARIABLES(Self, snr,
                snr.nodeAS1.s-Composite-state-type-identifier)
            CREATECOMPOSITESTATE(snr,
                snr.nodeAS1.s-Composite-state-type-identifier.idToNodeAS1)
        else
            snr.stateNodeRefinement := undefined
        endif
    endchoose
```

When a state node, state partition, or procedure node is created, it is added to 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*} ∪
  csgd.**s-***Named-start-node***-set**)
 psn.*freeActions* := *getFreeActions*(*csgd*.**s-***State-transition-graph*)
 CREATESTATETRANSITIONGRAPH(*psn*,*csgd*.**s-***State-transition-graph*)

Creating a composite state graph means creating its state transition graph.

CREATESTATETRANSITIONGRAPH(*psn*:*STATENODE*, *stgd*:*State-transition-graph*) ≡
 Self.*stateNodesToBeCreated* := *stgd*.**s-***State-node***-set**
 Self.*ehNodesToBeCreated* := *stgd*.**s-***Exception-handler-node***-set**
 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*)
 Self.*stateNodesToBeCreated* := *pg*.**s-***State-node***-set**
 Self.*ehNodesToBeCreated* := *pg*.**s-***Exception-handler-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***-set**
 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*.

2.3.2 System Execution

After initialisation, SDL agents start their execution. The execution of the system is modelled by the concurrent execution of all its agents.

2.3.2.1 Agent Set Execution

EXECAGENTSET ≡
  DELIVERSIGNALS

The behaviour of agent sets is formalised 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.self* = *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

2.3.2.2 Agent Execution

The execution of SDL agents is modelled by alternating phases, namely transition selection and transition firing, preceded by a start phase. 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.

An agent reaches the execution phase after it has completed its initialisation. The execution phase consists of three subphases as shown in the state diagram in Figure F3-6. Two of these subphases will in turn be refined, which is indicated by the double line.

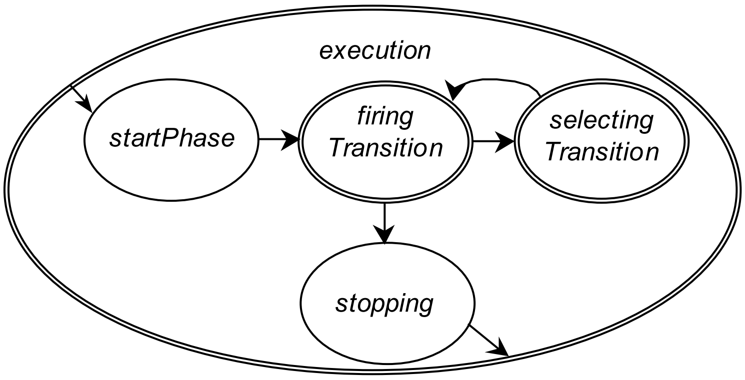


Figure F3-6/Z.100 – Activity phases of SDL agents: execution (level 2)

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* =$_{\text{def}}$
sa.stateAgent.isActive = *sa* ∨ *sa.owner.nodeAS1*.**s-***Agent-kind* ∈ {**BLOCK**, **SYSTEM**}

2.3.2.3 Starting Agent Execution

When the execution phase starts, several initialisations are made: the set of state nodes to be entered is initialised 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*,**DEFAULT**)}
Self.agentMode2 := *firingTransition*
Self.agentMode3 := *enteringStateNode*
Self.agentMode4 := *startPhase*
Self.currentLabel := *undefined*

2.3.2.4 Transition Selection

In agent mode *selectingTransition* (*agentMode2*), an SDL agent searches for a fireable transition. ITU-T Z.100 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 formalising the transition selection.

In order to structure the transition selection, several agent mode levels are defined. The uppermost level is shown in the diagram in Figure F3-7, where the agent mode *selectingTransition* is refined into four submodes (*agentMode3*). Some of these submodes will in turn be refined later.

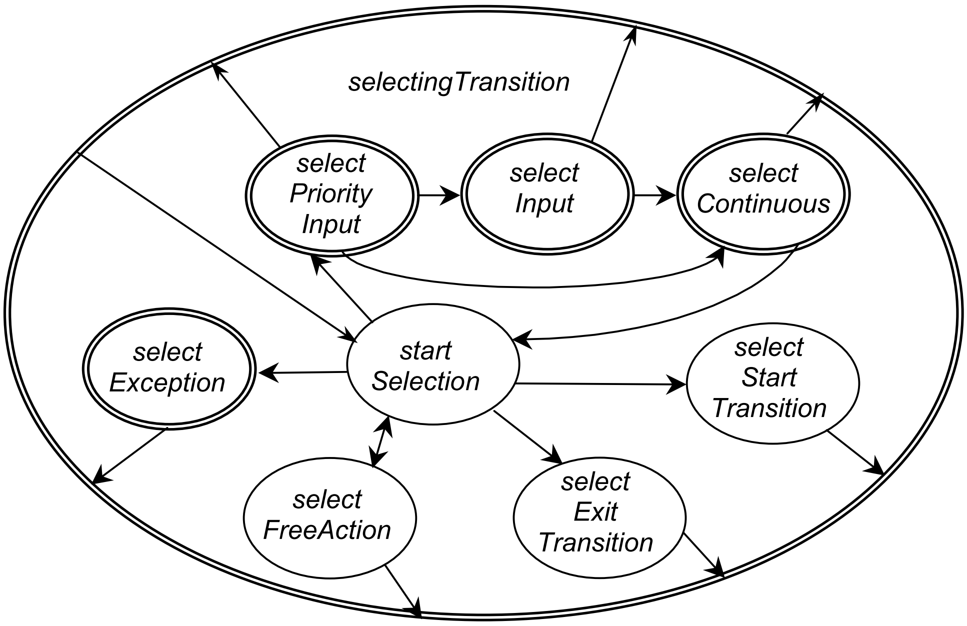


Figure F3-7/Z.100 – Activity phases of SDL agents: selecting transition (level 3)

SELECTTRANSITION ≡
if *Self.agentMode3* = *startSelection* **then**
SELECTTRANSITIONSTARTPHASE
elseif *Self.agentMode3* = *selectException* **then**

SELECTEXCEPTION
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 handle node, 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*:*TRANSITION*) ≡
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*:*TRANSITION*, *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.

HANDLENODEFOUND(*t*:*TRANSITION*) ≡
RESETEXCEPTIONHANDLER(*Self.exceptionScopesToBeChecked.head*)
Self.currentStateId := *Self.currentParentStateNode.stateId*
Self.currentExceptionInst := *undefined*
Self.currentSignalInst := *Self.currentExceptionInst*
Self.currentLabel := *t*.**s2**-*LABEL* // second label
Self.agentMode2 := *firingTransition*
Self.agentMode3 := *firingAction*

When a handle node is found, the start label of the transition is assigned, and the agent modes are set appropriately. The current exception instance becomes the current signal instance, which is consumed by the handle node.

2.3.2.5 Starting Selection of Transitions

When the selection of transition starts, several initialisations 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.currentExceptionInst* ≠ *undefined* **then**
Self.agentMode3 := *selectException*
Self.agentMode4 := *startPhase*
elseif *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

2.3.2.6 Exception Selection

Selection of a handle node is performed by checking active exception handler nodes until an applicable handle node is found, or until all active exception handler nodes have been checked. Inheritance is taken into account by checking, for each exception handler node, the inherited handler nodes.

The selection of exceptions consists of the subphases shown in the state diagram in Figure F3-8.

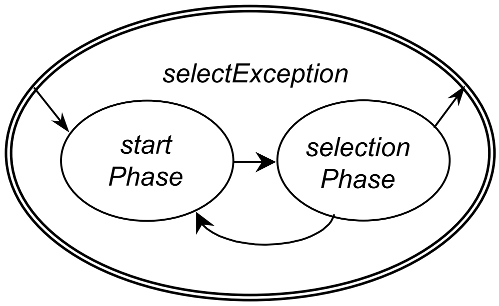


Figure F3-8/Z.100 – Activity phases of SDL agents: selection exceptions (level 4)

```
SelectException ≡
    if Self.agentMode4 = startPhase then
        SelExceptionStartPhase
    elseif Self.agentMode4 = selectionPhase then
        SelExceptionSelectionPhase
    endif
```

This ASM macro defines the upper level control structure of the exception selection. Depending on the agent mode *agentMode4*, further action is defined in the corresponding ASM macro.

```
SelExceptionStartPhase ≡
    Self.ehParentStateNodeChecked := Self.currentParentStateNode
    Self.exceptionScopesToBeChecked := nextActiveHandlerScope(Self,exceptionScopeSeq)
    Self.agentMode4 := selectionPhase
```

When the selection starts, several initialisations are made, and the selection is activated. The start phase may be repeated during the selection, if the current parent state node is modified.

```
nextActiveHandlerScope(sa: SdlAgent, scopeSeq: ExceptionScope*): ExceptionScope* =def
    if scopeSeq = empty then empty
    elseif activeHandler(sa, scopeSeq.head) ≠ undefined then scopeSeq
    else nextActiveHandlerScope(sa, scopeSeq.tail)
    endif
```

This function determines the scope of the next active exception handler, where the order is determined by the parameter *scopeSeq*.

```
SelExceptionSelectionPhase ≡
    if Self.exceptionScopesToBeChecked = empty then
        NextUpperStateNodeToBeChecked
    else
        let ehn = exceptionHandlerNode(Self, Self.ehParentStateNodeChecked,
            Self.exceptionScopesToBeChecked.head) in
            if ehn ≠ undefined then
                SelectHandleNode
            else
                NextStateNodeToBeChecked
            endif
        endlet
    endif

    where
        exceptionHandlerNode(sa: SdlAgent, psn: StateNode,
                scope: ExceptionScope): StateNode =def
                take({ehn ∈ StateNode: ehn.stateName = activeHandler(sa,scope) ∧
                    ehn.stateNodeKind = handlerNode ∧ ehn.parentStateNode = psn})
```

```
previousStateGraphNode(sn: STATENODE): STATENODE =def
        let psn = sn.parentStateNode in
            if psn = undefined then
                undefined
            elseif psn.stateNodeRefinement = compositeStateGraph then psn
            else previousStateGraphNode(psn)
            endif
        endlet

SELECTHANDLENODE ≡
        let t = take({t ∈ ehn.stateTransitions:
            t.s-SIGNAL = Self.currentExceptionInst.signalType}) in
            if t ≠ undefined then
                HANDLENODEFOUND(t)
            else
                let t = take({t ∈ ehn.stateTransitions: t.s-SIGNAL = ELSE}) in
                    if t ≠ undefined then
                        HANDLENODEFOUND(t)
                    else
                        NEXTSTATENODETOBECHECKED
                    endif
                endlet
            endif
        endlet

NEXTSTATENODETOBECHECKED ≡
        if Self.ehParentStateNodeChecked.inheritedStateNode ≠ undefined then
            Self.ehParentStateNodeChecked :=
                Self.ehParentStateNodeChecked.inheritedStateNode
        else
            Self.ehParentStateNodeChecked := Self.currentParentStateNode
            Self.exceptionScopesToBeChecked := nextActiveHandlerScope(Self,
                Self.exceptionScopesToBeChecked.tail)
        endif

NEXTUPPERSTATENODETOBECHECKED ≡
        let sn = Self.currentParentStateNode.previousStateGraphNode in
            if sn = undefined then
                UNDEFINEDBEHAVIOUR
            else
                Self.currentParentStateNode := sn
                if Self.currentParentStateNode.stateNodeKind = procedureNode then
                    RESTOREEXCEPTIONSCOPES(Self.currentParentStateNode)
                endif
                do forall sn1 ∈ collectCurrentSubStates({sn})
                    sn1.currentSubStates := ∅
                    sn1.currentExitPoints := ∅
                enddo
                Self.agentMode4 := startPhase
            endif
        endlet
    endwhere
```

During the exception selection phase, all applicable exception handler nodes are checked to select a handle node that is able to consume the current exception instance.

2.3.2.7 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-STATEENTRYPOINT}) in
            if t ≠ undefined then
                STARTTRANSITIONFOUND(t, Self.startNodeChecked.s-STATENODE)
            else
                Self.stateNodeChecked :=
                    take({sn1 ∈ snSet: DirectlyInheritsFrom(Self.stateNodeChecked,sn1)})
            endif
        endlet
    endif
```

Start transitions are associated directly with the refined node, and are distinguished by their state entry point.

2.3.2.8 Exit Transition Selection

```
SELECTEXITTRANSITION ≡
    if Self.stateNodeChecked = undefined then
        let snwex = take(Self.currentExitStateNodes) in
            if snwex ≠ undefined then
                Self.currentExitStateNodes := Self.currentExitStateNodes \ {snwex}
                Self.exitNodeChecked := snwex
                Self.stateNodeChecked := snwex.s-STATENODE
            endif
        endlet
    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 ∈ snSet: DirectlyInheritsFrom(Self.stateNodeChecked,sn1)})
            endif
        endlet
    endif
```

Exit transitions are associated with the containing node, and are distinguished by their state exit point.

2.3.2.9 Free Action Selection

```
SELECTFREEACTION ≡
        let fa = take({Self.stateNodeChecked.freeActions:
            fa.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* ∈ *snSet*: *DirectlyInheritsFrom*(*Self.stateNodeChecked*,*sn1*)})
endif
endlet

Free actions are associated directly with the refined node, and are distinguished by their connector name.

2.3.2.10 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.

The selection of a priority input consists of the subphases (*agentMode4*) shown in the diagram in Figure F3-9. 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*.

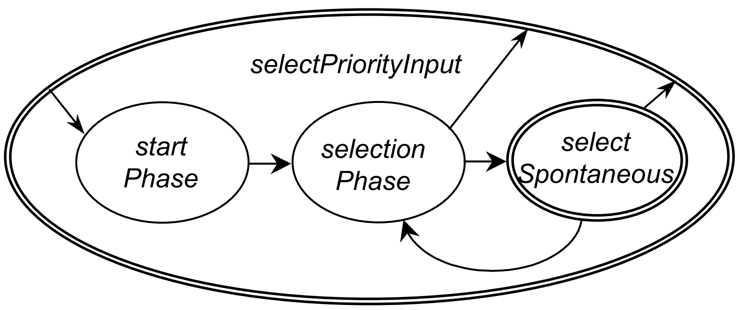


Figure F3-9/Z.100 – Activity phases of SDL agents: selecting priority inputs (level 4)

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.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 initialisations 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*

```
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 ≠ ∅ then
            if Self.stateNodeChecked = undefined then
                SELECTNEXTSTATENODE
            else
                CHECKFORINHERITEDSTATENODES
            endif
        else
            NEXTSIGNALTOBECHECKED
            Self.stateNodesToBeChecked := collectCurrentSubStates({Self.topStateNode})
            Self.stateNodeChecked := undefined
        endif

    SELECTNEXTSTATENODE ≡
        let sn = Self.stateNodesToBeChecked.selectNextStateNode in
            if 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}
            endif
        endlet

    CHECKFORINHERITEDSTATENODES ≡
        Self.stateNodeChecked := undefined
        let sn1 = Self.stateNodeChecked in
            if Self.signalChecked.signalType ∈          // check: redefinition possible
                {in.s-Signal-identifier | in∈ sn1.nodeAS1.s-Input-node-set} ∪
                sn1.nodeAS1.s-Save-signalset then
                Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \
                    {sn2 ∈ Self.stateNodesToBeChecked | InheritsFrom(sn1,sn2)}
            endif
        endlet

    NEXTSIGNALTOBECHECKED ≡
        let si = nextSignal(Self.signalChecked, Self.inputPortChecked) in
            if si ≠ undefined then
                Self.signalChecked := si
            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.

2.3.2.11 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.

The selection of an ordinary input consists of the subphases shown in the state diagram in Figure F3-10. 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*.

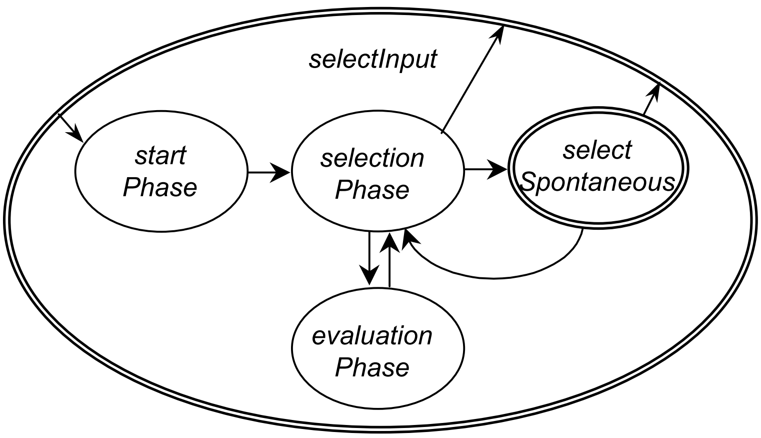


Figure F3-10/Z.100 – Activity phases of SDL agents: selecting inputs (level 4)

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 initialisations 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
        NEXTSTATENODETOBECHECKED
    elseif Self.Spontaneous then
        Self.agentMode4 := selectSpontaneous
    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:TRANSITION) ≡
            Self.transitionChecked := t
            Self.currentStateId := Self.stateNodeChecked.parentStateNode.stateId
            Self.currentLabel := t.s-LABEL
            Self.agentMode4 := evaluationPhase

        NEXTSTATENODETOBECHECKED ≡
            if Self.stateNodesToBeChecked ≠ ∅ then
                if Self.stateNodeChecked = undefined then
                    SELECTNEXTSTATENODE
                else
                    CHECKFORINHERITEDSTATENODES
                endif
            else
                if ¬ Self.SignalSaved then                    // implicit transition
                    DELETE(Self.signalChecked,Self.inport)
                endif
                NEXTSIGNALTOBECHECKED
                Self.stateNodesToBeChecked := collectCurrentSubStates({Self.topStateNode})
                Self.stateNodeChecked := undefined
            endif

        SELECTNEXTSTATENODE ≡
            let sn = Self.stateNodesToBeChecked.selectNextStateNode in
                if 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.nodeAS1.s-Save-signalset then
                Self.SignalSaved := True
            endif
        endif
    endlet

CHECKFORINHERITEDSTATENODES ≡
    Self.stateNodeChecked := undefined
    let sn1 = Self.stateNodeChecked in
        if Self.signalChecked.signalType ∈          // check: redefinition possible
            {in.s-Signal-identifier | in∈ sn1.nodeAS1.s-Input-node-set} ∪
            sn1.nodeAS1.s-Save-signalset then
            Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \
                {sn2 ∈ Self.stateNodesToBeChecked | InheritsFrom(sn1,sn2)}
        endif
    endlet

NEXTSIGNALTOBECHECKED ≡
    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-PRIMITIVE)
        endchoose
    elseif semvalue(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.

2.3.2.12 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 stared.

The selection of an continuous signals consists of the subphases shown in the state diagram in Figure F3-11. The control is identical to the selection of an ordinary input.

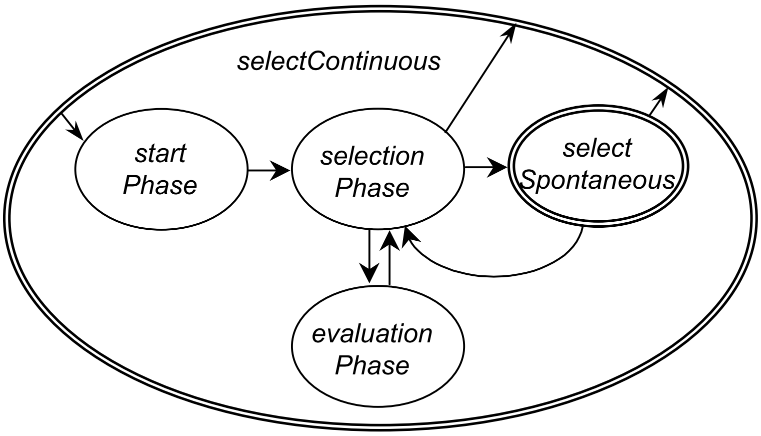


Figure F3-11/Z.100 – Activity phases of SDL agents: selecting continuous signals (level 4)

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 initialisations 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**
    NEXTSTATENODETOBECHECKED
  elseif *Self.Spontaneous* **then**
    Self.agentMode4 := *selectSpontaneous*
  else
    let *t* = *selectContinuousSignal*(*Self.transitionsToBeChecked*, *Self.continuousPriorities*) **in**

```
            if t ≠ undefined then
                Self.transitionsToBeChecked := Self.transitionsToBeChecked \ {t}
                if t.s-LABEL ≠ undefined then
                    EVALUATEENABLINGCONDITION(t)
                else
                    TRANSITIONFOUND(t)
                endif
            else
                NEXTSTATENODETOBECHECKED
            endif
        endlet
    endif

    where
        EVALUATEENABLINGCONDITION(t:TRANSITION) ≡
            Self.transitionChecked := t
            Self.currentStateId := Self.stateNodeChecked.parentStateNode.stateId
            Self.currentLabel := t.s-LABEL
            Self.agentMode4 := evaluationPhase

        NEXTSTATENODETOBECHECKED ≡
            if Self.stateNodesToBeChecked ≠ ∅ then
                if Self.stateNodeChecked = undefined then
                    SELECTNEXTSTATENODE
                else
                    CHECKFORINHERITEDSTATENODES
                endif
            else
                Self.agentMode3 := startSelection
                RETURNEXECRIGHT
            endif

        SELECTNEXTSTATENODE ≡
            let sn = Self.stateNodesToBeChecked.selectNextStateNode in
                if 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 := sn1.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-***PRIMITIVE*)
endchoose
elseif *semvalue*(*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.

2.3.2.13 Spontaneous Transition Selection

Selection of a spontaneous transition is performed by checking, at any time during the selection process, a single spontaneous transition.

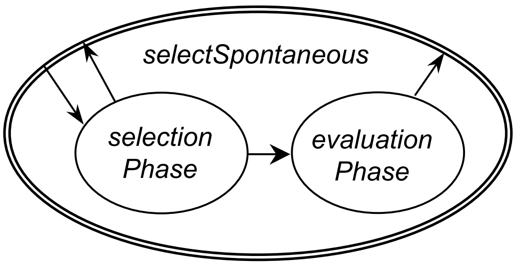


Figure F3-12/Z.100 – Activity phases of SDL agents: selecting spontaneous transitions (level 5)

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 in Figure F3-12.

SELECTSPONTANEOUS ≡
if *Self.agentMode5* = *selectionPhase* **then**
SELSPONTANEOUSSELECTIONPHASE
elseif *Self.agentMode5* = *evaluationPhase* **then**
SELSPONTANEOUSEVALUATIONPHASE
endif

This ASM macro defines the upper level control structure of the spontaneous transition selection. Depending on the agent mode*agentMode5*, further action is defined in the corresponding ASM macro. This control structure is part of the previous state diagram.

SELSPONTANEOUSSELECTIONPHASE ≡
if *Self.stateNodeChecked.stateTransitions.spontaneousTransitions* ≠ ∅ **then**
choose *t*: *t* ∈ *Self.stateNodeChecked.stateTransitions.spontaneousTransitions*
if *t*.**s-***LABEL* ≠ *undefined* **then**
EVALUATEENABLINGCONDITION(*t*)

else
Self.sender := *Self.self*
TRANSITIONFOUND(*t*)
endif
endchoose
endif

where
EVALUATEENABLINGCONDITION(*t*:*TRANSITION*) ≡
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-***PRIMITIVE*)
endchoose
elseif *semvalue*(*value*(*Self.transitionChecked*.**s-***LABEL*,*Self*)) **then**
Self.sender := *Self.self*
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.

2.3.2.14 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. See Figure F3-13.

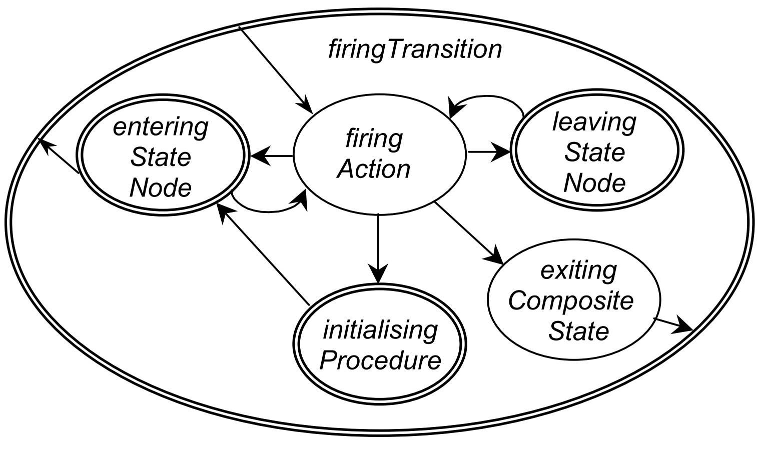


Figure F3-13/Z.100 – Activity phases of SDL agents: firing transitions (level 3)

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.

2.3.2.15 Firing of Actions

FIREACTION ≡
if *Self.currentLabel* ≠ *undefined* **then**
choose *b*: *b* ∈ *behaviour* ∧ *b*.**s-***LABEL* = *Self.currentLabel*
EVAL(*b*.**s-***PRIMITIVE*)
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.

2.3.2.16 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. See Figure F3-14.

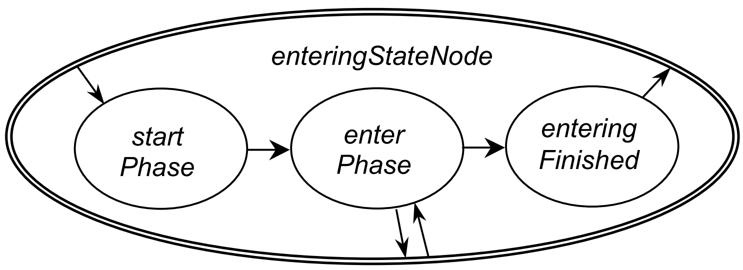


Figure F3-14/Z.100 – Activity phases of SDL agents: entering state node (level 4)

ENTERSTATENODESSTARTPHASE ≡
Self.agentMode4 := *enterPhase*

At the beginning of this phase, the set of entered state nodes is initialised. 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 := ∅
            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) ≡
        if snwen.s-STATENODE.inheritedStateNode ≠ undefined then
            // refinement possibly inherited
            Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪
                {mk-STATENODEWITHENTRYPOINT(snwen.s-STATENODE.inheritedStateNode,
                snwen.s-STATEENTRYPOINT)}
        else
            Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen}
        endif

    REFINEMENTSTATEAGGRNODE(snwen:STATENODEWITHENTRYPOINT) ≡
        Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪
            {snwen ∈ STATENODEWITHENTRYPOINT: snwen = mk-STATENODEWITHENTRYPOINT(sp,
            entryConnection(snwen.s-STATEENTRYPOINT, sp)) ∧
            sp ∈ snwen.s-STATENODE.statePartitionSet}
        let cstd = snwen.s-STATENODE.nodeAS1.s-Composite-state-type-identifier.idToNodeAS1 in
            if cstd.s-State-aggregation-node.s-Entry-procedure-definition ≠ undefined then
                CREATEPROCEDURE(cstd.s-State-aggregation-node.s-Entry-procedure-definition,
                    undefined)
            endif
        endlet

    REFINEMENTCOMPSTATENODE(snwen:STATENODEWITHENTRYPOINT) ≡
        Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen}
        let cstd = snwen.s-STATENODE.nodeAS1.s-Composite-state-type-identifier.idToNodeAS1 in
            if cstd.s-Composite-state-graph.s-Entry-procedure-definition ≠ undefined then
                CREATEPROCEDURE(cstd.s-Composite-state-graph.s-Entry-procedure-definition,
                    undefined)
            endif
        endlet
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.

2.3.2.17 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. See Figure F3-15.

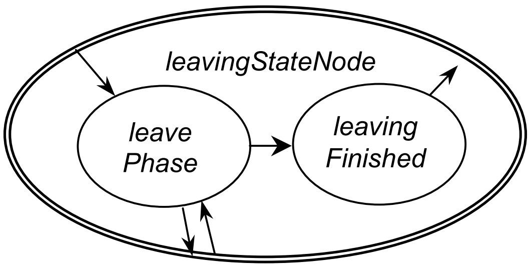


Figure F3-15/Z.100 – Activity phases of SDL agents: leaving state node (level 4)

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*}
if *sn.stateNodeRefinement* = *compositeStateGraph* **then**
let *cstd* = *sn.nodeAS1*.**s-***Composite-state-type-identifier.idToNodeAS1* **in**
if *cstd*.**s-***Composite-state-graph*.**s-***Exit-procedure-definition* ≠ *undefined* **then**
CREATEPROCEDURE(*cstd*.**s-***Composite-state-graph*.**s-***Exit-procedure-definition*,
undefined)
endif
endlet
elseif *sn.stateNodeRefinement* = *stateAggregationNode* **then**
let *cstd* = *sn.nodeAS1*.**s-***Composite-state-type-identifier.idToNodeAS1* **in**
if *cstd*.**s-***State-aggregation-node*.**s-***Exit-procedure-definition* ≠ *undefined* **then**
CREATEPROCEDURE(*cstd*.**s-***State-aggregation-node*.**s-***Exit-procedure-definition*,
undefined)
endif
endlet
endif
endif
endlet

In the leave phase, state nodes that have been collected are left, from bottom to top, with possible synchronisation 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.

2.3.2.18 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
```

2.3.2.19 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

2.3.3 Interface between Execution and Compilation

The execution of agents requires that certain behaviour parts called “compilation units” are 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*): associated with the containing state node
- exit transitions (*Named-return-node*): associated with the set of transitions of the containing state node
- exception transitions (*Handle-node*): associated with an exception handler
- 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.

3 Data semantics

3.1 Predefined Data

An operator is functional if it is predefined.

functional(*procedure*, *values*) $=_{\text{def}}$
procedure.**s-***Qualifier*.*head* ∈ *Package-qualifier*
∧ *procedure*.**s-***Qualifier*.*head*.**s-***Package-name*.s-*TOKEN* = “predefined”

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

compute (*procedure*, *values*) $=_{\text{def}}$
if *intype* (*procedure*, *IntegerType*) **then** *computeInteger*(*procedure*, *values*)
elseif *intype* (*procedure*, *BooleanType*) **then** *computeBoolean*(*procedure*, *values*)
elseif *intype* (*procedure*, *CharacterType*) **then** *computeChar*(*procedure*, *values*)
elseif *intype* (*procedure*, *RealType*) **then** *computeReal*(*procedure*, *values*)
elseif *intype* (*procedure*, *DurationType*) **then** *computeDuration*(*procedure*, *values*)
elseif *intype* (*procedure*, *TimeType*) **then** *computeTime*(*procedure*, *values*)
elseif *intype* (*procedure*, *Stringtype*) **then** *computeString*(*procedure*, *values*)
elseif *intype* (*procedure*, *Arraytype*) **then** *computeArray*(*procedure*, *values*)
elseif *intype* (*procedure*, *Powersettype*) **then** *computePowerset*(*procedure*, *values*)
elseif *intype* (*procedure*, *Bagtype*) **then** *computeBag*(*procedure*, *values*)
else
undefined

endif

computeSideEffects(*procedure*, *values*, *state*, *id*) =def
if isstructureoperator(*procedure*) **then**
structComputeObjects(*procedure*, *values*, *id*)
else
objectvalues

To determine whether a given signature refers to a predefined operator, the following predicate can be used:

matchingoperator: *PROCEDURE* × *NAME* × *IDENTIFIER** → *BOOLEAN*

To determine whether the spelling of a name ends in a certain sequence of characters, the function *compareTail* can be used:

compareTail(*t1*: *TOKEN*, *t2*: *TOKEN*): *TOKEN* $=_{\text{def}}$
if *t1.length* < *t2.length* **then**
False
elseif *t1.length* > *t2.length* **then**
compareTail(*t1.tail*, *t2*)
else
t1 = *t2*
endif

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

definingSort(*p*: *PROCEDURE*): *Identifier* $=_{\text{def}}$
p.parentAS1.nodeAS1ToId

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

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

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*("predefined")>, "Boolean")
IntegerType: *Identifier* $=_{\text{def}}$ **mk-***Identifier*(<**mk-***Package-qualifier*("predefined")>, "Integer")
CharacterType: *Identifier* $=_{\text{def}}$ **mk-***Identifier*(<**mk-***Package-qualifier*("predefined")>, "Character")
RealType: *Identifier* $=_{\text{def}}$ **mk-***Identifier*(<**mk-***Package-qualifier*("predefined")>, "Real")
TimeType: *Identifier* $=_{\text{def}}$ **mk-***Identifier*(<**mk-***Package-qualifier*("predefined")>, "Time")
DurationType: *Identifier* $=_{\text{def}}$ **mk-***Identifier*(<**mk-***Package-qualifier*("predefined")>, "Duration")
Stringtype: *Identifier* $=_{\text{def}}$ **mk-***Identifier*(<**mk-***Package-qualifier*("predefined")>, "String")
Arraytype: *Identifier* $=_{\text{def}}$ **mk-***Identifier*(<**mk-***Package-qualifier*("predefined")>, "Array")
Powersettype: *Identifier* $=_{\text{def}}$ **mk-***Identifier*(<**mk-***Package-qualifier*("predefined")>, "Powerset")
Bagtype: *Identifier* $=_{\text{def}}$ **mk-***Identifier*(<**mk-***Package-qualifier*("predefined")>, "Bag")

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

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

To raise an exception, the function *raise* can be used. The predefined exceptions are distinguished from normal return values as they are *Identifier*s.

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

The following operation signatures are also well-known:

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-***Identifier*(<**mk-***Package-qualifier*("predefined"),
mk-*Data-type-qualifier*("Boolean") >, "True"), *BooleanType*, *undefined*)

3.1.2 Boolean

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

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

computeBoolean(procedure: *PROCEDURE*, *values*: *VALUE**): *VALUE* $=_{\text{def}}$
let *restype* = *definingSort*(*procedure*) **in**
if *matchingoperator*(*procedure*, “not”, *BooleanType*) **then**
mk-*SDLBOOLEAN* (¬ *values*.*head*.*semvalue*, *restype*)
elseif *matchingoperator*(*procedure*, “and”, <*BooleanType*, *BooleanType*>) **then**
mk-*SDLBOOLEAN* (*values*.*head*.*semvalue* ∧ *values*.*tail*.*head*.*semvalue*, *restype*)
elseif *matchingoperator*(*procedure*, “or”, <*BooleanType*, *BooleanType*>) **then**
mk-*SDLBOOLEAN* (*values*.*head*.*semvalue* ∨ *value*.*tail*.*head*.*semvalue*, *restype*)
elseif *matchingoperator*(*procedure*, “xor”, <*BooleanType*, *BooleanType*>) **then**
mk-*SDLBOOLEAN* (¬ (*values*. *head*.*semvalue* ⇔ *values*.*tail*.*head*.*semvalue*), *restype*)
elseif *matchingoperator*(*procedure*, “=>”, <*BooleanType*, *BooleanType*>) **then**
mk-*SDLBOOLEAN* (*values*.*head*.*semvalue* => *values*.*tail*.*head*.*semvalue*, *restype*)
endlet

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

3.1.3 Integer

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

computeInteger (*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUE* $=_{\text{def}}$
let *restype* = *definingSort*(*procedure*) **in**
if *procedure* ∈ *Literal* **then**
integerLiteral(0,*procedure*.*spelling*, *restype*)
elseif *procedure*.**s-***Name* = “-” ∧ *values*.*length* = 1 **then**
mk-*SDLINTEGER* (0 - *values*.*head*.**s-***NAT*, *restype*)
elseif *procedure*. ∈ {“+”, “-”, “*”, “/”, “mod”, “rem”, “<”, “>” , “<=”, “>=”, “power”} **then**
let *val1* = *values*[1].**s-***NAT*, *val2* = *values*[2].**s-***NAT* **in**
case *procedure*.*procName* **in**
| “+” : **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”:
                if val2 = 0 then
                    raise(DivisionByZero)
                else
                    mk-SDLInteger (intPower(val1,val2), restype)
                endif
            | “<” : mk-SDLBool(val1 < val2, restype)
            | “<=” : mk-SDLBool(val1 <= val2, restype)
            | “>” : mk-SDLBool(val1 > val2, restype)
            | “>=”: mk-SDLBool(val1 >= val2, restype)
            endcase
        endlet
    endif
```

The function numbervalue determines the *Nat* associated with a single character in the range “0” to “9”.

```
numberValue(c:Name): Nat =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: Name, type: Identifier): SDLInteger =def
    if proc = empty then
        mk-SDLInteger (num, type)
    else
        integerLiteral(num*10 + numberValue(proc.head), type)
    endif
```

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

```
intDiv(a: Nat, b: Nat):Nat =def
    if a >= 0 ∧ b > a then                    0
    elseif a >= 0 ∧  b <= a ∧ b > 0 then      1 + intDiv(a-b, b)
    elseif a >= 0 ∧ b < 0 then                - intDiv(a, -b)
    elseif a < 0 ∧ b < 0 then                 intDiv (-a, -b)
    elseif a < 0 ∧ b > 0 then                 - intDiv (-a, b)
```

endif

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

intMod(*a*: *NAT*, *b*: *NAT*):*NAT* $=_{\text{def}}$
if $a >= 0 \wedge b > 0$ **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*)
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 *b* > 0 **then** *a* * *intPower*(*a*, *b*-1)
else *intDiv*(*intPower*(*a*, *b*+1), *b*)
endif

3.1.4 Character

Character values are represented by their name.

SDLCHARACTER $=_{\text{def}}$ *NAME* × *Identifier*

computeChar (*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUE* $=_{\text{def}}$
let *restype* = *definingSort*(*procedure*) **in**
if *procedure* ∈ *Literal* **then**
mk-*SDLCHARACTER*(*procedure*.*procName*, *restype*)
elseif procedure.*procName* = “num” **then**
mk-*SDLCHARACTER*(*charValue*(*values*.*head*.**s-***Name*), *restype*)
elseif procedure.*procName* = “chr” **then**
mk-*SDLCHARACTER*(*charChr*(intvalue(*values*.*head*)),*restype*)
endif
endlet

The function charvalue returns the numeral value of the character.

charValue(*ch*: *Name*): *Name* $=_{\text{def}}$
let *literals* = *CharacterType*.**s-***Literal-signature***-set in**
take({*L*.**s-***Constant-expression* | *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 *literals* = *CharacterType*.**s-***Literal-signature***-set**,
result = **mk-***SDLINTEGER* (*a*, *IntegerType*) **in**
take({*L*.**s-***Literal-name* | *L* ∈ *literals*: *L*.**s-***Result* = *result*})

3.1.5 Real

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

```
SDLREAL =def NAT × NAT × Identifier

computeReal (procedure: PROCEDURE, values: VALUE*): VALUE =def
    let  restype = definingSort(procedure) in
    if procedure ∈ Literal then
        realLiteral(0,1,procedure.spelling, restype)
    elseif procedure.procName = "-" ∧ values.length = 1 then
        mk-SDLREAL(0 - values.head.s-NAT, values.head.s2-NAT)
    elseif proycedures ∈ {"+", "-", "*", "/", "<", ">", "<=", ">="} then
        let num1 = values[1].s-NAT, den1 = values[1].s2-NAT,
            num2 = values[2].s-NAT, den2 = values[2].s2-NAT in
            case procedure.procName in
            |  "+" => 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-SDLBOOL(num1*den2 < num2*den1, restype)
            | "<=" => mk-SDLBOOL(num1*den2 <= num2*den1, restype)
            | ">" => mk-SDLBOOL(num1*den2 >= num2*den1, restype)
            | ">="=> mk-SDLBOOL(num1*den2 >= num2*den1, restype)
            endcase
        endlet
    elseif procedure = "float" then
        mk-SDLREAL(intvalue(values.head), 1, restype)
    elseif procedure = "fix" then
        mk-SDLInt(computeFix(values.head.s-NAT, values.head.s2-NAT), IntegerType)
```

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

```
realLiteral(num: NAT, den: NAT, proc: Name, type: Identifier): SDLREAL =def
    if proc = empty then
        mk-SDLREAL(num, den, type)
    elseif proc.head = "." then
        realLiteral(num*10,den*10,proc.tail)
    elseif den = 1 then
        realLiteral(num*10 + numberValue(proc.head), den, type)
    else
        realLiteral(num*10 + numberValue(proc.head), den, type)
```

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
```

3.1.6 Duration

The domain *SDLDURATION* is based on the domain *SDLREAL*.

```
SDLDURATION =def SDLREAL

computeDuration (procedure: PROCEDURE, values: VALUE*): VALUE =def
    computeReal(procedure, values)
```

3.1.7 Time

The domain *SDLTIME* is based on the domain *SDLREAL*.

```
SDLTIME =def SDLREAL

computeTime(procedure: PROCEDURE, values: VALUE*): VALUE =def
    let restype = definingSort(procedure) in
    if procedure ∈ Literal then
        realLiteral(0,1,procedure.spelling, restype)
    case procedure.procName in
        | "time"=> mk-SDLREAL(values.head.s-NAT, values.head.s2-NAT, restype)
        | "<"=> computeReal(procedure, values)
        | "<="=> computeReal(procedure, values)
        | ">"=> computeReal(procedure, values)
        | ">="=> computeReal(procedure, values)
        |  "+"=> computeReal(procedure, values)
        |  "-"=>
            if matchingoperator(procedure, "-", <TimeType, DurationType>) then
                computeReal(procedure, values)
            else
                let res = computeReal(procedure,values) in
                    mk-SDLDURATION(res.s-NAT, res.s2-NAT, restype)
                endlet
    endcase
```

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*): VALUE =def
    let restype = definingSort(procedure) in
    case procedure.procName in
    | "emptystring"=>    mk-SDLSTRING(<>, 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, index = values[2].s-NAT in
            if index < 0 ∨ index > string.length then
                    raise(InvalidIndex)
            else
                    string[index]
    | "modify"=>
            let index = values[2].s-NAT in
            let val = substr(values[1]. s-VALUE-seq, 1, index) ⁀ <values[3]>
                    ⁀ substr(values[1].s-VALUE-seq, index+1,
                        values[1].s-VALUE-seq.length - index) in
                if InvalidIndex in val then raise(InvalidIndex)
                else
                    mk-SDLSTRING(val, restype)
                endif
            endlet endlet
    | "substring"=> let val = substr(values[1], values[2].s-NAT, values[3].s-NAT) in
                        if InvalidIndex in val then raise(InvalidIndex)
```

```
                        else mk-SDLSTRING(val, restype) endif
                    endlet
    | "remove"=>
            let index = values[2].s-NAT in
            let val = substr(values[1] .s-VALUE-seq, 1, index) ⁀
                        substr(values[1].s-VALUE-seq, index+1, values[1].s-VALUE-seq.length - index) in
                if InvalidIndex in val then raise(InvalidIndex) else
                    mk-SDLSTRING(val, restype)
                endif
            endlet endlet
    endcase
```

The function substr computes the substring of a string value.

```
substr(str: VALUE*,start: NAT, len: NAT): VALUE* =def
    if start <= 0 ∨ len <= 0 ∨ start+len-1 > str.length then
        < raise(InvalidIndex) >
    elseif len = 0 then
        <>
    else
        substr(str,start,len) ⁀ <str[start+len-1] >
```

3.1.9 Array

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

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

VALUEPAIR = VALUE × VALUE

computeArray (procedure: PROCEDURE, values: VALUE*): VALUE =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[1], value = values[2] in
            mk-SDLARRAY(modifyArray(a.s-VALUEPAIR-set, index, value), a.s-VALUE, restype)
    elseif name = "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
    endif

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

3.1.10 Powerset

A powerset is represented as a set.

```
SDLPOWERSET =def VALUE-set × Identifier
```

computePowerset (*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUE* $=_{\text{def}}$
let *restype* = *definingSort*(*procedure*) **in**
case *procedure*.*procName* **in**
| "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-***SDLBOOLEAN* (*values*[1].**s-***VALUE***-set** ∩ *values*[2].**s-***VALUE***-set**, *restype*)
| "or"=> **mk-***SDLBOOLEAN* (*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(*Empty*)
else
values[1]. **s-***VALUE***-set**.*take*
endif
endcase

3.1.11 Bag

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

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

FREQUENCY $=_{\text{def}}$ *VALUE* × *NAT*

computeBag (*procedure*: *PROCEDURE*, *values*: *VALUE**): *VALUE* $=_{\text{def}}$
let *restype* = *definingSort*(*procedure*) **in**
case *procedure*.*procName* **in**
| "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*(bagsize(*values*[1].**s-***VALUE***-set**), *IntegerType*)
| "take"=> *values*[1].*take*.**s-***VALUE*
endcase

bagcount(*item*: *VALUE*, *bag*: *SDLBAG*): *NAT* $=_{\text{def}}$
let *elem1* = {*elem*.**s-***NAT* | *elem* ∈ *bag*.**s-***FREQUENCY***-set**: *elem*.**s-***VALUE* = *item* } **in**
if *elem1* = *empty* **then** 0 **else** *elem1*.**s-***NAT* **endif**

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)}

bagdel(*item*: *VALUE*, *bag*: *SDLBAG*): *FREQUENCY***-set** $=_{\text{def}}$
if *bagcount*(*item*, *bag*).**s-***NAT* ≠ 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)}

baginbag(*smaller*: *SDLBAG* , *larger*: *SDLBAG*): *BOOLEAN* $=_{def}$
∀ *elem* ∈ *smaller*.**s-***FREQUENCY***-set**: *bagcount*(*elem*.**s-***VALUE*, *larger*) < *elem*.**s-***NAT*

bagand(*a*: *SDLBAG* ,*b*: *SDLBAG*): *FREQUENCY***-set** $=_{def}$
{ **mk-***FREQUENCY* (*x*.**s-***VALUE* , *min*(*bagcount*(x-*VALUE*,a),*bagcount*(x-*VALUE*,b) |
x ∈ *a*.**s-***FREQUENCY* : *bagcount*(*x*.**s-***VALUE*, *b*) > 0} **where**
min(a,b) = **if** *a*>*b* **then** *a* **else** *b* **endif**

bagor(*a*: *SDLBAG* ,*b*: *SDLBAG*): *FREQUENCY***-set** $=_{def}$
{ **mk-***FREQUENCY*(*x*.**s-***VALUE* , *bagcount*(x.**s-***VALUE*,a) + *bagcount*(x-*VALUE*,b)
| *x* ∈ *a*.**s-***FREQUENCY* }
∪ { *x* | *x* ∈ *b*.**s-***FREQUENCY*: *bagcount*(*x*.**s-***VALUE*, *a*) = 0}

baglength(*a*: *SDLBAG*):*NAT* $=_{def}$
if *a* = *empty* **then** 0
else let *x* = *a*.*take* **in**
x.**s-***NAT* + *baglength*(*a* \ {x})

3.2 Pid Types

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

PID $=_{def}$ (*SDLAGENT* × *Interface-definition*) ∪ { *null* }

static *null*: → *X*

3.3 Constructed Types

3.3.1 Structures

A structure value is identified by its type name, and the field list.

SDLSTRUCTURE $=_{def}$ *FIELD***-set** × *Identifier*
FIELD $=_{def}$ *Name* × *VALUE*

The function *specialName* determines whether a procedure name is one of the implied operator names for a structure type, and returns the field name.

specialName(*procedure:PROCEDURE*, *suffix:TOKEN*): *BOOLEAN* $=_{def}$
if *compareTail*(*procedure*.**s-***Name*.**s-***TOKEN*, *suffix*) **then** *undefined*
else
let *h* = < *procedure*.*procName*[*n*] | *n* in 1.. *procedure*.*procName*.*length* – *suffix*.*length* > **in**
if *h*∈ *fieldNames*(*procedure*.*type*) **then**
h
else
undefined
endif
endlet

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

structExtract(*fieldname:Name*, *fields*: *SDLSTRUCTURE*): *VALUE* $=_{def}$
let *value* = { *f*.**s-**Value | *f* ∈ *fields*: *f*.**s-***Name* = *fieldname*} **in**
if *value* = ∅ **then** *raise*(*UndefinedField*)
else *value*.*take*

```
        endif
```

The function *structModify* returns a new structure with one field changed.

```
    structModify(fieldname: Name, fields: SDLSTRUCTURE, value: VALUE): SDLSTRUCTURE =def
        { f.s-VALUE | f ∈ fields: f.s-Name ≠ fieldname} ∪
        { mk-Field(fieldname, value) }

    computeStruct (procedure:PROCEDURE, values: VALUE*): VALUE =def
        if values.head ∈ OBJECT ∧ values.head = null then raise(InvalidReference)
        elseif specialName(procedure, "Extract") ≠ undefined then
            if values.head ∈ OBJECT then
                structExtract(specialName(procedure, "Extract"), objectValue (state, values.head))
            else
                structExtract(specialName(procedure, "Extract"), values.head.s-FIELD)
            endif
        elseif specialName(procedure, "Modify") ≠ undefined then
            if values.head ∈ OBJECT then
                structModify(specialName(procedure, "Modify"),
                                        objectValue(values.head), values.tail))
            else
                structModify(specialName(procedure, "Modify"), values.head, values.tail.head)
            endif

    structComputeObjects(procedure:PROCEDURE, values:VALUE*, state:STATE):STATE =def
        if values.head ∈ OBJECT ∧ specialName(procedure,"Modify") ∧ ¬ values.head = null then
            modifyObject(values.head, structcompute(procedure, values), state)
        else
            objectvalues
```

3.3.2 Literals

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

```
    SDLLITERALS =def Literal-signature × Identifier

    computeLiteral(procedure:PROCEDURE, values:VALUE*): VALUE =def
        let restype = definingSort(procedure) in
        if procedure.procName ∈ { "<", ">","<=",">=" } then
            let v1 = values.head.s-Literal-signature.s-Result in
            let v2 = values.head.s-Literal-signature.s-Result in
            case procedure.procName in
            | ">" => mk-SDLBOOLEAN (v1 > v2, BooleanType)
            | ">=" => mk-SDLBOOLEAN (v1 >= v2, BooleanType)
            | "<" => mk-SDLBOOLEAN (v1 < v2, BooleanType)
            | "<=" => mk-SDLBOOLEAN (v1 <= v2, BooleanType)
            endcase
            endlet endlet
        elseif procedure.procName = "first" then
            literalMinimum (procedure.s-Result.idToNodeAS1.s-Literal-signature-set)
        elseif procedure.procName = "first" then
            literalMaximum (procedure.s-Result.idToNodeAS1.s-Literal-signature-set)
        elsif procedure. procName = "succ" then
            literalSucc(procedure.s-Result.idToNodeAS1.s-Literal-signature-set, values.first)
        elsif procedure. procName = "pred" then
            literalPred(procedure.s-Result.idToNodeAS1.s-Literal-signature-se, values.first)
        elsif procedure. procName = "num" then
            values.head.s-Literal-signature.s-Result
        endif
```

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

literalMaximum(s: *Literal-signature*-**set**): *Constant-expression* $=_{\text{def}}$
take({*s1*.**s-***Constant-expression*
| *s1* ∈ *s*: ∀ *s2*∈*s*:*s2*.**s-***Constant-expression* > *s1*.**s-***Constant-expression*})

literalSucc(s: *Literal-signature*-**set,** *val: Literal-signature*): *Constant-expression* $=_{\text{def}}$
take({*s1*.**s-***Constant-expression*
| *s1* ∈ *s*: *s1*.**s-***Constant-expression* = val.s-*Constant-expression.semvalue* + 1})

literalPred(s: *Literal-signature*-**set,** *val: Literal-signature*): *Constant-expression* $=_{\text{def}}$
take({*s1*.**s-***Constant-expression*
| *s1* ∈ *s*: *s1*.**s-***Constant-expression* = val.s-*Constant-expression.semvalue* - 1})

3.4 Object Types

Values of object types are represented with an object identification and a value. When a new object is created, an object identifier is assigned to it. That object identifier denotes the object even when the value of the object changes.

OBJECT $=_{\text{def}}$ *OBJECTIDENTIFIER*

Associations between objects and their values are described by the ObjectValue domain

OBJECTVALUE $=_{\text{def}}$ *OBJECT* × *VALUE*

The function objectValue retrieves the value associated with an object:

objectValue(*state*: *STATE*, *object*: *OBJECT*): *VALUE* $=_{\text{def}}$
{ o.**s-***VALUE* | o ∈ state.**s-**ObjectValue-**set**: o.**s-***OBJECT* = object }.take

The function modifyObject re-associates an object with a different value.

modifyObject(*state:STATE*, *object:OBJECT*, *value:VALUE*): *STATE* $=_{\text{def}}$
mk-*STATE*(*state*.**s-***NAMEDVALUE*-**set**, *state*.**s-**SuperState-**set**,
{ *o* | *o* ∈ *state*.**s-***OBJECTVALUE*-**set**: *o*.**s-***OBJECT* ≠ *object*} ∪
{ **mk-***OBJECTVALUE*(*object*, *value*) })

3.5 State Access

The domain *STATE* consists of sub states (associations of values for a specific *STATEID*), super states (associations between super state and substate), and bindings for the objects. In case a certain variable is bound to an in/out parameter in a sub state, it refers to the variable name in the caller's sub state.

STATE $=_{\text{def}}$ *NAMEDVALUE*-**set** × *SUPERSTATE*-**set** × *OBJECTVALUE*-**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*): *STATE* $=_{\text{def}}$
let *newsub* = initDeclarations(*newid*, *declarations*),
newsuper = **mk-** *SUPERSTATE*(*id*, *newid*) **in**
mk-*STATE*(*state*.**s-***NAMEDVALUE*-**set** ∪ *newsub* , *state*.**s-***SUPERSTATE*-**set** ∪ *newsuper*,
state.**s-***OBJECTVALUE*-**set**)

initProcedureState(*state*: *STATE*, *newid*: *STATEID*, *id*: *STATEID*, *declarations*: *DECLARATION*-**set**,

*values:VALUE**, *variables:* (*VALUE* ∪ *Identifier*)*): *STATE* $=_{\text{def}}$
let *newsub* = *assignValues*(initDeclarations(*newid*, *declarations*), *newid*,*declarations*,
values, *variables*),
newsuper = **mk-** *SUPERSTATE*(*id*, *newid*) **in**
mk-*STATE*(*state*.**s-***NAMEDVALUE***-set** ∪ *newsub* , *state*.**s-**SuperState**-set** ∪ *newsuper*,
state.**s-**ObjectValue**-set**)

initDeclarations(*newid*: *STATEID*, *decls*: *DECLARATION***-set**): *NAMEDVALUE***-set** $=_{\text{def}}$
{ **mk-***NAMEDVALUE*(*newid*, *d*.**s-***Variable-identifier*, *d*.**s-***Constant-expression*)
| *d* ∈ *decls*: *d* ∈ *Variable-definition*} ∪
{ **mk-***NAMEDVALUE*(*newid*, mk-*Identifier*(*d.parentAS1.nodeAS1ToId*.**s-***Qualifier*,
d.**s-***Parameter*.**s-***Variable-name*), *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-definition**) $=_{\text{def}}$
if *values* = *empty* **then**
∅
else
let *varname* = *decls.head*.**s-***Parameter*.**s-***Variable-name* **in**
case *decls.head*.**s-***PARAMETERKIND* **in**
| *in*=> *assignValues*(*setValue*(*namedvalues*, *varname*, *values.head*),
decls.tail, *values.tail*, *variables.tail*)
|*out*=> *assignValues*(*namedvalues*, *decls.tail*, *values.tail*, *variables.tail*)
|*inout*=> *assignValues*(*setValue*(*namedvalues*, *varname*, *variables.head*))
endcase
endlet

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

setValue(*namedvalues*: *NAMEDVALUE***-set**, *id*: *STATEID*, *varname:Identifier*, *value:VALUE*)

:*NAMEDVALUE***-set** $=_{\text{def}}$
{ *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** $=_{\text{def}}$
{ *b*.**s-***BOUNDVALUE* | *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 no value is associated, it must be associated in a caller. If a value is found for that triplet, it is the result. If a variable name is found, the associated value must be found in the caller.

eval(*name:Identifier*, *state:STATE*, *id:STATEID*): *VALUE* $=_{\text{def}}$
let *val* = *getValue*(*state*.**s-***NAMEDVALUE***-set**, *id*, *name*) **in**
if *val* = *empty* **then**
value(*name*, *state*, *caller*(*state*, *id*))
elseif *val.take* ∈ *VALUE* **then** *val.take*
else
value(*val*, *state*, *caller*(*state*, *id*))

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

update(*name:Identifier*, *value:VALUE*, *state:STATE*, *id:STATEID*): *STATE* $=_{\text{def}}$
let *val* = *getValue*(*state*.**s-***NAMEDVALUE***-set**, *id*, *name*) **in**
if *val* = *empty* **then**
update(*name*, *value*, *state*, *caller*(*state*, *id*))

```
        elseif val.take ∈ VALUE then
            mk-STATE(setValue(state.s-NAMEDVALUE-set, id, name, value),
                        state.s-SUPERSTATE-set, state.s-OBJECTVALUE-set)
        else
            update(val.take, value, state, id)
```

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): STATE =def
    if isObjectVariable(variablename) then
        if value ∈ OBJECT then
            if isCompatibleTo1(variable.variableSort, value.valuesort) then
                update(variablename, value, state, id)
            else
                update(variablename, variable.variabletype.nullvalue, state, id)
        endif
    else
        if isValueVariable(variablename) then
            if value ∈ OBJECT then
                if value = null then raise(InvalidReference)
                else update(variablename, value.value, state, id)
            else
                update(variablename, value, state, id)
        else
            // Pid variable
            if compatiblepid(variable.variablesort, value.interface) then
                update(variablename, value, state, id)
            else
                update(variablename, null, state, id)
```

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: s.s2-STATEID = id})
```

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

```
variableSort(variablename: Variable-identifier): Data-type-definition =def
    variablename.idToNodeAS1.s-Sort-reference-identifier.idToNodeAS1
```

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

```
isValueVariable(variablename: Variable-identifier): BOOLEAN =def
    variablename.variableSort ∈ Value-data-type-definition
```

The predicate *isObjectVariable* holds if the *variablename* refers to a variable of a object type.

```
isObjectVariable(variablename: Variable-identifier): BOOLEAN =def
    variablename.variableSort ∈ Object-data-type-definition
```

The predicates *assignCopies* and *assignClones* determine whether an assignment will invoke assign or clone, respectively.

```
assignCopies(variablename: Variable-identifier, value: VALUE): BOOLEAN =def
    isValueVariable(variablename)

assignClones(variablename: Variable-identifier, value: VALUE): BOOLEAN =def
    isObjectVariable(variablename) ∧ value.type ∈ Value-data-type-definition
```

3.6 Specialisation

The function *dynamicType* determines the dynamic type of a value.

```
dynamicType(v: VALUE): Identifier =def
```

case v **in**
| *SDLBOOLEAN* (*,*t*) => *t*
| *SDLINTEGER* (*, *t*) => *t*
| *SDLCHARACTER*(*, *t*) => *t*
| *SDLREAL*(*,*, *t*) => *t*
| *SDLSTRING*(*,*t*) => *t*
| *SDLLITERALS*(*,*t*) => *t*
| *SDLSTRUCTURE*(*,*t*) => *t*
| *PID*(*, *t*) => *t*
| *null* => *undefined*
endcase

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* $=_{\text{def}}$
if *procedure* ∈ *Static-operation-signature* **then**
procedure.**s-***Procedure-identifier*
elseif ¬ *allVirtualArgsSet*(*procedure*.**s-***Formal-argument***-seq**, *values*) **then**
raise(*InvalidReference*)
else
let *c* = *allDynamicCandidates*(*procedure*) **in**
let *c1* = *matchingCandidates*(*c*, *values*) **in**
bestMatch(*c1*)
endlet
endlet
endif

The function *allVirtualArgsSet* determines whether there are any null arguments in place of a virtual formal parameter.

allVirtualArgsSet(*args:Formal-argument**, *values:VALUE**): *BOOLEAN* $=_{\text{def}}$
if *args* = *empty* **then**
True
elseif *args*.*head* ∈ *Nonvirtual-argument* **then**
allVirtualArgsSet(*args*.*tail*, *values*.*tail*)
else
¬ *values*.*head*.isnil ∧ *allVirtualArgsSet*(*args*.*tail*, *values*.*tail*)
endif

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

allDynamicCandidates(*procedure:PROCEDURE*): *BOOLEAN* $=_{\text{def}}$
{ *p* | *p* ∈ Dynamic-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*, *dynamicTypes*(*values*))

The function *matchingCandidates* returns the most specialized signature.

bestMatch(*procedures:PROCEDURE*-**set**): *PROCEDURE* $=_{\text{def}}$
take({ *p* | *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$_1$(*p*.head.**s-**Argument, *q*.head.**s-**Argument) ∧
isSignatureCompatible(*p*.tail, *q*.tail)
endif

dynamicTypes(*values:VALUE**): *Identifier** $=_{\mathrm{def}}$
<*dynamicType*(*v*) | *v* **in** *values* >

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* $=_{\mathrm{def}}$
∃ *cond* ∈ *syntype*.**s-**Range-condition:
conditionItemCheck(*cond*, *value*, *syntype*.**s-**Parent-sort-identifier)

The predicate *conditionItemCheck* holds if the condition is true for the value of the given type.

conditionItemCheck(*item*: *Condition-item*, *value: VALUE* , *type: Identifier*): *BOOLEAN* $=_{\mathrm{def}}$
case *item* **of**
| *Open-range*(*opname*, *expr*): *eval*(*rangeOpSignature*(*opname*,*type*), <*value*, *type*>)
| *Closed-range*(*o1*, *o2*): *conditionItemCheck*(*o1*, *value*, *type*) ∧
conditionItemCheck (*o1*, *value*, *type*)
endcase

The function *rangeOpSignature* returns an binary operation signature given an operation name and a parameter type.

rangeOpSignature(*opname:Name*, *type:Identifier*): *Static-operation-signature* $=_{\mathrm{def}}$
mk-*Static-operation-signature* (*opname*,
<**mk-***Nonvirtual-argument*(*type*), **mk-***Nonvirtual-argument*(*type*)>, *BooleanType*)

4 Example

4.1 SDL Example Specification

In Figure F3-16, a system consisting of single block Block1 and a delaying channel segment C1 is specified in SDL. Block1 is sub structured into a process set and a channel segment C11, where the process set contains two instances of Process1 when the system starts execution. This specification will serve as a running example in the remainder of this Annex; in particular, it will be used to illustrate the various steps from an SDL specification to the formal ASM semantics.

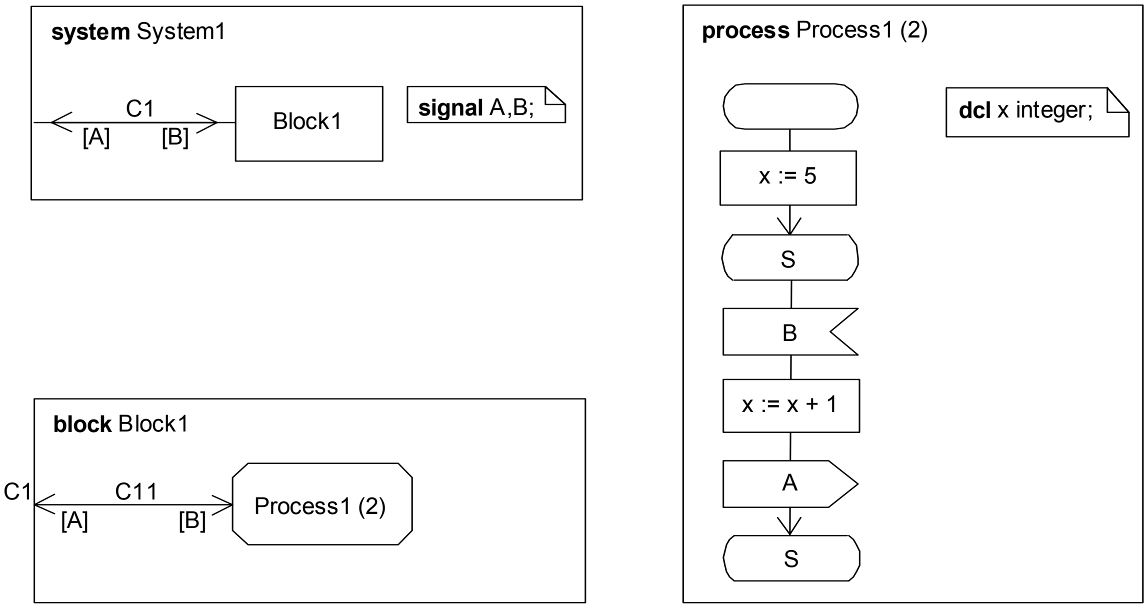


Figure F3-16/Z.100 – Example of an SDL System Specification

4.2 AST of the Example Specification

The AST of the SDL specification shown in Figure F3-16 is defined in several steps as illustrated in Figures F3-17 to F3-20. In order to keep the presentation simple, we used a special dot-notation for identifiers instead of presenting the whole abstract syntax of them. There are no abstract syntax tree nodes associated to the non terminals with "**-set**" and with "*". Instead, the items that belong to their set or sequence are listed.

SDL-specification
    Package-definition
        Package-name
            Impl_p_1
        Agent-type-definition
            Agent-type-name
                Impl_at_1
            Agent-kind
                SYSTEM
            Signal-definition
                A
            Signal-definition
                B
            (include AST of Block1 here)
            Gate-definition
                Gate-name
                    Impl_g_1
                In-signal-identifier
                    Impl_p_1.Impl_at_1.B
                Out-signal-identifier
                    Impl_p_1.Impl_at_1.A
            Channel-definition
                Channel-name
                    C1
                Channel-path
                    Originating-gate
                        Gate-identifier
                            Impl_p_1.Impl_at_1.Impl_g_1
                    Destination-gate
                        Gate-identifier
                            Impl_p_1.Impl_at_1.Impl_at_2.Impl_g_2
                    Signal-identifier
                        Impl_p_1.Impl_at_1.B
                Channel-path
                    Originating-gate
                        Gate-identifier
                            Impl_p_1.Impl_at_1.Impl_at_2.Impl_g_2
                    Destination-gate
                        Gate-identifier
                            Impl_p_1.Impl_at_1.Impl_g_1
                    Signal-identifier
                        Impl_p_1.Impl_at_1.A
    Agent-definition
        Agent-name
            System1
        Number-of-instances
            Initial-number
                1
            Maximum-number
                1
        Agent-type-identifier
            Impl_p_1.Impl_at_1

Figure F3-17/Z.100 – Abstract Syntax Tree of the SDL Specification of System1

Agent-type-definition
  Agent-type-name
    Impl_at_2
  Agent-kind
    BLOCK
  (include AST of Process1 here)
  Gate-definition
    Gate-name
      Impl_g_2
    In-signal-identifier
      Impl_p_1.Impl_at_1.B
    Out-signal-identifier
      Impl_p_1.Impl_at_1.A
  Channel-definition
    Channel-name
      C11
    Channel-path
      Originating-gate
        Gate-identifier
          Impl_p_1.Impl_at_1.Impl_at_2.Impl_g_2
      Destination-gate
        Gate-identifier
          Impl_p_1.Impl_at_1.Impl_at_2.Impl_at_3.Impl_g_3
      Signal-identifier
        Impl_p_1.Impl_at_1.B
    Channel-path
      Originating-gate
        Gate-identifier
          Impl_p_1.Impl_at_1.Impl_at_2.Impl_at_3.Impl_g_3
      Destination-gate
        Gate-identifier
          Impl_p_1.Impl_at_1.Impl_at_2.Impl_g_2
      Signal-identifier
        Impl_p_1.Impl_at_1.A
Agent-definition
  Agent-name
    Block1
  Number-of-instances
    Initial-number
      1
    Maximum-number
      1
  Agent-type-identifier
    Impl_p_1.Impl_at_1.Impl_at_2

Figure F3-18/Z.100 – Abstract Syntax Tree of the SDL Specification of Block1

Agent-type-definition
    Agent-type-name
        Impl_at_3
    Agent-kind
        PROCESS
    Variable-definition
        Variable-name
            x
        Data-type-identifier
            Predefined.integer
    Gate-definition
        Gate-name
            Impl_g_3
        In-signal-identifier
            Impl_p_1.Impl_at_1.B
        Out-signal-identifier
            Impl_p_1.Impl_at_1.A
    State-transition-graph
        State-start-node
            Transition
                Graph-node
                    Task-node
                        Assignment
                            Variable-identifier
                                Impl_p_1.Impl_at_1.Impl_at_2.Impl_at_3.x
                            Expression
                                Constant-expression
                                    Operation-application
                                        Operation-identifier
                                            Predefined.Integer.5
                Terminator
                    Nextstate-node
                        State-name
                            S
        (include AST of state node S here)
Agent-definition
    Agent-name
        Process1
    Number-of-instances
        Initial-number
            2
    Agent-type-identifier
        Impl_p_1.Impl_at_1.Impl_at_2.Impl_at_3

Figure F3-19/Z.100 – Abstract Syntax Tree of the SDL Specification of Process1

```
State-node
    State-name
        S
    Input-node
        Signal-identifier
            Impl_p_1.Impl_at_1.B
        Transition
            Graph-node
                Task-node
                    Assignment
                        Variable-identifier
                            Impl_p_1.Impl_at_1.Impl_at_2.Impl_at_3.x
                        Expression
                            Active-expression
                                Operation-application
                                    Operation-identifier
                                        Predefined.Integer.+
                                    Expression
                                        Active-expression
                                            Variable-access
                                                Variable-identifier
                                                    Impl_p_1.Impl_at_1.Impl_at_2.Impl_at_3.x
                                    Expression
                                        Constant-expression
                                            Operation-application
                                                Operation-identifier
                                                    Predefined.Integer.1
            Graph-node
                Output-node
                    Signal-identifier
                        Impl_p_1.Impl_at_1.A
            Terminator
                Nextstate-node
                    State-name
                        S
```

Figure F3-20/Z.100 – Abstract Syntax Tree of the SDL Specification of State Node S

4.3 Initialisation of the Example

The initialisation of the example proceeds as described in 2.2, the necessary pre-initial state is described in 2.3.1.1. The AS1 representation of the specification as shown in 4.2 is also part of the pre-initial state.

TBD: some figures showing the initialisation process.

4.4 Compilation of the Example

To exemplify the compilation function, consider the labelling of process graph nodes of the SDL example process named *Process1* as illustrated in Figure F3-21. The labels shown represent the function *uniqueLabel*, the unique labels 2 and 3 are represented by adding a hyphen and a number, as in $l_{1\text{-}2}$. Sub-labels, that appear during the compilation of expressions are represented in the form $l_{1(2)}$. The labels shown do only represent the unique labels given by the unique labelling functions. For the presentation here it is enough to know which labels are equal and which are not. This is decided by the name of the labels. Based on this labelling, an ASM model of the SDL example is obtained according to the compilation function of 2.2.

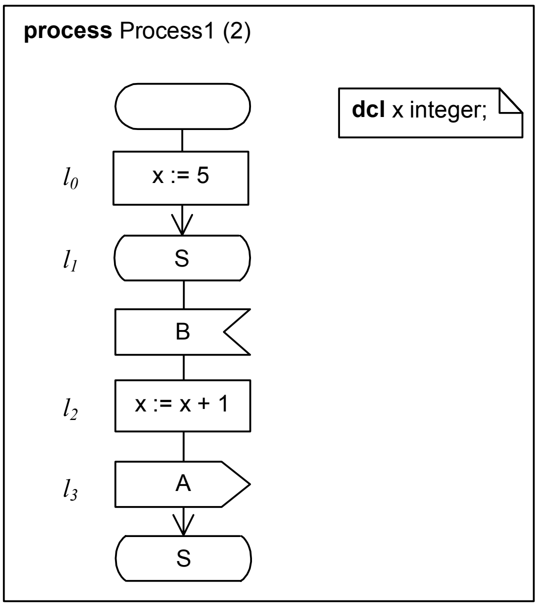


Figure F3-21/Z.100 – Labelling of control-flow graph nodes

TBD: The result of the compilation.

APPENDIX I

Collected abstract syntax

Name	::	*TOKEN*
Package-name	=	*Name*
Agent-type-name	=	*Name*
Agent-name	=	*Name*
State-type-name	=	*Name*
State-name	=	*Name*
Data-type-name	=	*Name*
Procedure-name	=	*Name*
Signal-name	=	*Name*
Interface-name	=	*Name*
Literal-name	=	*Name*
Operation-name	=	*Name*
Syntype-name	=	*Name*
Timer-name	=	*Name*
Gate-name	=	*Name*
Exception-name	=	*Name*
Exception-handler-name	=	*Name*
Connector-name	=	*Name*
State-entry-point-name	=	*Name*
State-exit-point-name	=	*Name*
Channel-name	=	*Name*
Variable-name	=	*Name*
Identifier	::	*Qualifier Name*
Qualifier	=	*Path-item+*
Agent-identifier	=	*Identifier*
Agent-type-identifier	=	*Identifier*
Procedure-identifier	=	*Identifier*
Signal-identifier	=	*Identifier*
Data-type-identifier	=	*Identifier*
Sort-reference-identifier	=	*Sort-identifier*
	|	*Syntype-identifier*
	|	*Expanded-sort-identifier*
	|	*Reference-sort-identifier*
Sort-identifier	=	*Identifier*
Syntype-identifier	=	*Identifier*
Expanded-sort-identifier	=	*Sort-identifier*
Reference-sort-identifier	=	*Sort-identifier*
Timer-identifier	=	*Identifier*
Gate-identifier	=	*Identifier*
Exception-identifier	=	*Identifier*
Composite-state-type-identifier	=	*Identifier*
Channel-identifier	=	*Identifier*
Literal-identifier	=	*Identifier*
Operation-identifier	=	*Identifier*
Variable-identifier	=	*Identifier*
Path-item	=	*Package-qualifier*
	|	*Agent-type-qualifier*
	|	*Agent-qualifier*

	|	*State-type-qualifier*
	|	*State-qualifier*
	|	*Data-type-qualifier*
	|	*Procedure-qualifier*
	|	*Signal-qualifier*
	|	*Interface-qualifier*
Package-qualifier	::	*Package-name*
Agent-type-qualifier	::	*Agent-type-name*
Agent-qualifier	::	*Agent-name*
State-type-qualifier	::	*State-type-name*
State-qualifier	::	*State-name*
Data-type-qualifier	::	*Data-type-name*
Procedure-qualifier	::	*Procedure-name*
Signal-qualifier	::	*Signal-name*
Interface-qualifier	::	*Interface-name*
Informal-text	::	...
SDL-specification	::	[*Agent-definition*] *Package-definition*-**set**
Package-definition	::	*Package-name*
		Package-definition-**set**
		Data-type-definition-**set**
		Syntype-definition-**set**
		Signal-definition-**set**
		Exception-definition-**set**
		Agent-type-definition-**set**
		Composite-state-type-definition-**set**
		Procedure-definition-**set**
Agent-type-definition	::	*Agent-type-name*
		Agent-kind
		[*Agent-type-identifier*]
		*Agent-formal-parameter**
		Data-type-definition-**set**
		Syntype-definition-**set**
		Signal-definition-**set**
		Timer-definition-**set**
		Exception-definition-**set**
		Variable-definition-**set**
		Agent-type-definition-**set**
		Composite-state-type-definition-**set**
		Procedure-definition-**set**
		Agent-definition-**set**
		Gate-definition-**set**
		Channel-definition-**set**
		[*State-machine-definition*]
Agent-kind	=	**SYSTEM** | **BLOCK** | **PROCESS**
Agent-definition	::	*Agent-name*
		Number-of-instances
		Agent-type-identifier
Number-of-instances	::	*Initial-number*
		[*Maximum-number*]
Initial-number	=	*NAT*
Maximum-number	=	*NAT*
Agent-formal-parameter	=	*Parameter*
Parameter	::	*Variable-name Sort-reference-identifier*

Variable-definition	::	*Variable-name* *Sort-reference-identifier* [*Constant-expression*]
Signal-definition	::	*Signal-name Sort-reference-identifier**
Gate-definition	::	*Gate-name* *In-signal-identifier*-**set** *Out-signal-identifier*-**set**
In-signal-identifier	=	*Signal-identifier*
Out-signal-identifier	=	*Signal-identifier*
Channel-definition	::	*Channel-name* [**NODELAY**] *Channel-path*-**set**
Channel-path	::	*Originating-gate* *Destination-gate* *Signal-identifier*-**set**
Originating-gate	=	*Gate-identifier*
Destination-gate	=	*Gate-identifier*
Timer-definition	::	*Timer-name Sort-reference-identifier**
Exception-definition	::	*Exception-name* *Sort-reference-identifier**
Exception-handler-node	::	*Exception-handler-name* [*On-exception*] *Handle-node*-**set** [*Else-handle-node*]
On-exception	::	*Exception-handler-name*
Handle-node	::	*Exception-identifier* [*Variable-identifier*]* [*On-exception*] *Transition*
Else-handle-node	::	[*On-exception*] *Transition*
Procedure-definition	::	*Procedure-name* *Procedure-formal-parameter** [*Result*] [*Procedure-identifier*] *Data-type-definition*-**set** *Syntype-definition*-**set** *Variable-definition*-**set** *Composite-state-type-definition*-**set** *Procedure-definition*-**set** *Procedure-graph*
Procedure-formal-parameter	=	*In-parameter*
	|	*Inout-parameter*
	|	*Out-parameter*
In-parameter	::	*Parameter*
Inout-parameter	::	*Parameter*
Out-parameter	::	*Parameter*

Procedure-graph	::	[*On-exception*] [*Procedure-start-node*] *State-node*-**set** *Free-action*-**set** *Exception-handler-node*-**set**
Result	::	*Sort-reference-identifier*
State-machine-definition	::	*State-name Composite-state-type-identifier*
State-transition-graph	::	[*On-exception*] *State-start-node* *State-node*-**set** *Free-action*-**set** *Exception-handler-node*-**set**
State-start-node	::	[*On-exception*] [*State-entry-point-name*] *Transition*
Procedure-start-node	::	[*On-exception*] *Transition*
State-node	::	*State-name* [*On-exception*] *Save-signalset* *Input-node*-**set** *Spontaneous-transition*-**set** *Continuous-signal*-**set** *Connect-node*-**set** [*Composite-state-type-identifier*]
Save-signalset	=	*Signal-identifier*-**set**
Input-node	::	[**PRIORITY**] *Signal-identifier* [*Variable-identifier*]* [*Provided-expression*] [*On-exception*] *Transition*
Provided-expression	=	*Boolean-expression*
Boolean-expression	=	*Expression*
Spontaneous-transition	::	[*On-exception*] [*Provided-expression*] *Transition*
Continuous-signal	::	[*On-exception*] *Continuous-expression* [*Priority-name*] *Transition*
Continuous-expression	=	*Boolean-expression*
Priority-name	=	*NAT*
Free-action	::	*Connector-name Transition*
Transition	::	*Graph-node** { *Terminator* | *Decision-node* }
Graph-node	::	{ *Task-node* | *Output-node* | *Create-request-node* | *Call-node* | *Compound-node* | *Set-node* | *Reset-node* } [*On-exception*]

Task-node	=	*Assignment*
	|	*Assignment-attempt*
	|	*Informal-text*
Assignment	::	*Variable-identifier Expression*
Assignment-attempt	::	*Variable-identifier Expression*
Output-node	::	*Signal-identifier*
		[*Expression*]*
		[*Signal-destination*]
		Direct-via
Signal-destination	=	*Expression*
	|	*Agent-identifier*
	|	**THIS**
Direct-via	=	(*Channel-identifier* | *Gate-identifier*)**-set**
Create-request-node	::	*Agent-identifier*
		[*Expression*]*
Call-node	::	[**THIS**] *Procedure-identifier* [*Expression*]*
Compound-node	::	*Connector-name*
		*Variable-definition***-set**
		[*Exception-handler-node*]
		*Init-graph-node**
		Transition
		*Step-graph-node**
Init-graph-node	=	*Graph-node*
Step-graph-node	=	*Graph-node*
Set-node	::	*Time-expression Timer-identifier Expression**
Time-expression	=	*Expression*
Reset-node	::	*Timer-identifier Expression**
Terminator	::	{ *Nextstate-node*
		| *Stop-node*
		| *Return-node*
		| *Join-node*
		| *Continue-node*
		| *Break-node*
		| *Raise-node*
		}
		[*On-exception*]
Nextstate-node	=	*Named-nextstate* | *Dash-nextstate*
Named-nextstate	::	*State-name* [*Nextstate-parameters*]
Nextstate-parameters	::	[*Expression*]*
		[*State-entry-point-name*]
Dash-nextstate	::	[**HISTORY**]
Stop-node	::	()
Return-node	=	*Action-return-node*
	|	*Value-return-node*
	|	*Named-return-node*
Action-return-node	::	()
Value-return-node	::	*Expression*
Named-return-node	::	*State-exit-point-name*
Join-node	::	*Connector-name*

Break-node	::	*Connector-name*
Continue-node	::	*Connector-name*
Raise-node	::	*Exception-identifier* [*Expression*]*
Decision-node	::	*Decision-question*
		[*On-exception*]
		*Decision-answer***-set**
		[*Else-answer*]
Decision-question	=	*Expression* | *Informal-text*
Decision-answer	::	{ *Range-condition* | *Informal-text* }
		Transition
Else-answer	::	*Transition*
Composite-state-type-definition	::	*State-type-name*
		[*Composite-state-type-identifier*]
		*Composite-state-formal-parameter**
		*State-entry-point-definition***-set**
		*State-exit-point-definition***-set**
		*Gate-definition***-set**
		*Data-type-definition***-set**
		*Syntype-definition***-set**
		*Exception-definition***-set**
		*Composite-state-type-definition***-set**
		*Variable-definition***-set**
		*Procedure-definition***-set**
		[*Composite-state-graph* | *State-aggregation-node*]
Composite-state-formal-parameter	=	*Agent-formal-parameter*
State-entry-point-definition	=	*Name*
State-exit-point-definition	=	*Name*
Connect-node	::	[*State-exit-point-name*] [*On-exception*] *Transition*
Composite-state-graph	::	[*State-transition-graph*]
		[*Entry-procedure-definition*]
		[*Exit-procedure-definition*]
		*Named-start-node***-set**
State-aggregation-node	::	*State-partition**
		[*Entry-procedure-definition*]
		[*Exit-procedure-definition*]
State-partition	::	*Name*
		Composite-state-type-identifier
		*Connection-definition***-set**
Connection-definition	::	*Entry-connection-definition*
	|	*Exit-connection-definition*
Entry-connection-definition	::	*Outer-entry-point Inner-entry-point*
Outer-entry-point	::	{ *State-entry-point-name* | **DEFAULT** }
Inner-entry-point	::	{ *State-entry-point-name* | **DEFAULT** }
Exit-connection-definition	::	*Outer-exit-point Inner-exit-point*
Outer-exit-point	::	{ *State-exit-point-name* | **DEFAULT** }
Inner-exit-point	::	{ *State-exit-point-name* | **DEFAULT** }
Entry-procedure-definition	=	*Procedure-definition*
Exit-procedure-definition	=	*Procedure-definition*

Named-start-node	::	*State-entry-point-name*
		[*On-exception*]
		Transition
Data-type-definition	=	*Value-data-type-definition*
	|	*Object-data-type-definition*
	|	*Interface-definition*
Value-data-type-definition	::	*Sort*
		Data-type-identifier
		Literal-signature-**set**
		Static-operation-signature-**set**
		Dynamic-operation-signature-**set**
Object-data-type-definition	::	*Sort*
		Data-type-identifier
		Literal-signature-**set**
		Static-operation-signature-**set**
		Dynamic-operation-signature-**set**
Interface-definition	::	*Sort*
		*Data-type-identifier**
Sort	=	*Name*
Literal-signature	::	*Literal-name Result* [*Constant-expression*]
Static-operation-signature	=	*Operation-signature*
Dynamic-operation-signature	=	*Operation-signature*
Operation-signature	::	*Operation-name Formal-argument** [*Result*]
Formal-argument	=	*Virtual-argument* | *Nonvirtual-argument*
Virtual-argument	::	*Argument*
Nonvirtual-argument	::	*Argument*
Argument	=	*Sort-reference-identifier*
Syntype-definition	::	*Syntype-name*
		Parent-sort-identifier
		Range-condition
Parent-sort-identifier	=	*Sort-identifier*
Range-condition	::	*Condition-item*-**set**
Condition-item	=	*Open-range*
	|	*Closed-range*
Open-range	::	*Operation-identifier Constant-expression*
Closed-range	::	*Open-range Open-range*
Expression	=	*Constant-expression*
	|	*Active-expression*
Constant-expression	=	*Literal*
	|	*Conditional-expression*
	|	*Equality-expression*
	|	*Operation-application*
	|	*Range-check-expression*
Active-expression	=	*Variable-access*
	|	*Conditional-expression*
	|	*Operation-application*
	|	*Equality-expression*
	|	*Imperative-expression*
	|	*Range-check-expression*
	|	*Value-returning-call-node*
Imperative-expression	=	*Now-expression*
	|	*Pid-expression*

	|	*Timer-active-expression*
	|	*Any-expression*
Literal	::	*Literal-identifier*
Conditional-expression	::	*Boolean-expression*
		Consequence-expression
		Alternative-expression
Consequence-expression	=	*Expression*
Alternative-expression	=	*Expression*
Equality-expression	::	*First-operand Second-operand*
First-operand	=	*Expression*
Second-operand	=	*Expression*
Operation-application	::	*Operation-identifier* [*Expression*]*
Range-check-expression	::	*Range-condition Expression*
Variable-access	=	*Variable-identifier*
Now-expression	::	()
Pid-expression	=	*Self-expression*
	|	*Parent-expression*
	|	*Offspring-expression*
	|	*Sender-expression*
Self-expression	::	()
Parent-expression	::	()
Offspring-expression	::	()
Sender-expression	::	()
Timer-active-expression	::	*Timer-identifier Expression**
Any-expression	::	*Sort-reference-identifier*
Value-returning-call-node	::	[**THIS**] *Procedure-identifier*
		[*Expression*]*

APPENDIX II

Index

II.1 Functions

II.2 Domains

II.3 AS1 Nonterminals

II.4 Macros

II.5 Programs

SERIES OF ITU-T RECOMMENDATIONS

Series A	Organization of the work of ITU-T
Series B	Means of expression: definitions, symbols, classification
Series C	General telecommunication statistics
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	TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits
Series N	Maintenance: international sound programme and television transmission circuits
Series O	Specifications of measuring equipment
Series P	Telephone transmission quality, telephone installations, local line networks
Series Q	Switching and signalling
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 and open system communications
Series Y	Global information infrastructure and Internet protocol aspects
Series Z	**Languages and general software aspects for telecommunication systems**

Geneva, 2001