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

<table>
<thead>
<tr>
<th><strong>FORMAL DESCRIPTION TECHNIQUES (FDT)</strong></th>
<th><strong>Z.100–Z.109</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Specification and Description Language (SDL)</td>
<td></td>
</tr>
<tr>
<td>Application of Formal Description Techniques</td>
<td>Z.110–Z.119</td>
</tr>
<tr>
<td>Message Sequence Chart</td>
<td>Z.120–Z.129</td>
</tr>
<tr>
<td><strong>PROGRAMMING LANGUAGES</strong></td>
<td></td>
</tr>
<tr>
<td>CHILL: The ITU-T programming language</td>
<td>Z.200–Z.209</td>
</tr>
<tr>
<td><strong>MAN-MACHINE LANGUAGE</strong></td>
<td></td>
</tr>
<tr>
<td>General principles</td>
<td>Z.300–Z.309</td>
</tr>
<tr>
<td>Basic syntax and dialogue procedures</td>
<td>Z.310–Z.319</td>
</tr>
<tr>
<td>Extended MML for visual display terminals</td>
<td>Z.320–Z.329</td>
</tr>
<tr>
<td>Specification of the man-machine interface</td>
<td>Z.330–Z.399</td>
</tr>
<tr>
<td><strong>QUALITY OF TELECOMMUNICATION SOFTWARE</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z.400–Z.499</td>
</tr>
<tr>
<td><strong>METHODS FOR VALIDATION AND TESTING</strong></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Z.500–Z.599</td>
</tr>
</tbody>
</table>

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

© ITU 2002

All rights reserved. No part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from ITU.
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 General information</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Overview of the dynamic semantics</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Definitions from Annex F1</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Definitions from Annex F2</td>
<td>3</td>
</tr>
<tr>
<td>2 Behaviour semantics</td>
<td>3</td>
</tr>
<tr>
<td>2.1 SDL Abstract Machine definition</td>
<td>3</td>
</tr>
<tr>
<td>2.1.1 Signal flow model</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 SDL agents</td>
<td>3</td>
</tr>
<tr>
<td>2.1.3 Interface to the Data Type Part</td>
<td>18</td>
</tr>
<tr>
<td>2.1.4 Behaviour primitives</td>
<td>20</td>
</tr>
<tr>
<td>2.1.5 Undefined Behaviour</td>
<td>35</td>
</tr>
<tr>
<td>2.2 Compilation Function</td>
<td>35</td>
</tr>
<tr>
<td>2.2.1 States and Triggers</td>
<td>36</td>
</tr>
<tr>
<td>2.2.2 Terminators</td>
<td>38</td>
</tr>
<tr>
<td>2.2.3 Actions</td>
<td>39</td>
</tr>
<tr>
<td>2.2.4 Start Labels</td>
<td>43</td>
</tr>
<tr>
<td>2.3 SDL Abstract Machine Programs</td>
<td>44</td>
</tr>
<tr>
<td>2.3.1 System Initialisation</td>
<td>44</td>
</tr>
<tr>
<td>2.3.2 System Execution</td>
<td>58</td>
</tr>
<tr>
<td>2.3.3 Interface between Execution and Compilation</td>
<td>79</td>
</tr>
<tr>
<td>3 Data semantics</td>
<td>79</td>
</tr>
<tr>
<td>3.1 Predefined Data</td>
<td>79</td>
</tr>
<tr>
<td>3.1.1 Well-known definitions</td>
<td>80</td>
</tr>
<tr>
<td>3.1.2 Boolean</td>
<td>81</td>
</tr>
<tr>
<td>3.1.3 Integer</td>
<td>81</td>
</tr>
<tr>
<td>3.1.4 Character</td>
<td>83</td>
</tr>
<tr>
<td>3.1.5 Real</td>
<td>83</td>
</tr>
<tr>
<td>3.1.6 Duration</td>
<td>84</td>
</tr>
<tr>
<td>3.1.7 Time</td>
<td>85</td>
</tr>
<tr>
<td>3.1.8 String</td>
<td>85</td>
</tr>
<tr>
<td>3.1.9 Array</td>
<td>86</td>
</tr>
<tr>
<td>3.1.10 Powerset</td>
<td>86</td>
</tr>
<tr>
<td>3.1.11 Bag</td>
<td>87</td>
</tr>
<tr>
<td>3.2 Pid Types</td>
<td>88</td>
</tr>
<tr>
<td>3.3 Constructed Types</td>
<td>88</td>
</tr>
<tr>
<td>3.3.1 Structures</td>
<td>88</td>
</tr>
<tr>
<td>3.3.2 Literals</td>
<td>89</td>
</tr>
<tr>
<td>3.4</td>
<td>Object Types</td>
</tr>
<tr>
<td>3.5</td>
<td>State Access</td>
</tr>
<tr>
<td>3.6</td>
<td>Specialisation</td>
</tr>
<tr>
<td>3.7</td>
<td>Operators and Methods</td>
</tr>
<tr>
<td>3.8</td>
<td>Syntypes</td>
</tr>
<tr>
<td>4</td>
<td>Example</td>
</tr>
<tr>
<td>4.1</td>
<td>SDL Example Specification</td>
</tr>
<tr>
<td>4.2</td>
<td>AST of the Example Specification</td>
</tr>
<tr>
<td>4.3</td>
<td>Initialisation of the Example</td>
</tr>
<tr>
<td>4.4</td>
<td>Compilation of the Example</td>
</tr>
<tr>
<td>APPENDIX I – Collected abstract syntax</td>
<td>101</td>
</tr>
<tr>
<td>APPENDIX II – Index</td>
<td>109</td>
</tr>
<tr>
<td>II.1</td>
<td>Functions</td>
</tr>
<tr>
<td>II.2</td>
<td>Domains</td>
</tr>
<tr>
<td>II.3</td>
<td>AS1 Nonterminals</td>
</tr>
<tr>
<td>II.4</td>
<td>Macros</td>
</tr>
<tr>
<td>II.5</td>
<td>Programs</td>
</tr>
</tbody>
</table>
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.

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`, `parentASI`, `parentAS0`, `parentAS0ofKind`, `parentASIofKind`, `isAncestorAS0`, `isAncestorASI`, `rootNodeASI`, `rootNodeAS0`;
- the operation symbols `*, +, -set, =, ≠, ∧, ∨, ⇒, ⇔, ¬, ∃, ∀, >, ≥, <, ≤, ∅, 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:

\[ \text{idToNodeAS1: Identifier} \rightarrow \text{DefinitionAS1} \]

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

\[ \text{nodeAS1ToId: DefinitionAS1} \rightarrow \text{Identifier} \]

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

\[ \text{PLAINSIGNAL} \] represents the set of signal types as declared by an SDL specification.

\[ \text{PLAINSIGNAL} = \{ \text{sid} \in \text{Identifier: sid.idToNodeAS1} \in \text{Signal-definition} \} \cup \{ \text{NONE} \} \]

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 \[ \text{SIGNAL} \]:

\[ \text{SIGNAL} = \text{PLAINSIGNAL} \cup \text{EXCEPTION} \cup \text{TIMER} \]

Dynamically created plain signal instances (plain signals for short) are elements of a dynamic domain \[ \text{PLAIN SIGNAL INST} \]. Since plain signals can also be created and sent by the environment, this domain is shared.

The domain \[ \text{SIGNAL INST} \] contains all kinds of signal instances (signals for short).
shared domain PLAINSIGNALINST
initially PLAINSIGNALINST = ∅

SIGNALINST =_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 =_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 =_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* =_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 =_def_ Identifier-set

TOARG =_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 = 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* = def < si in g.schedule : (now ≥ si.arrival) >

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

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.

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

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 \( siSeq \).

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

The rule macros \textit{INSERT} and \textit{DELETE} update the schedule of a gate \( g \) by assigning some new signal list to \( g.schedule \).

\[
\text{INSERT}(si: SIGNALINST, t: TIME, g: GATE) \equiv \\
g.schedule := \text{ insert}(si, g.schedule) \\
si.arrival := t
\]

\[
\text{DELETE}(si: SIGNALINST, g: GATE) \equiv \\
g.schedule := \text{ delete}(si, g.schedule) \\
si.arrival := \text{ undefined}
\]

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.

\[
\text{nextSignal}(si: SIGNALINST, siSeq: SIGNALINST*): SIGNALINST* = def \\
\text{if siSeq = empty then } \text{ undefined } \\
\text{elseif siSeq.head = si then } \\
\text{if siSeq.tail = empty then } \text{ undefined } \\
\text{else } siSeq.tail.head} \\
\text{endif} \\
\text{else nextSignal(si, siSeq.tail)} \\
\text{endif}
\]
The function \texttt{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{verbatim}
selectContinuousSignal(tSet: TRANSITION-set, nSet: NAT-set): TRANSITION =
def if \forall t1 \in tSet: t1.s-NAT \in nSet then undefined
else take({ t \in tSet: t.s-NAT \notin nSet \land \forall t1 \in tSet: (t1.s-NAT \notin nSet \Rightarrow t.s-NAT \leq t1.s-NAT)})
endif
\end{verbatim}

\subsection{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 \texttt{LINK}. The elements of \texttt{LINK} are SAM agents, such that their behaviour is defined through \texttt{LINK-PROGRAM}.

\begin{verbatim}
LINK = AGENT
\end{verbatim}

Intuitively, elements of \texttt{LINK} are considered as point-to-point connection primitives for the transport of signals. More specifically, each \texttt{l} of \texttt{LINK} is able to convey certain signal types, as specified by \texttt{l.with}, from an originating gate \texttt{l.from} to a destination gate \texttt{l.to}.

\begin{verbatim}
controlled from: LINK \rightarrow GATE
controlled to: LINK \rightarrow GATE
controlled noDelay: LINK \rightarrow NODELAY
controlled with: LINK \rightarrow SIGNAL-set
\end{verbatim}

\subsection*{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 \texttt{delay} stating the dependency on external conditions and events. In a given SAM state, \texttt{delay} associates finite time intervals from a domain \texttt{DURATION} to the elements of \texttt{LINK}, where the duration of a particular signal delay appears to be chosen non-deterministically.

\begin{verbatim}
static domain DURATION
monitored delay: LINK \rightarrow DURATION
\end{verbatim}

\subsection*{Integrity Constraints}

There are two important integrity constraints on the function \texttt{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 \texttt{l}, the values of now + \texttt{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.

\subsection*{Channel Behaviour}

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

But, how does a link agent \texttt{l} know whether it is applicable to a signal \texttt{si}? Now, this decision does of course depend on the values of \texttt{si.toArg}, \texttt{si.viaArg}, \texttt{si.signalType} and \texttt{l.with}. In other words, \texttt{l} is a legal choice for the transportation of \texttt{si} only, if the following two conditions hold:

1) \texttt{si.signalType} \in \texttt{l.with}; and
2) there exists an applicable path connecting \textit{L.to} to some final destination matching with the address information and the path constraints of \textit{si}.

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

\begin{verbatim}
FORWARDSIGNAL \equiv
  if \mathit{Self}.from.queue \neq \textit{empty} then
    let \textit{si} = \mathit{Self}.from.queue.head in
    if \textit{Applicable} (\textit{si}.signalType, \textit{si}.toArg, \textit{si}.viaArg, \mathit{Self}.from, \mathit{Self}) then
      \textbf{DELETE} (\textit{si}, \mathit{Self}.from)
      \textbf{INSERT} (\textit{si}, new + \mathit{Self}.delay, \mathit{Self}.to)
      \textit{si}.viaArg \explain{=} \textit{si}.viaArg \setminus
      \{ \mathit{Self}.from.nodeAS1.nodeAS1ToId, \mathit{Self}.nodeAS1.nodeAS1ToId \}
    endif
  endlet
endlet
\end{verbatim}

\subsection{2.1.4 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 \textit{Applicable} formally states all conditions that must be satisfied.

\begin{verbatim}
\textit{Applicable} (s; \textit{SIGNAL}, \textit{toArg}; \textit{TOARG}, \textit{viaArg}; \textit{VIAARG}, \textit{g}; \textit{GATE}, \textit{l}; \textit{LINK}): \mathit{BOOLEAN} \explain{=} \textit{def}
\exists \textit{commPath} \in \{ \textit{ISeq} \in \textit{LINK}^{*}: (\textit{ISeq} \neq \textit{empty} \land \textit{Connected} (\textit{ISeq}.head, \textit{ISeq}.tail)) \}\:
\big( \forall \textit{l} \in \textit{commPath}: \textit{s} \in \textit{l}.\textit{with} \land \textit{owner} \neq \textit{undefined} \land
  \textit{commPath}.head.from \explain{=} \textit{g} \land
  \textit{l} \neq \textit{undefined} \land \textit{commPath}.head = \textit{l} \land \textit{true} \land
  \neg \exists \textit{l} \in \textit{LINK}: (\textit{l}.from \explain{=} \textit{commPath}.last.to \land \textit{s} \in \textit{l}.\textit{with}) \land
  // the path is complete
  \textit{viaArg} \subseteq \textit{commPath}.\textit{commPathIds} \land
  \textit{if} \textit{toArg} \in \textit{Agent-identifier} \textit{then}
  \textit{commPath}.last.to.\textit{myAgent}.\textit{nodeAS1}.\textit{nodeAS1ToId} = \textit{toArg} \land \textit{true} \land
  \textit{if} \textit{toArg} \in \textit{PID} \land \textit{toArg} \neq \textit{null} \textit{then}
  \exists \textit{sa} \in \textit{AGENT}: (\textit{sa}.owner = \textit{commPath}.last.to.\textit{myAgent} \land \textit{sa} = \textit{toArg}.s-\textit{AGENT}) \land \textit{true} \land
\end{verbatim}

where

\begin{verbatim}
\textit{Connected}(\textit{l}; \textit{LINK}; \textit{ISeq}; \textit{LINK}^{*}): \mathit{BOOLEAN} \explain{=} \textit{def}
\textbf{if} \textit{ISeq} \textit{empty} \textbf{then} \textit{true} \textbf{else} \textit{l}.\textit{to} \textit{ISeq}.head.from \textbf{then} \textit{Connected}(\textit{ISeq}.head, \textit{ISeq}.tail) \textbf{else} \textit{false} \land \textit{true} \land
\end{verbatim}

\begin{verbatim}
\textit{commPathIds}(\textit{ISeq}; \textit{LINK}^{*}): \textit{Identifier-set} \explain{=} \textit{def}
\big\{ \{ \textit{g}.\textit{nodeAS1}.\textit{nodeAS1ToId} \mid \textit{g} \in \textit{GATE} \lor \exists \textit{l} \in \textit{ISeq}: (\textit{g} \in \textit{l}.from \lor \textit{g} \in \textit{l}.\textit{to}) \} \lor
\{ \textit{l}.\textit{nodeAS1}.\textit{nodeAS1ToId} \mid \textit{l} \in \textit{LINK} \land \exists \textit{lSeq} \land \textit{l}.\textit{nodeAS1} \neq \textit{undefined} \} \big\}
\end{verbatim}

\end{verbatim}

\subsection{2.1.5 Timers}

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

\begin{verbatim}
\textit{TIMER} \explain{=} \textit{def} \{ \textit{tid} \in \textit{Identifier}; \textit{tid}.idToNodeAS1 \in \textit{Timer-definition} \}
\textit{TIMERINST} \explain{=} \textit{def} \textit{PID} \times \textit{TIMER} \times \textit{VALUE}^{*}
\end{verbatim}

The information associated with timers is accessed using the functions defined on \textit{SIGNAL}.
Active Timers

To indicate whether a timer instance \( tmi \) is active or not, there is a corresponding derived predicate \( \text{Active} \):

\[
\text{Active}(tmi: \text{TIMERINST}) : \text{BOOLEAN} = \text{def} \ tmi \in \text{Self.inport.schedule}
\]

Timer Operations

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

\[
\begin{align*}
\text{static duration} : \text{TIMER} &\rightarrow \text{DURATION} \\
\text{SETTIMER}(tm: \text{TIMER}, vSeq : \text{VALUE}\ast, t: \text{TIME}) &\equiv \\
&\text{let } tmi = \text{mk-TIMERINST}(\text{Self.self, tm, vSeq}) \text{ in} \\
&\text{if } t = \text{undefined} \text{ then} \\
&\quad \text{Self.inport.schedule} := \text{insert}(tmi, \text{now} + t.\text{duration}, \text{delete}(tmi, \text{Self.inport.schedule})) \\
&\quad \text{si.arrival} := \text{now} + t.\text{duration} \\
&\text{else} \\
&\quad \text{Self.inport.schedule} := \text{insert}(tmi, t, \text{delete}(tmi, \text{Self.inport.schedule})) \\
&\quad \text{si.arrival} := t \\
&\text{endif} \\
&\text{endlet} \\
\text{RESETTIMER}(tm: \text{TIMER}, vSeq : \text{VALUE}\ast) &\equiv \\
&\text{let } tmi = \text{mk-TIMERINST}(\text{Self.self, tm, vSeq}) \text{ in} \\
&\text{if } \text{Active}(tmi) \text{ then} \\
&\quad \text{DELETE}(tmi, \text{Self.inport}) \\
&\text{endif} \\
&\text{endlet}
\end{align*}
\]

2.1.1.6 Exceptions

Like timers, exceptions are also identified with exception signals.

\[
\begin{align*}
\text{EXCEPTION} &\text{=def} \{ \text{eid} \in \text{Identifier} : \text{eid.idToNodeAS1} \in \text{Exception-definition} \} \\
\text{EXCEPTIONINST} &\text{=def} \text{PID} \times \text{EXCEPTION} \times \text{VALUE}\ast
\end{align*}
\]

The information associated with exceptions is accessed using the functions defined on \( \text{SIGNAL} \).

Exception Handlers

Exception handlers are modelled by state nodes. Every agent keeps track of its active exception handlers with the function \( \text{activeHandler} \). 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.

\[
\begin{align*}
\text{EXCEPTIONSCOPE} &\text{=def} \{ \text{esEntireGraph, esCompositeState, esCompositeStateGraph, esCurrentState, esStimulusOrStart, esExceptionState, esHandleClause, esAction} \} \\
\text{exceptionScopeSeq} &\text{=def} < \text{esAction, esHandleClause, esExceptionState, esStimulusOrStart, esCurrentState, esCompositeStateGraph, esCompositeState, esEntireGraph} >
\end{align*}
\]

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.

\[
\begin{align*}
\text{SETEXCEPTIONHANDLER}(\text{handlerName: Exception-handler-name, scope: EXCEPTIONSCOPE}) &\equiv \\
&\text{activeHandler}(\text{Self,scope}) := \text{handlerName}
\end{align*}
\]
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 =def AGENT

SDLAGENTSET =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 = Ø

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

The first group of declarations and definitions introduces a controlled domain STATENODE, and a number of derived domains.
The second group of declarations introduces controlled functions defined on the domain \texttt{STATE\textsc{node}}, they can be understood as a state node control block and are used to model the state machine by a hierarchical inheritance state graph.

\textbf{controlled currentSubStates}: \texttt{STATE\textsc{node}} $\rightarrow$ \texttt{STATE\textsc{node-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.

\begin{verbatim}
collectCurrentSubStates(snSet: \texttt{STATE\textsc{node-set}}): \texttt{STATE\textsc{node-set}} = \text{def}
  let sn = take(snSet) in
  if sn = undefined then $\emptyset$
  else $\{sn\} \cup collectCurrentSubStates(snSet \setminus \{sn\} \cup sn.currentSubStates)
endlet
\end{verbatim}

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

\textbf{controlled currentExitPoints}: \texttt{STATE\textsc{node}} $\rightarrow$ \texttt{STATE\textsc{exitpoint-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.

\begin{verbatim}
DirectlyInheritsFrom(sn1: \texttt{STATE\textsc{node}}, sn2: \texttt{STATE\textsc{node}}): \texttt{boolean} = \text{def}
  if sn2.parentStateNode $\in$ sn1.parentStateNode.inheritedStateNodes \&
    sn1.stateName = sn2.stateName \&
    ($\neg \exists sn3 \in \texttt{STATE\textsc{node}}: 
    sn3.parentStateNode $\in$ sn1.parentStateNode.inheritedStateNodes \&
    sn2.parentStateNode $\in$ sn3.parentStateNode.inheritedStateNodes \&
    sn3.stateName = sn2.stateName) \text{ then True}
  else False
endif
\end{verbatim}

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

\begin{verbatim}
InheritsFrom(sn1: \texttt{STATE\textsc{node}}, sn2: \texttt{STATE\textsc{node}}): \texttt{boolean} = \text{def}
  sn2.parentStateNode $\in$ sn1.parentStateNode.inheritedStateNodes \&
  sn1.stateName = sn2.stateName
\end{verbatim}

This predicate determines whether two state nodes are inherited.

\begin{verbatim}
DirectlyRefinedBy(sn1: \texttt{STATE\textsc{node}}, sn2: \texttt{STATE\textsc{node}}): \texttt{boolean} = \text{def}
  sn2.parentStateNode = sn1
\end{verbatim}

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

\begin{verbatim}
InheritsFromOrRefinedBy(sn1: \texttt{STATE\textsc{node}}, sn2: \texttt{STATE\textsc{node}}): \texttt{boolean} = \text{def}
  $\exists n \in \texttt{nat}: \text{InheritsFromOrRefinedByStep(sn1, sn2, n)}$
\end{verbatim}

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

\begin{verbatim}
InheritsFromOrRefinedByStep(sn1: \texttt{STATE\textsc{node}}, sn2: \texttt{STATE\textsc{node}}, n: \texttt{nat}): \texttt{boolean} = \text{def}
  if $n = 0$ then False
  elseif $n = 1$ then DirectlyRefinedBy(sn1, sn2) \lor DirectlyInheritsFrom(sn1, sn2)
  else $\exists sn3 \in \texttt{STATE\textsc{node}}:
    \((\text{InheritsFromOrRefinedByStep(sn1, sn3, 1)} \land 
    \text{InheritsFromOrRefinedByStep(sn3, sn2, n-1)})$\end{verbatim}
This predicate determines whether \( sn1 \) inherits from or is refined by \( sn2 \), taking transitivity of this relationship into account.

\[
\text{selectNextStateNode}(snSet: \text{STATENODE-set}) : \text{STATENODE} = \text{def} \\
\text{let } sn = \text{take}(\{sn1 \in snSet: (\neg \exists sn2 \in snSet: \text{InheritsFromOrRefinedBy}(sn1, sn2))\}) \text{ in} \\
\quad \text{if } sn = \text{undefined} \text{ then undefined} \\
\quad \text{elseif } \exists sn1 \in snSet: \text{DirectlyInheritsFrom}(sn1, sn) \lor sn = sn1.\text{inheritedStateNode} \text{ then} \\
\qquad \text{selectNextStateNode}(snSet \setminus \{sn\}) \\
\quad \text{else } sn \\
\text{endif}
\]

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.

\[
\text{inheritedStateNodes}(sn: \text{STATENODE}) : \text{STATE\text{-}NODE-set} = \text{def} \\
\quad \text{if } sn.\text{inheritedStateNode} = \text{undefined} \text{ then } \emptyset \\
\quad \text{else } \{sn.\text{inheritedStateNode}\} \cup sn.\text{inheritedStateNode}.\text{inheritedStateNodes} \\
\text{endif}
\]

This function defines, for a given state node, the set of inherited state nodes.

\[
\text{parentStateNodes}(sn: \text{STATENODE}) : \text{STATENODE-set} = \text{def} \\
\quad \text{if } sn.\text{parentStateNode} = \text{undefined} \text{ then } \emptyset \\
\quad \text{else } \{sn.\text{parentStateNode}\} \cup sn.\text{parentStateNode}.\text{parentStateNodes} \\
\text{endif}
\]

This function defines, for a given state node, the set of inherited state nodes.

\[
\text{mostSpecialisedStateNode}(sn: \text{STATENODE}) : \text{STATENODE} = \text{def} \\
\text{let } sn1 = \text{take}(\{sn2 \in \text{STATENODE}: \text{InheritsFrom}(sn2, sn)\}) \text{ in} \\
\quad \text{if } sn1 = \text{undefined} \text{ then } sn \text{ else } sn1.\text{mostSpecialisedStateNode} \text{ endif}
\]

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.

\[
\text{selectInheritedStateNode}(sn: \text{STATENODE}, snSet: \text{STATE\text{-}NODE-set}) : \text{STATENODE} = \text{def} \\
\text{take}(\{sn1 \in snSet: \text{DirectlyInheritsFrom}(sn, sn1)\})
\]

This function yields a state node that may be left next, provided \( snSet \) is a valid set of state nodes to be left.

\[
\text{getPreviousStatePartition}(sn: \text{STATENODE}) : \text{STATENODE} = \text{def} \\
\quad \text{if } sn.\text{stateNodeKind} = \text{statePartition} \land \\
\qquad \neg \exists sn1 \in sn.\text{parentStateNodes}: sn1.\text{stateNodeKind} = \text{procedureNode} \land \\
\qquad \text{then } sn.\text{mostSpecialisedStateNode} \\
\quad \text{else } \text{getPreviousStatePartition}(sn.\text{parentStateNode}) \\
\text{endif}
\]

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

\[
\text{controlled resultLabel}: \text{STATENODE} \rightarrow \text{LABEL}
\]

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.

\[
\text{controlled callingProcedureNode}: \text{STATENODE} \rightarrow \text{STATENODE}
\]

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: STATEENTRYPOINT × STATENODE → STATEENTRYPOINT
controlled exitConnection: STATEEXITPOINT × STATENODE → 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.
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.

A unary function nodeASI 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.

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

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

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.

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

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.

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.

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.

During the firing of input transitions, the signal instance removed from the input port is available through the function currentSignalInst.
controlled topStateNode: SDLAGENT \rightarrow \textit{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 \rightarrow \textit{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 \rightarrow \textit{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 \rightarrow \textit{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 \rightarrow \textit{EXCEPTIONINST}

The current exception instance is used within exception handling.

controlled scopeName: SDLAGENT \times \textit{STATEID} \rightarrow \textit{Connector-name}
controlled scopeContinueLabel: SDLAGENT \times \textit{STATEID} \rightarrow \textit{CONTINUELABEL}
controlled scopeStepLabel: SDLAGENT \times \textit{STATEID} \rightarrow \textit{STEPLABEL}

These functions are used to execute compound nodes.

\textbf{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 \rightarrow \textit{State-node-set}
controlled statePartitionsToBeCreated: SDLAGENT \rightarrow \textit{State-partition-set}
controlled ehNodesToBeCreated: SDLAGENT \rightarrow \textit{Exception-handler-node-set}
controlled stateNodesToBeRefined: SDLAGENT \rightarrow \textit{STATENODE-set}
controlled stateNodesToBeSpecialised: SDLAGENT \rightarrow \textit{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.

\textbf{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 \rightarrow \textit{SIGNALINST}^*
controlled stateNodesToBeChecked: SDLAGENT \rightarrow \textit{STATENODE-set}
controlled stateNodeChecked: SDLAGENT \rightarrow \textit{STATENODE}
controlled startNodeChecked: SDLAGENT \rightarrow \textit{STATENODEWITHENTRYPOINT}
controlled exitNodeChecked: SDLAGENT \rightarrow \textit{STATENODEWITHEXITPOINT}
controlled transitionsToBeChecked: SDLAGENT \rightarrow \textit{TRANSITION-set}
controlled transitionChecked: SDLAGENT \rightarrow \textit{TRANSITION}
controlled signalChecked: SDLAGENT \rightarrow \textit{SIGNALINST}
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: SDLAGENT ∪ STATENODE → 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: SDLAGENT ∪ STATENODE → STATENODE

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

controlled currentProcedureStateNode: SDLAGENT ∪ STATENODE → STATENODE

Refers to the current procedure state node.

controlled activeHandler: (AGENT ∪ STATENODE) × EXCEPTIONSCOPE → Exception-handler-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.

GateUnconnected(g; GATE): BOOLEAN = def

∀ cd ∈ g.myAgent.nodeAS1.s-Channel-definition-set. ∀ cp ∈ cd.s-Channel-path-set:
(g.nodeAS1 ≠ cp.s-Originating-gate.idToNodeAS1 ∧
g.nodeAS1 ≠ cp.s-Destination-gate.idToNodeAS1)

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

ingates(a; AGENT): GATE-set = def

if a.IsAgentSet then
{ g ∈ GATE: g.myAgent = a ∧ g.direction = inDir ∧ g.GateUnconnected }
else
a.owner.ingates
endif

outgates(a; AGENT): GATE-set = def

if a.IsAgentSet then
{ g ∈ GATE: g.myAgent = a ∧ g.direction = outDir ∧ g.GateUnconnected }
else
a.owner.outgates
endif

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.

TRANSITION = def SIGNAL × LABEL × NAT × LABEL × STATEEXITPOINT

STARTTRANSITION = def LABEL × STATEENTRYPOINT

FREEACTION = def Connector-name × LABEL
Given a set of transitions, several derived functions are defined to select particular subsets:

\[
\text{priorityInputTransitions}(tSet: \text{TRANSITION-set}) : \text{TRANSITION-set} = \{ t \in tSet : t.s\text{-SIGNAL} \neq \text{undefined} \land t.s\text{-LABEL} = \text{undefined} \land t.s\text{-NAT} \neq \text{undefined} \}
\]

\[
\text{inputTransitions}(tSet: \text{TRANSITION-set}) : \text{TRANSITION-set} = \{ t \in tSet : t.s\text{-SIGNAL} \neq \text{undefined} \land t.s\text{-NAT} = \text{undefined} \}
\]

\[
\text{continuousSignalTransitions}(tSet: \text{TRANSITION-set}) : \text{TRANSITION-set} = \{ t \in tSet : t.s\text{-SIGNAL} = \text{undefined} \land t.s\text{-LABEL} \neq \text{undefined} \land t.s\text{-NAT} \neq \text{undefined} \}
\]

\[
\text{spontaneousTransitions}(tSet: \text{TRANSITION-set}) : \text{TRANSITION-set} = \{ t \in tSet : t.s\text{-SIGNAL} = \text{NONE} \}
\]

\[
\text{exitTransitions}(tSet: \text{TRANSITION-set}) : \text{TRANSITION-set} = \{ t \in tSet : t.s\text{-STATEEXITPOINT} \neq \text{undefined} \}
\]

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 \textit{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 \textit{state} defined on process instances.

\textit{derived domain \textit{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 \textit{initAgentState} and \textit{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 \textit{STATEID}. A \textit{STATE} contains associations between a number of \textit{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 \textit{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 \textit{initProcedureState} is

- List of parameter values and variable names.

\textit{controlled domain \textit{STATEID}}

\[
\textit{DECLARATION} = \text{def} \text{Procedure-formal-parameter} \cup \text{Variable-definition}
\]

\[
\textit{PARAMETERKIND} = \text{def} \{ \text{in}, \text{out}, \text{inout} \}
\]

\[
\textit{initAgentState}: \text{STATE} \times \text{STATEID} \times \text{STATEID} \times \text{DECLARATION-set} \rightarrow \text{STATE}
\]

\[
\textit{initProcedureState}: \text{STATE} \times \text{STATEID} \times \text{STATEID} \times \text{DECLARATION-set} \times (\text{VALUE} \cup \text{Identifier})* \rightarrow \text{STATE}
\]

The domain \textit{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 \textit{VALUE}. Moreover, there is a domain for variable names that is used by both parts of the semantics. The domain \textit{VARIABLE} denotes variables independent from their assigned values.

\[
\text{VALUE} = \text{def } SDLI\text{NTEGER} \cup SDL\text{BOOLEAN} \cup SDL\text{REAL} \cup SDL\text{CHARACTER} \cup SDL\text{STRING} \\
\quad \cup PID \cup \text{OBJECT} \cup SDL\text{LITERSALS} \cup SDL\text{STURE} \cup SDL\text{ARRAY} \cup SDL\text{POWERSET} \\
\]

\textbf{derived domain VALUE}

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.

\[
\text{STATEORexception} = \text{STATE} \cup \text{EXCEPTION} \\
\text{VALUEORexception} = \text{VALUE} \cup \text{EXCEPTION}
\]

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

\[
\begin{align*}
\text{assign} &: \text{Variable-identifier} \times \text{VALUE} \times \text{STATE} \times \text{STATEID} \rightarrow \text{STATEORexception} \\
\text{assignClones} &: \text{Variable-identifier} \times \text{VALUE} \rightarrow \text{BOOLEAN} \\
\text{assignCopies} &: \text{Variable-identifier} \times \text{VALUE} \rightarrow \text{BOOLEAN} \\
\text{objectsAssign} &: \text{OBJECTVALUE}^* \times \text{Variable-identifier} \times \text{VALUE} \rightarrow \text{OBJECTVALUE}^*
\end{align*}
\]

The function \textit{eval} retrieves the value associated with a variable for a given state and state id. \textit{assign} associates a new value with a given variable. Since an assignment may involve invocation of an operator copy or assign, there are two predicates \textit{assignClones} and \textit{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, \textit{objectsAssign} returns the new values for the objects. There is a rule macro using this function, which is doing the real assignment.

\[
\begin{align*}
\text{ASSIGN}(\text{variableName}, \text{value}, \text{state}, \text{id}) &= \\
\text{if assignClones(variableName, value) then} & \\
\quad \text{tmpval} := \text{CALLCLONE}(\text{variableName}, \text{value}) \\
\quad \text{state} := \text{assign}(\text{variableName}, \text{tmpval}, \text{state}, \text{id}) \\
\quad \text{SETOBJECTS(Self, objectsAssign(getObjects(Self, variableName, value)))} \\
\text{else} & \\
\quad \text{if assignCopies(variableName, value) then} & \\
\quad \quad \text{tmpval} := \text{CALLCOPY}(\text{variableName}, \text{value}) \\
\quad \quad \text{state} := \text{assign}(\text{variableName}, \text{tmpval}, \text{state}, \text{id}) \\
\quad \quad \text{SETOBJECTS(Self, objectsAssign(getObjects(Self, variableName, value)))} \\
\text{else} & \\
\quad \text{state} := \text{assign}(\text{variableName}, \text{value}, \text{state}, \text{id}) & \\
\quad \text{SETOBJECTS(Self, objectsAssign(getObjects(Self, variableName, value)))}
\end{align*}
\]

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 \textit{getObjects} returns the object values for a given agent; the macro \textit{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.

\[
\text{eval}: \text{Variable-identifier} \times \text{STATE} \times \text{STATEID} \rightarrow \text{VALUE}
\]

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

static dispatch: \( \text{PROCEDURE} \times \text{VALUE}^* \rightarrow \text{Identifier} \).

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: \( \text{PROCEDURE} \times \text{VALUE}^* \rightarrow \text{BOOLEAN} \)

compute: \( \text{PROCEDURE} \times \text{VALUE}^* \rightarrow \text{VALUEorException} \)

computeSideEffects: \( \text{PROCEDURE} \times \text{VALUE}^* \times \text{STATE} \times \text{STATEId} \rightarrow \text{OBJECTVALUE}^* \)

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

```latex
\begin{align*}
\text{derived domain } & \text{SDLBOOLEAN} \\
\text{derived domain } & \text{SDLINTEGER} \\
\text{derived semvalue}: & \text{SDLBOOLEAN} \rightarrow \text{BOOLEAN}
\end{align*}
```

### 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`.

```latex
\begin{align*}
\text{domain } & \text{OBJECTIDENTIFIER} \\
\text{initially } & \text{OBJECTIDENTIFIER} = \{ \text{null} \}
\end{align*}
```

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

```latex
\begin{align*}
\text{mkObjectId}: & \rightarrow \text{OBJECTIDENTIFIER}
\end{align*}
```

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: \( \rightarrow \text{BEHAVIOUR} \)

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.

```latex
\begin{align*}
\text{STARTLABEL} = & \text{def LABEL} \\
\text{BEHAVIOUR} = & \text{def STARTLABEL} \times \text{PRIMITIVE-set} \\
\text{PRIMITIVE} = & \text{def LABEL} \times \text{ACTION}
\end{align*}
```
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.

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

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.

\[ \text{CONTINUELABEL} = \text{def} \ LABEL \]
\[ \text{VALUELABEL} = \text{def} \ LABEL \]

Functions

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

controlled value: VALUELABEL \times SDLAGENT \rightarrow VALUE

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

ITU-T Z.100/Annex F3 (01/2000) 21
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 $\text{Eval}$. Macro $\text{Eval}$ 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 $\text{ACTION}$ are pairwise disjoint.

\[
\text{Eval}(a: \text{ACTION}) \equiv \\
\text{if } a \in \text{VAR} \text{ then } \text{EvalVAR}(a) \\
\text{elseif } a \in \text{OPERATIONAPPLICATION} \text{ then } \text{EvalOPERATIONAPPLICATION}(a) \\
\text{elseif } a \in \text{CALL} \text{ then } \text{EvalCALL}(a) \\
\text{elseif } a \in \text{RETURN} \text{ then } \text{EvalRETURN}(a) \\
\text{elseif } a \in \text{TASK} \text{ then } \text{EvalTASK}(a) \\
\text{elseif } a \in \text{ASSIGNPARAMETERS} \text{ then } \text{EvalASSIGNPARAMETERS}(a) \\
\text{elseif } a \in \text{EQUALITY} \text{ then } \text{EvalEQUALITY}(a) \\
\text{elseif } a \in \text{DECISION} \text{ then } \text{EvalDECISION}(a) \\
\text{elseif } a \in \text{OUTPUT} \text{ then } \text{EvalOUTPUT}(a) \\
\text{elseif } a \in \text{CREATE} \text{ then } \text{EvalCREATE}(a) \\
\text{elseif } a \in \text{SET} \text{ then } \text{EvalSET}(a) \\
\text{elseif } a \in \text{RESET} \text{ then } \text{EvalRESET}(a) \\
\text{elseif } a \in \text{TIMERACTIVE} \text{ then } \text{EvalTIMERACTIVE}(a) \\
\text{elseif } a \in \text{SETHANDLER} \text{ then } \text{EvalSETHANDLER}(a) \\
\text{elseif } a \in \text{RAISE} \text{ then } \text{EvalRAISE}(a) \\
\text{elseif } a \in \text{STOP} \text{ then } \text{EvalSTOP}(a) \\
\text{elseif } a \in \text{SYSTEMVALUE} \text{ then } \text{EvalSYSTEMVALUE}(a) \\
\text{elseif } a \in \text{ANYVALUE} \text{ then } \text{EvalANYVALUE}(a) \\
\text{elseif } a \in \text{SETRANGECHECKVALUE} \text{ then } \text{EvalSETRANGECHECKVALUE}(a) \\
\text{elseif } a \in \text{SCOPE} \text{ then } \text{EvalSCOPE}(a) \\
\text{elseif } a \in \text{SKIP} \text{ then } \text{EvalSKIP}(a) \\
\text{elseif } a \in \text{BREAK} \text{ then } \text{EvalBREAK}(a) \\
\text{elseif } a \in \text{CONTINUE} \text{ then } \text{EvalCONTINUE}(a) \\
\text{elseif } a \in \text{ENTERSTATENODE} \text{ then } \text{EvalENTERSTATENODE}(a) \\
\text{elseif } a \in \text{LEAVESTATENODE} \text{ then } \text{EvalLEAVESTATENODE}(a) \\
\text{endif}
\]
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 \texttt{VAR} is a tuple consisting of a variable name and a so-called continue label. The macro \texttt{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 \texttt{Var} is defined as cartesian product of the domain \texttt{Variable-identifier} of variable names and domain \texttt{CONTINUELABEL} of labels.

\[ \texttt{VAR} = \texttt{def} \texttt{Variable-identifier} \times \texttt{CONTINUELABEL} \]

Behaviour

The value of a variable in the current state of the executing agent is determined by function \texttt{eval} and written at \texttt{Self.currentLabel}. In order to avoid conflicts with other agents, the function value takes a further argument of type \texttt{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.

\[ \texttt{EVALVAR}(a: \texttt{VAR}) \equiv \]
\[ \begin{align*}
\text{value}(\text{Self.currentLabel}, \text{Self}) & := \text{eval}(a.s-\text{Variable-identifier}, \text{Self.stateAgent.state}) \\
\text{Self.currentLabel} & := a.s-\text{CONTINUELABEL}
\end{align*} \]

Reference clauses

For definition of function \texttt{value}, refer to 2.1.4.1. Definition of function \texttt{eval} can be found in 2.1.3.1. Function \texttt{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 \texttt{+}, \texttt{-} on the set of integers are predefined procedures. A predefined procedure will be executed by function \texttt{compute}. A non-functional operation which is handled with function \texttt{dispatch} which determines depending on the current values the correct procedure identifier.

Representation

\[ \texttt{OPERATIONAPPLICATION} = \texttt{def} \texttt{PROCEDURE} \times \texttt{VALUELABEL}^{*} \times \texttt{CONTINUELABEL} \]

Behaviour

\[ \texttt{EVALOPERATIONAPPLICATION}(a: \texttt{OPERATIONAPPLICATION}) \equiv \]
\[ \begin{align*}
\text{if } & \text{functional}(a.s-\text{PROCEDURE}, \text{values}(a.s-\text{VALUELABEL}-\text{seq}, \text{Self})) \text{ then} \\
\text{value}(\text{Self.currentLabel}, \text{Self}) & := \text{compute}(a.s-\text{PROCEDURE}, \text{values}(a.s-\text{VALUELABEL}-\text{seq}, \text{Self})) \\
\text{Self.currentLabel} & := a.s-\text{CONTINUELABEL}
\end{align*} \]
\[ \text{else} \]
\[ \begin{align*}
\text{let } pd & := \text{idToNodeAS1}(\text{dispatch}(a.s-\text{PROCEDURE}, \text{values}(a.s-\text{VALUELABEL}-\text{seq}, \text{Self}))) \text{ in} \\
\text{CREATEPROCEDURE}(pd, a.s-\text{CONTINUELABEL}) \\
\text{endlet}
\end{align*} \]
\[ \text{endif} \]

Reference clauses

For definition of function \texttt{value}, refer to 2.1.4.1. Definition of predicate \texttt{functional} and function \texttt{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.

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

Behaviour
\[
\text{EVALCALL}(a: \text{CALL}) \equiv \\
\text{CREATEPROCEDURE}(a.s-\text{Procedure-identifier.idToNodeASI}, a.s-\text{CONTINUELABEL})
\]

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

\[
\text{SAVEEXCEPTIONSCOPES}(sn: \text{STATENODE}) \equiv \\
\text{do forall scope: scope } \in \text{EXCEPTIONSCOPE} \\
\text{activeHandler}(sn, scope) \equiv \text{activeHandler}(\text{Self}, scope) \equiv \text{RESETEXCEPTIONHANDLER}(scope) \\
\text{enddo}
\]

\[
\text{SAVEPROCEDURECONTROLBLOCK}(sn: \text{STATENODE}, cl: \text{CONTINUELABEL}) \equiv \\
\text{sn.agentMode1} := \text{Self.agentMode1} \\
\text{sn.agentMode2} := \text{Self.agentMode2} \\
\text{sn.agentMode3} := \text{Self.agentMode3} \\
\text{sn.agentMode4} := \text{Self.agentMode4} \\
\text{sn.agentMode5} := \text{Self.agentMode5} \\
\text{sn.currentStateId} := \text{Self.currentStateId} \\
\text{sn.currentLabel} := \text{Self.currentLabel} \\
\text{sn.continueLabel} := \text{Self.cl} \\
\text{sn.currentParentStateNode} := \text{Self.currentParentStateNode} \\
\text{sn.previousStateNode} := \text{Self.previousStateNode}
\]

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**

\[
\text{RETURN} = \text{def} \ \textit{VALUELABEL} \cup \textit{STATEEXITPOINT}
\]

**Behaviour**

\[
\text{EVALRETURN}(a: \text{RETURN}) \equiv
\begin{array}{l}
\text{if } a \in \textit{VALUELABEL} \text{ then} \\
\quad \text{EVALEXITPROCEDURE}(a.\textit{s-VALUELABEL}) \\
\text{else} \\
\quad \text{EVALEXITCOMPOSITESTATE}(a.\textit{s-STATEEXITPOINT})
\end{array}
\]

\[
\text{EVALEXITPROCEDURE}(vl: \textit{VALUELABEL}) =
\begin{array}{l}
\text{value}(\text{Self.callingProcedureNode.resultLabel, Self}) := \text{value}(vl, \text{Self}) \\
\text{RESTOREPROCEDURECONTROLBLOCK(\text{Self.callingProcedureNode})} \\
\text{RESTOREEXCEPTIONSCOPES(\text{Self.callingProcedureNode})}
\end{array}
\]

\[
\text{EVALEXITCOMPOSITESTATE}(sep: \textit{STATEEXITPOINT}) \equiv
\begin{array}{l}
\text{Self.stateNodeToBeExited} := \\
\quad \text{mk-STATENODEWITHEXITPOINT(\text{Self.currentParentStateNode, sep})} \\
\text{Self.agentMode3} := \text{exitingCompositeState}
\end{array}
\]

\[
\text{RESTOREPROCEDURECONTROLBLOCK}(sn: \textit{STATENODE}) \equiv
\begin{array}{l}
\text{Self.agentMode1} := sn.\text{agentMode1} \\
\text{Self.agentMode2} := sn.\text{agentMode2} \\
\text{Self.agentMode3} := sn.\text{agentMode3} \\
\text{Self.agentMode4} := sn.\text{agentMode4} \\
\text{Self.agentMode5} := sn.\text{agentMode5} \\
\text{Self.currentStateId} := sn.\text{currentStateId} \\
\text{Self.currentLabel} := sn.\text{continueLabel} \\
\text{Self.continueLabel} := sn.\text{continueLabel} \\
\text{Self.currentParentStateNode} := sn.\text{currentParentStateNode} \\
\text{Self.previousStateNode} := sn.\text{previousStateNode}
\end{array}
\]

\[
\text{RESTOREEXCEPTIONSCOPES}(sn: \textit{STATENODE}) \equiv
\begin{array}{l}
\text{do forall scope : scope } \in \textit{EXCEPTIONSCOPE} \\
\quad \text{activeHandler(\text{Self, scope}) := activeHandler(sn, scope)}
\end{array}
\]

**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.

\[
\text{TASK} = \text{def} \ \textit{Variable-identifier} \times \textit{VALUELABEL} \times \textit{BOOLEAN} \times \textit{CONTINUELABEL}
\]
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.

\[
\text{EVALTASK}(a:\text{TASK}) \equiv \\
\text{ASSIGN}(a.\text{s-Variable-identifier}, \text{value}(a.\text{s-VALUELABEL}, \text{Self}.\text{stateAgent.state}, \text{Self}.\text{currentStateId}) \\
\text{Self}.\text{currentStateId} := a.\text{s-CONTINUELABEL})
\]

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 \text{ASSIGNPARAMETERS} is
defined as a tuple consisting of a variable identifier, a natural number, and a continue label.

Representation
An action of type \text{ASSIGNPARAMETERS} is defined as a tuple consisting of a variable identifier, a natural number, and a
continue label.

\[
\text{ASSIGNPARAMETERS} = \text{def } \text{Variable-identifier} \times \text{NAT} \times \text{CONTINUELABEL}
\]

Behaviour

\[
\text{EVALASSIGNPARAMETERS}(a:\text{ASSIGNPARAMETERS}) \equiv \\
\text{let } v = \text{Self}.\text{currentSignalInstplainSignalValues}[a.\text{s-NAT}] \text{ in} \\
\text{ASSIGN}(a.\text{s-Variable-identifier}, v, \text{Self}.\text{stateAgent.state}, \text{Self}.\text{currentStateId}) \\
\text{endlet} \\
\text{Self}.\text{currentStateId} := a.\text{s-CONTINUELABEL}
\]

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 \text{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

\[
\text{EQUALITY} = \text{def } \text{VALUELABEL} \times \text{VALUELABEL} \times \text{CONTINUELABEL}
\]

Behaviour

\[
\text{EVALEQUALITY}(a:\text{EQUALITY}) \equiv \\
\text{if } \text{value}(a.\text{s-VALUELABEL}, \text{Self}) = \text{value}(a.\text{s2-VALUELABEL}, \text{Self}) \text{ then} \\
\text{value}(a.\text{s-CONTINUELABEL}, \text{Self}) := \text{True} \\
\text{else} \\
\text{value}(a.\text{s-CONTINUELABEL}, \text{Self}) := \text{False} \\
\text{endif} \\
\text{Self}.\text{currentStateId} := a.\text{s-CONTINUELABEL}
\]

Reference clauses
No references.
2.1.4.9 Primitive Decision

Explanation
The decision primitive is used for the evaluation of decisions. A decision in \textit{DETECTION} consists of a value label and a set of answer. An answer in \textit{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 \textit{DETECTION} consists of a value label and a set of answer. An answer in \textit{ANSWER} is a tuple consisting of a value label and a continue label.

\[
\textit{DETECTION} = \text{def} \quad \textit{VALUELABEL} \times \textit{ANSWER-set} \times \textit{CONTINUELABEL} \]

\[
\textit{ANSWER} = \text{def} \quad \textit{VALUELABEL} \times \textit{CONTINUELABEL} \]

Behaviour
Macro \textsc{evalDecision} chooses an answer such that the decision-value given by the corresponding value label coincides with the answer-value.

\[
\text{EVALDECISION}(d: \textit{DETECTION}) \equiv \\
\text{if} \quad \text{value}(d.s.-\textit{VALUELABEL}, \text{Self}) \in \{ \text{value}(an.s.-\textit{VALUELABEL}, \text{Self}) \mid an \in d.s.-\textit{ANSWER-set} \} \text{ then} \\
\quad \text{choose} \quad an \in d.s.-\textit{ANSWER-set} \land \\
\quad \text{value}(a.s.-\textit{VALUELABEL}, \text{Self}) = \text{value}(an.s.-\textit{VALUELABEL}, \text{Self}) \\
\quad \text{Self.currentLabel} := an.s.-\textit{CONTINUELABEL} \\
\text{endchoose} \\
\text{elseif} \quad a.s.-\textit{CONTINUELABEL} \neq \text{undefined} \text{ then} \\
\quad \text{Self.currentLabel} := d.s.-\textit{CONTINUELABEL} \\
\text{else} \quad \text{RAISEEXCEPTION(OutOfRange, empty)} \\
\text{endif} 
\]

Reference clauses
For definition of function \textit{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 \textit{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 \textit{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.

\[
\textit{OUTPUT} = \text{def} \quad \textit{SIGNAL} \times \text{VALUELABEL}* \times \text{VALUELABEL} \times \textit{VIAARG} \times \textit{CONTINUELABEL} 
\]

Behaviour
Macro \textsc{evalOutput} defines signal output by macro \textsc{signalOutput}, which takes the signal, a value sequence, the destination and the path as arguments.

\[
\text{EVALOUTPUT}(a: \textit{OUTPUT}) \equiv \\
\text{signalOutput}(a.s.-\textit{SIGNAL}, \text{values}(a.s.-\textit{VALUELABEL}-\text{seq}, \text{Self}), \text{value}(a.s.-\textit{VALUELABEL}, \text{Self}), \\
\quad a.s.-\textit{VIAARG}) \\
\text{Self.currentLabel} := a.s.-\textit{CONTINUELABEL} 
\]

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 \textit{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 \textit{ToArg} and \textit{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 \textit{ToArg}, \textit{ViaArg}. In the rule below, this choice is stated in abstract terms using the predicate \textit{Applicable} (see 2.1.1.4). If the constraints can not be met, the signal instance is discarded.

\[
\text{SIGNALOUTPUT}(s:\text{SIGNAL}, \text{vSeq}:\text{VALUE}*; \text{toArg}:\text{TOARG}, \text{viaArg}:\text{VIAARG}) \equiv \\
\text{if } \text{toArg} \in \text{PID} \land s \notin \text{toArg.s-Interface-definition} \text{ then} \\
\text{RAISEEXCEPTION(InvalidReference, empty)} \\
\text{else} \\
\text{choose } g: g \in \text{Self.outgates} \land \text{Applicable}(s, \text{toArg}, \text{viaArg}, g, \text{undefined}) \\
\text{extend } \text{PLAINSIGNALINST with } si \\
\text{si.plainSignalType} := s \\
\text{si.plainSignalValues} := \text{vSeq} \\
\text{si.toArg} := \text{toArg} \\
\text{si.viaArg} := \text{viaArg} \\
\text{si.plainSignalSender} := \text{Self.self} \\
\text{INSERT(si, now, g)} \\
\text{endextend} \\
\text{endchoose} \\
\text{endif}
\]

Reference clauses

For definition of function \textit{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 \textit{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 \textit{CREATE} is defined as tuple consisting of an agent-definition, a sequence of value labels, and a continue label.

\[
\text{CREATE} = \text{def } \text{Agent-identifier} \times \text{VALUELABEL}* \times \text{CONTINUELABEL}
\]

Behaviour

\[
\text{EVALCREATE}(a: \text{CREATE}) \equiv \\
\text{let } sas = \text{take}([\{ sas \in \text{SDLAGENTSET}: \text{sas.nodeASI} = a.a.s-Agent-definition.idToNodeASI \land sas \in \text{Self.getAgentSetsWithinScope} \}]) \text{ in} \\
\text{if } \text{sas.nodeASI.s-Number-of-instances.s-Maximum-number} \neq \text{undefined} \text{ then} \\
\text{let } n = |\{ \text{sa} \in \text{SDLAGENT}: \text{sa.owner} = \text{sas} \} | \text{ in} \\
\text{if } n < \text{sas.nodeASI.s-Number-of-instances.s-Maximum-number} \text{ then} \\
\text{CREATEAGENT}(\text{sas, Self.self, sas.nodeASI.s-Agent-type-definition}) \\
\text{else} \\
\text{Self.offspring} := \text{null} \\
\text{endif} \\
\text{endlet} \\
\text{else} \\
\text{CREATEAGENT}(\text{sas, Self.self, sas.nodeASI.s-Agent-type-definition}) \\
\text{endif}
\]
endlet
Self.currentLabel := a.s-CONTINUELABEL

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

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 \textit{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 \textit{SETTIMER}.

Representation
An action of type \textit{SET} is defined as tuple consisting of a time label, a timer, a sequence of value labels, and a continue label.

\[
SET = def \text{TIMELABEL} \times \text{TIMER} \times VALUELABEL* \times \text{CONTINUELABEL}
\]

Domains
\[
\text{TIMELABEL} = def \text{VALUELABEL}
\]

Behaviour
Macro \textit{EVALSET} defines the setting of a timer by macro \textit{SETTIMER}.

\[
\text{EVALSET}(a::SET) \equiv
\text{SETTIMER}(a.s-TIMER, values(a.s-VALUELABEL-seq, Self), value(a.s-TIMELABEL,Self))
Self.currentLabel := a.s-CONTINUELABEL
\]

Reference clauses
Definition of macro \textit{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 \textit{RESETTIMER}.

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

\[
RESET = def \text{TIMER} \times VALUELABEL* \times \text{CONTINUELABEL}
\]

Behaviour
Macro \textit{EVALRESET} specifies a reset of a timer with macro \textit{RESETTIMER}.
EVALRESET(\texttt{a.Reset}) \equiv
\begin{align*}
\text{RESETTIMER}(\texttt{a.s-TIMER}, \text{values}(\texttt{a.s-VALUETYPELABEL-seq, Self})) \\
\text{Self.currentLabel} := \texttt{a.s-CONTINUELABEL}
\end{align*}

Reference clauses
Definition of macro \texttt{RESET TIMER} can be found in 2.1.1.5.

2.1.4.14 Primitive \texttt{TimerActive}

Explanation
The \texttt{TimerActive} primitive is used for expressing a \texttt{timer active} expression. The primitive specifies the \texttt{timer active} check using the function \texttt{Active}.

Representation
An action of type \texttt{timer active} is defined as tuple consisting of a \texttt{timer}, a sequence of value labels, and a continue label.

\[
\text{TIMERACTIVE} = \text{def} \text{TIMER} \times \text{VALUETYPELABEL}^* \times \text{CONTINUELABEL}
\]

Behaviour
Macro \texttt{EVALTIMERACTIVE} specifies the evaluation of a \texttt{timer active} expression.

\[
\text{EVALTIMERACTIVE}(t: \text{TIMERACTIVE}) \equiv
\begin{align*}
\text{let } tmi & = \text{mk-TIMERINST}() \text{Self, t.s-TIMER, values(t.s-VALUETYPELABEL-seq, Self) } \text{in} \\
\text{value}(\text{Self.currentLabel, Self}) & := \text{semvalue}() \text{Active(tmi)} \\
\text{Self.currentLabel} & := \text{t.s-CONTINUELABEL}
\end{align*}
\]

Reference clauses
Definition of function \texttt{Active} can be found in 2.1.1.5.

2.1.4.15 Primitive \texttt{SetHandler}

Explanation
The \texttt{SetHandler} primitive is used for expressing the setting of an exception handler. An action of type \texttt{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 \texttt{SETEXCEPTIONHANDLER}.

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

\[
\text{SETHANDLER} = \text{def} \text{Exception-handler-name} \times \text{EXCEPTIONSCOPE} \times \text{CONTINUELABEL}
\]

Behaviour
Macro \texttt{EVALSETHANDLER} defines the setting of exception handler with macro \texttt{SETEXCEPTIONHANDLER}.

\[
\text{EVALSETHANDLER}(a: \text{SETHANDLER}) \equiv
\begin{align*}
\text{SETEXCEPTIONHANDLER}(a.s-\text{Exception-handler-name}, a.s-\text{EXCEPTIONSCOPE}) \\
\text{Self.currentLabel} := a.s-\text{CONTINUELABEL}
\end{align*}
\]

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.

\[ \text{RAISE} = \text{def} \ Exception\text{-identifier} \times \text{VALUELABEL}^* \]

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

\[ \text{EVALRAISE}(a:\text{RAISE}) \equiv \text{RAISEEXCEPTION}(a.s\text{-Exception}\text{-identifier}, \text{values}(a.s\text{-VALUELABEL}\text{-seq}, \text{Self})) \]

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
\[ \text{STOP} = \text{def} \ \{ \text{stop} \} \]

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

\[ \text{EVALSTOP}(a:\text{STOP}) \equiv \text{Self.}\text{agentMode2} := \text{stopping} \]

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

\[ \text{ANYVALUE} = \text{def} \ Sort\text{-identifier} \times \text{CONTINUELABEL} \]

Behaviour

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

2.1.4.20 Primitive SetRangeCheckLabel

Explanation
The \text{SETRANGECHECKVALUE} Primitive is used to set the value to be used in a range check.

Representation

\[ \text{SETRANGECHECKVALUE} = \text{def} \ VALUELABEL \times \text{CONTINUELABEL} \]

\textbf{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

\[ \text{SCOPE} = \text{def} \ \text{Connector-name} \times \text{Variable-definition-set} \times \text{STARTLABEL} \times \text{STEPLABEL} \times \text{CONTINUELABEL} \]

\[ \text{STEPLABEL} = \text{def} \ LABEL \]
Behaviour

\[ \text{EVALSCOPE}(a; \text{SCOPE}) \equiv \]
\[ \text{CREATECOMPOUNDNODEVARIABLES}(\text{Self}, a) \]
\[ \text{Self.currentLabel} := a.s-\text{STARTLABEL} \]

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
\[ \text{SKIP} = \text{def} \text{Connector-name} \cup \text{CONTINUELABEL} \]

Behaviour

\[ \text{EVALSKIP}(a; \text{SKIP}) \equiv \]
\[ \text{if } a \in \text{Connector-name} \text{ then} \]
\[ \text{Self.stateNodeChecked} := \text{Self.parentStateNode} \]
\[ \text{Self.currentConnector} := a.s-\text{Connector-name} \]
\[ \text{Self.agentMode2} := \text{selectingTransition} \]
\[ \text{Self.agentMode3} := \text{startSelection} \]
\[ \text{else} \]
\[ \text{Self.currentLabel} := a.s-\text{CONTINUELABEL} \]
\[ \text{endif} \]

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
\[ \text{BREAK} = \text{def} \text{Connector-name} \]

Behaviour

\[ \text{EVALBREAK}(a; \text{BREAK}) \equiv \]
\[ \text{if } \text{scopeName}(\text{Self}, \text{Self.currentStateId}) = a.s-\text{Connector-name} \text{ then} \]
\[ \text{Self.currentLabel} := \text{scopeContinueLabel}(\text{Self}, \text{Self.currentStateId}) \]
\[ \text{endif} \]
\[ \text{Self.currentStateId} := \text{caller}(\text{Self.stateAgent.state}, \text{Self.currentStateId}) \]

2.1.4.24 Primitive Continue

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

Representation
\[ \text{CONTINUE} = \text{def} \text{Connector-name} \]
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
\[
\text{ENTERSTATENODE} = \text{def} \ ( \text{State-name} \cup \{ \text{HISTORY, DASH} \} \times \text{STATEENTRYPOINT} \times \text{VALUELABEL}^* )
\]

Behaviour
\[
\text{EVALENTERSTATENODE}(a:\text{ENTERSTATENODE}) \equiv \\
\text{choose } sn: \text{sn } \in \text{STATENODE} \land \text{sn.stateName } = a.s-\text{State-name} \land \\
\text{sn.stateNodeKind } = \text{stateNode} \land \text{sn.parentStateNode } = \text{Self.currentParentStateNode} \\
\text{Self.stateNodesToBeEntered } := \\
\{ \text{mk-STATENODEWITHENTRYPOINT}(sn, a.s-\text{STATEENTRYPOINT}) \} \\
\text{endchoose} \\
\text{Self.agentMode3 } := \text{enteringStateNode} \\
\text{Self.agentMode4 } := \text{startPhase} \\
\text{Self.currentLabel } := \text{undefined} \\
\text{Self.continueLabel } := \text{undefined}
\]

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
\[
\text{LEAVESTATENODE} = \text{def} \ \text{State-name} \times \text{CONTINUELABEL}
\]

Behaviour
\[
\text{EVALLEAVESTATENODE}(a:\text{LEAVESTATENODE}) \equiv \\
\text{choose } sn: \text{sn } \in \text{STATENODE} \land \text{sn.stateName } = a.s-\text{State-name} \land \\
\text{sn.stateNodeKind } = \text{stateNode} \land \text{sn.parentStateNode } = \text{Self.currentParentStateNode} \\
\text{// assertion: } sn = \text{Self.previousStateNode} \\
\text{Self.stateNodesToBeLeft } := \text{collectCurrentSubStates}\{sn\} \\
\text{endchoose} \\
\text{Self.agentMode3 } := \text{leavingStateNode} \\
\text{Self.agentMode4 } := \text{leavePhase}
\]
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-Handler-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 }
```
Here we present the function that compiles an SDL state machine description into an ASM representation. A special labeling 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 \textit{LABEL}.

\textbf{static domain \textit{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.

\textbf{monitored uniqueLabel: DefinitionAS1 \times \textit{NAT} \rightarrow \textit{LABEL}}

For this function, it holds that:

\textbf{constraint} \forall d_1, d_2 \in \text{DefinitionAS1}: \forall i_1, i_2 \in \text{NAT}:
\textit{uniqueLabel}(d_1, i_1) = \textit{uniqueLabel}(d_2, i_2) \Leftrightarrow (d_1=d_2 \land 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 \textit{Decision-node} and \textit{Range-condition}.

\textbf{setToSeq(s: X-set): X^* =_{def} if s=\emptyset then empty else let el=c.take in <el> \cap setToSeq(s \setminus \{ el \}) endlet endif}

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

\textbf{compile: DefinitionAS1 \rightarrow BEHAVIOUR}

\textbf{compileExpr: DefinitionAS1 \times \textit{LABEL} \rightarrow BEHAVIOUR}

The computed value of an expression $e$ is always stored at \textit{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 \textit{startLabel} is defined also with a series of patterns in 2.2.4.

\subsection{2.2.1 States and Triggers}

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

\textbf{compile(a: DefinitionAS1): BEHAVIOUR =_{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=\textit{Variable-definition( name, *, init)} =>
| if init \neq \textit{undefined} then
| compileExpr(init, uniqueLabel(v,1)) \cup
| \{ \textit{mk-PRIMITIVE(uniqueLabel(v,1), uniqueLabel(name,uniqueLabel(init,1), False, undefined)} \}
| else \emptyset |
endif

| State-transition-graph(*, start, states, freeActions, handlers) =>
|    compile(start) \n|    \{ compile(s) | s ∈ states \} \n|    \{ compile(f) | f ∈ freeActions \} \n|    \{ compile(h) | h ∈ handlers \}

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

| h=Handle-node(*, vars, onexcept, transition) =>
|    if onexcept = undefined then \∅ else compileExpr(onexcept, transition.startLabel) endif \n|    \{ mk-PRIMITIVE(uniqueLabel(h,idx),
|        if vars[idx] ≠ undefined then
|            mk-ASSIGNPARAMETERS(var[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(onexcept, transition) =>
|    if onexcept = undefined then \∅ else compileExpr(onexcept, transition.startLabel) endif \n|    compile(transition)

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

| State-start-node(onexcept, *, transition) =>
|    if onexcept = undefined then \∅ else compileExpr(onexcept, transition.startLabel) endif \n|    \{ compile(transition) \n
| Procedure-start-node(onexcept, transition) =>
|    if onexcept = undefined then \∅ else compileExpr(onexcept, transition.startLabel) endif \n|    \{ compile(transition) \n
| Named-start-node(*, onexcept, trans) =>
|    if onexcept = undefined then \∅ else compileExpr(onexcept, transition.startLabel) endif \n|    \{ compile(transition) \n
| State-node(*, *, *, inputs, spontaneous, continuous, *) =>
|    \{ compile(i) | i ∈ inputs \} \n|    \{ compile(s) | s ∈ spontaneous \} \n|    \{ compile(c) | c ∈ continuous \}

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

| Input-node(*, *, vars, provided, onexcept, transition) =>
|    compileExpr(provided, undefined) \n|    if onexcept = undefined then \∅ else compileExpr(onexcept, transition.startLabel) endif \n|    \{ mk-PRIMITIVE(uniqueLabel(h,idx),
|        if vars[idx] ≠ undefined then
|            mk-ASSIGNPARAMETERS(var[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)
2.2.2 Terminators

Terminator(terminator, oneXcep) =>
   if oneXcep = undefined then ∅ else compileExpr(terminator, terminator.startLabel) endif Θ compileExpr(terminator, next)

Named-nextstate(stateName, undefined) =>
   mk-PRIMITIVE(uniqueLabel(n,1),
   mk-LEAVESTATE_NODE(t.parentASI, parentASI s-State-node, nodes.startLabel)) Θ compileNodes Θ compile(endnode)

where
   compileNodes = def
   if nodes = empty then ∅
   else compileExpr(nodes.last, endnode.startLabel) Θ
       \[ \bigcup \{ compileExpr(nodes[i], nodes[i+1], startLabel) \mid i \in [1..nodes.length-1] \} \]
   endif

end where

Spontaneous-transition(oneXcep, provided, transition) =>
   if oneXcep = undefined then ∅ else compileExpr(oneXcep, transition.startLabel) endif Θ compileExpr(provided, undefined) Θ compile(transition)

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

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

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

\( n = \text{Transition}(nodes, endnode) \) =>
   \{ mk-PRIMITIVE(uniqueLabel(a,1),
   mk-LEAVESTATE_NODE(t.parentASI, parentASI s-State-node, nodes.startLabel)) \} Θ compileNodes Θ compile(endnode)

\( n = \text{Named-nextstate}(stateName, undefined) \) =>
   \{ mk-PRIMITIVE(uniqueLabel(n,1),
   mk-ENTERSTATE_NODE(stateName, undefined, empty)) \}

\( n = \text{Named-nextstate}(stateName, Nextstate-parameters(exprList, entry)) \) =>
   if exprList = empty then ∅
   else compileExpr(exprList.last, uniqueLabel(n,1)) Θ
       \[ \bigcup \{ compileExpr(exprList[i], exprList[i+1], startLabel) \mid i \in [1..exprList.length-1] \} \]
   endif
   \{ mk-PRIMITIVE(uniqueLabel(n,1),
   mk-ENTERSTATE_NODE(stateName, entry, <uniqueLabel(e,1) \mid e \in exprList>>) \}

\( n = \text{Dash-nextstate}(undefined) \) =>
   \{ mk-PRIMITIVE(uniqueLabel(n,1), mk-ENTERSTATE_NODE(DASH, undefined, empty)) \}

\( n = \text{Dash-nextstate}(HISTORY) \) =>
   \{ mk-PRIMITIVE(uniqueLabel(n,1), mk-ENTERSTATE_NODE(HISTORY, undefined, empty)) \}

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

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

\( v = \text{Value-return-node(expr)} \) =>
   compileExpr(expr, uniqueLabel(v,1)) Θ
   \{ mk-PRIMITIVE(uniqueLabel(v,1), mk-RETURN(uniqueLabel(v,1))) \}
where scope: EXCEPTIONSCOPE =def case o.parentASI 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

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: DefinitionASI, next: LABEL): BEHAVIOUR =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.

| a=On exception(n) =>
  | {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(e,1)) ∪
  | U{ compileExpr(exprList[i], exprList[i+1].startLabel) | i ∈ 1.. exprList.length-1 } endif ∪
  | {mk-PRIMITIVE(uniqueLabel(e,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(questoin, aseq[1].startLabel) ∪
  | { compileExpr(aseq[idx].s-implicit,
      if idx=aseq.length then uniqueLabel(d, 1) else aseq[idx+1].startLabel endif)
    | if idx ∈ toSet(1.. iseq.length) } ∪
  | { mk-PRIMITIVE(uniqueLabel(d, 1),
    mk-DECISION(uniqueLabel(question, 1),
      { mk-ANSEWER(uniqueLabel(ans.s-implicit, 1), ans.s-Transition.startLabel)
      | ans ∈ answerset },
      if elseanswer=undefined then undefined else elseanswer.s-Transition endif})) } |
  | U { compile(ans.s-Transition) | ans ∈ answerset } ∪
  | compile(elseanswer.s-Transition) } |
\begin{verbatim}
| Decision-node => esAction
| Connect-node => esStimulusOrStart
| Named-start-node => esStimulusOrStart
| otherwise undefined
endcase

endwhere

| Graph-node(action, onexcep) =>
   if onexcep = undefined then \emptyset 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 \emptyset
      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 \emptyset
      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 \emptyset
   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 \emptyset
   else compileExpr(exprList.last, uniqueLabel(c,1)) ∪
       \{ compileExpr(exprList[i], exprList[i+1], startLabel) | i ∈ 1.. exprList.length-1 \}
       endif ∪
       (let paramDef = procedureId.idToNodeASI.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,\

\end{verbatim}
if initNodes = empty then trans.startLabel else initNodes.head.startLabel endif, 
if stepNodes = empty then trans.startLabel else stepNodes.head.startLabel endif, 
next)) } \cup 

compile(expr) \cup 
compileExpr(trans, undefined) \cup 
if stepNodes = empty then \emptyset \else \cup 
compileExpr(\text{stepNodes}.last, trans.startLabel) \cup 
( \cup \ | \ \text{compileExpr}(\text{stepNodes}[i], \text{stepNodes}[i+1].startLabel) \mid i \in 1.. \text{stepNodes}.length-1 } 
\text{if initNodes = empty then } \emptyset \else \cup 
\text{compileExpr}(\text{initNodes}.last, trans.startLabel) \cup 
( \cup \ | \ \text{compileExpr}(\text{initNodes}[i], \text{initNodes}[i+1].startLabel) \mid i \in 1.. \text{initNodes}.length-1 } 
\text{endif}

| s = \text{Set-node(expr, timerId, exprList)} \Rightarrow 
| if exprList = empty then \emptyset \else \cup 
\text{compileExpr}(\text{exprList}.last, expr.startLabel) \cup 
( \cup \ | \ \text{compileExpr}(\text{exprList}[i], \text{exprList}[i+1].startLabel) \mid i \in 1.. \text{exprList}.length-1 } 
\text{compileExpr(expr, uniqueLabel(s,1)) \cup } 
\{ \text{mk-PRIMITIVE(uniqueLabel(s,1), } 
\text{mk-SET(uniqueLabel(expr,1), timerId, <uniqueLabel(e,1) \mid e \in exprList >, next)) } 
| r = \text{Reset-node(timerId, exprList)} \Rightarrow 
| if exprList = empty then \emptyset \else \cup 
\text{compileExpr(\text{exprList}.last, uniqueLabel(r,1)) \cup } 
( \cup \ | \ \text{compileExpr(\text{exprList}[i], \text{exprList}[i+1].startLabel) \mid i \in 1.. \text{exprList}.length-1 } 
\{ \text{mk-PRIMITIVE(uniqueLabel(r,1), } 
\text{mk-RESET(timerId, <uniqueLabel(e,1) \mid e \in exprList >, next)) } 
| r = \text{Range-condition(items)} \Rightarrow 
\text{(let iseq = items.setToSeq in } 
\{ \text{mk-PRIMITIVE(uniqueLabel(r,1), } 
\text{mk-OPERATIONAPPLICATION(sdlTrue.idToNodeAS1, empty, } 
\text{uniqueLabel(r, iseq.length+1()))) } \cup 
\{ \text{compileExpr(iseq[\text{id}], uniqueLabel(r, \text{id}))) \mid \text{idx} \in \text{toList(1.. iseq.length}} } \cup 
\{ \text{mk-PRIMITIVE(uniqueLabel(r, \text{id}), } 
\text{mk-OPERATIONAPPLICATION(sdlOr, } 
\text{< uniqueLabel(r, \text{id}+1), uniqueLabel(iseq[\text{id}]+1) >, } 
\text{if id=1 then next else iseq[\text{id}].startLabel endif)) } 
\mid \text{idx} \in \text{toList(1.. iseq.length}} } \cup 
\{ \text{mk-PRIMITIVE(uniqueLabel(r, 0), mk-BREAK(undefined)) } 
\text{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 = \text{Open-range(id, expr)} \Rightarrow 
\text{compileExpr(expr, uniqueLabel(o, 1)) \cup } 
\{ \text{mk-PRIMITIVE(uniqueLabel(o, 1), } 
\text{mk-OPERATIONAPPLICATION(id.idToNodeAS1, } 
\text{< rangeCheckValue, uniqueLabel(expr, 1) >, next)) } 
| c = \text{Closed-range(r1, r2)} \Rightarrow 
\text{compileExpr(r1, r2.startLabel) \cup } 
\text{compileExpr(r2, uniqueLabel(c, 1)) \cup } 
\{ \text{mk-PRIMITIVE(uniqueLabel(c, 1), } 
\text{mk-OPERATIONAPPLICATION(sdlAnd, < uniqueLabel(r1, 1), uniqueLabel(r2, 1) >, next)) } 

ITU-T Z.100/Annex F3 (01/2000)
\[ \text{Literal}(id) => \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(c,1)), \]
\[ \text{mk-OPERATIONAPPLICATION}(id.idToNodeASI, empty, next) \} \]

\[ c=\text{Conditional-expression}(boolExpr, consExpr, altExpr) => \]
\[ \text{compileExpr}(boolExpr, uniqueLabel(c, 2)) \cup \]
\[ \text{compileExpr}(consExpr, next) \cup \]
\[ \text{compileExpr}(altExpr, next) \cup \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(c,2), \]
\[ \text{mk-OPERATIONAPPLICATION}(sdlTrue.idToNodeASI, empty, uniqueLabel(c, 1))) \} \cup \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(c, 1), \]
\[ \text{mk-DECISION}(uniqueLabel(boolExpr, 1), \]
\[ \{ \text{mk-ANSWER}(uniqueLabel(c, 2), consExpr.startLabel), altExpr.startLabel) \} \}

\[ e=\text{Equality-expression(first, second)} => \]
\[ \text{compileExpr}(first, second.startLabel) \cup \]
\[ \text{compileExpr}(second, uniqueLabel(e, 1)) \cup \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(e, 1), \]
\[ \text{mk-EQUALITY}(uniqueLabel(first, 1), uniqueLabel(second, 1), next)) \}

\[ o=\text{Operation-application(id, exprList)} => \]
\[ \text{if } exprList = \text{empty then } \emptyset \]
\[ \text{else } \text{compileExpr}(exprList.last, uniqueLabel(c, 1)) \cup \]
\[ \bigcup \{ \text{compileExpr}(exprList[i], exprList[i+1].startLabel) \mid i \in 1.. exprList.length-1 \} \]
\[ \text{endif} \cup \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(c, 1), \]
\[ \text{mk-OPERATIONAPPLICATION}(id.idToNodeASI, \]
\[ < uniqueLabel(e, 1) \mid e \in exprList >, \]
\[ next) \} \}

\[ r=\text{Range-check-expression(range, expr)} => \]
\[ \text{compileExpr}(expr, uniqueLabel(r,2)) \cup \]
\[ \text{compileExpr}(range, undefined) \cup \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(r,2), \]
\[ \text{mk-SETRANGECHECKVALUE}(uniqueLabel(expr,1), uniqueLabel(r,1))) \} \cup \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(r, 1), \]
\[ \text{mk-SCOPE}(undefined, \emptyset, range.startLabel, undefined, next)) \}

\[ v=\text{Variable-access(id)} => \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(c,1), \text{mk-VAR}(id, next)) \}

\[ n=\text{Now-expression()} => \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(n,1), \text{mk-SYSTEMVALUE}(kNow, next)) \}

\[ s=\text{Self-expression()} => \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(n,1), \text{mk-SYSTEMVALUE}(kSelf, next)) \}

\[ p=\text{Parent-expression()} => \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(n,1), \text{mk-SYSTEMVALUE}(kParent, next)) \}

\[ o=\text{Offspring-expression()} => \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(n,1), \text{mk-SYSTEMVALUE}(kOffspring, next)) \}

\[ s=\text{Sender-expression()} => \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(n,1), \text{mk-SYSTEMVALUE}(kSender, next)) \}

\[ t=\text{Timer-active-expression(id, exprList)} => \]
\[ \text{if } exprList = \text{empty then } \emptyset \]
\[ \text{else } \text{compileExpr}(exprList.last, uniqueLabel(c, 1)) \cup \]
\[ \bigcup \{ \text{compileExpr}(exprList[i], exprList[i+1].startLabel) \mid i \in 1.. exprList.length-1 \} \]
\[ \text{endif} \cup \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(t,1), \]
\[ \text{mk-TIMERACTIVE}(id, uniqueLabel(second,1), < uniqueLabel(e, 1) \mid e \in exprList >, next) \} \}

\[ a=\text{Any-expression(id)} => \]
\[ \{ \text{mk-PRIMITIVE}(uniqueLabel(a,1), \text{mk-ANYVALUE}(id, next)) \} \]
2.2.4 Start Labels

This clause introduces the function \texttt{startLabel} which is responsible to define the start labels of all behavioural syntax constructs.

\begin{verbatim}
startLabel(x: DefinitionASI): LABEL = 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.parentASI \in 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 \neq undefined then startLabel(dest)
            elseif expr = empty then uniqueLabel(o,1)
            else startLabel(expr.head) endif
        else 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)
    endcase // end of the compileExpr function definition
\end{verbatim}
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.)

\[ n = \text{Named-nextstate}(\ast, \text{Nextstate-parameters}(\text{exprList}, \ast)) \]
\[ a = \text{Action-return-node}() \]
\[ v = \text{Value-return-node}(\text{expr}) \]
\[ n = \text{Named-return-node}(\text{expr}) \]
\[ j = \text{Join-node}(\ast) \]
\[ b = \text{Break-node}(\ast) \]
\[ c = \text{Continue-node}(\ast) \]
\[ r = \text{Raise-node}(\ast, \ast) \]
\[ d = \text{Decision-node}(\text{question}, \ast, \ast, \ast) \]
\[ \text{Decision-answer}(r, \ast) \]
\[ n = \text{Named-start-node}(\ast, \ast, \text{trans}) \]
\[ o = \text{Open-range}(\ast, \text{expr}) \]
\[ c = \text{Closed-range}(\ast, \ast) \]
\[ l = \text{Literal}(\ast) \]
\[ c = \text{Conditional-expression}(\ast, \ast, \ast) \]
\[ \text{Equality-expression}(\text{first}, \ast) \]
\[ r = \text{Range-check-expression}(\ast, \text{expr}) \]
\[ v = \text{Variable-access}(\text{id}) \]
\[ o = \text{Operation-application}(\ast, \text{exprList}) \]
\[ v = \text{Identifier}(\ast, \ast) \]
\[ n = \text{Now-expression}() \]
\[ s = \text{Self-expression}() \]
\[ p = \text{Parent-expression}() \]
\[ o = \text{Offspring-expression}() \]
\[ s = \text{Sender-expression}() \]
\[ t = \text{Timer-active-expression}(\ast, \text{exprList}) \]
\[ a = \text{Any-expression}(\ast) \]
\[ v = \text{Value-returning-call-node}(\ast, \text{exprList}) \]

endcase

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.

\[
\text{initially} \quad \begin{align*}
\text{ behaviour } &= \text{ rootNodeAS1}\text{.compile } \land \\
\text{ if } \text{ rootNodeAS1.s-Agent-definition } \neq \text{ undefined } \text{ then } \\
& \quad \text{ system.nodeAS1 } = \text{ rootNodeAS1.s-Agent-definition } \land \\
& \quad \text{ system.owner } = \text{ undefined } \land \\
& \quad \text{ system.agentModel } = \text{ initialisation } \land \\
& \quad \text{ system.program } = \text{ AGENT-SET-PROGRAM } \\
\text{ else } & \quad \text{ system.program } = \text{ undefined } \\
\text{ endif }
\end{align*}
\]

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 \( \text{ rootNodeAS1.s-Agent-definition } \neq \text{ undefined} \). 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 (\text{ system.owner } = \text{ undefined}). The SAM program of the agent \text{ 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.

\[
\text{CREATEALLAGENTSETS}(\text{ow:AGENT}, \text{ atd:Agent-type-definition}) \equiv \\
\text{ do forall } \text{ ad: ad } \in \text{ atd.collectAllAgentDefinitions} \quad \text{ CREATEAGENTSET}(\text{ow, ad}) \quad \text{ enddo}
\]

where

\[
\text{collectAllAgentDefinitions}(\text{atd: Agent-type-definition}): \text{ Agent-definition-set } \equiv \\
\text{ if } \text{ atd.s-Agent-type-identifier } = \text{ undefined } \text{ then } \\
\text{ atd.s-Agent-definition-set } \\
\text{ else } \\
\text{ atd.s-Agent-definition-set } \cup \\
\text{ atd.s-Agent-type-identifier.idToNodeAS1.collectAllAgentDefinitions }
\]

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

\[
\text{CREATEAGENTSET}(\text{ow:SDLAGENT, ad:Agent-definition}) \equiv \\
\text{ extend } \text{ AGENT with sas} \quad \\
\text{ sas.nodeAS1 } := \text{ ad } \\
\text{ sas.owner } := \text{ ow } \\
\text{ CREATEALLGATES (ow, ad.s-Agent-type-identifier.idToNodeAS1) } \\
\text{ sas.program } := \text{ AGENT-SET-PROGRAM } \\
\text{ sas.agentModel1 } := \text{ initialisation }
\]

\text{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 nodeASI, 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.agentModel = initialisation then
        INITAGENTSET
    endif
    if Self.agentModel = 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.nodeASI.s-Agent-type-identifier.idToNodeASI.s-Agent-kind = SYSTEM then
        CREATEALLGATES(Self, Self.nodeASI.s-Agent-type-identifier.idToNodeASI)
    endif
    CREATEALLAGENTS(Self, null, Self.nodeASI)
    Self.agentModel1 := 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.
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

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:

```plaintext
if Self.agentMode1 = initialisation then
    INITAGENT
elsif 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.agentModel1 := 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.
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) ≡
\[
\text{do forall } gd: gd \in atd.collectAllGateDefinitions \\
\text{ CREATEGATE}(ow, gd) \\
\text{ enddo }
\]

where

\[
\text{collectAllGateDefinitions}(atd: Agent-type-definition): \text{Gate-definition-set} =
\text{if } atd.s-Agent-type-identifier = \text{undefined then} \\
\text{ atd.s-Gate-definition-set} \\
\text{else} \\
\text{ atd.s-Gate-definition-set} \cup \\
\text{ atd.s-Agent-type-identifier.idToNodeAS1.collectAllGateDefinitions} \\
\text{ endif}
\]

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 $\neq \emptyset$ then
  extend $GATE$ with $g$
  
  $g$.myAgent := $ow$
  $g$.nodeASI := $gd$
  $g$.schedule := empty
  $g$.direction := inDir
endextend
endif

if $gd.s$-Out-signal-identifier-set $\neq \emptyset$ then
  extend $GATE$ with $g$
  
  $g$.myAgent := $ow$
  $g$.nodeASI := $gd$
  $g$.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 := $ow$
  $g$.nodeASI := 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 $cd$ : $cd$ $\in$ $atd$.collectAllChannelDefinitions
    CREATECHANNEL($ow$, $cd$)
  enddo

where
  collectAllChannelDefinitions($atd$:$Agent-type-definition$): Channel-definition-set = def
  if $atd.s$-Agent-type-identifier = undefined then
    $atd.s$-Channel-definition-set
  else
    $atd.s$-Channel-definition-set $\cup$
    $atd.s$-Agent-type-identifier.idToNodeASI.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 $cp$ : $cp$ $\in$ $cd$.s-Channel-path-set
    CREATECHANNELPATH($ow$, $cd$.s-NODELAY, $cp$, $cd$)
Creating a channel amounts to creating the specified channel paths.

\[
\text{CREATECHANNELPATH}(\text{ow}: \text{AGENT}, \text{nd}: \text{NODELAY}, \text{cp}: \text{Channel-path}, \text{cd}: \text{Channel-definition}) \equiv \\
\begin{align*}
\text{choose fromGate} & : \text{fromGate} \in \text{GATE} \land \text{fromGate.nodeAS1} = \text{cp.s-Originating-gate.idToNodeAS1} \land \\
& (\text{OuterGate}(\text{ow}, \text{fromGate}, \text{inDir}) \lor \text{InnerGate}(\text{ow}, \text{fromGate}, \text{outDir}) ) \\
\text{choose toGate} & : \text{toGate} \in \text{GATE} \land \text{toGate.nodeAS1} = \text{cp.s-Destination-gate.idToNodeAS1} \land \\
& (\text{OuterGate}(\text{ow}, \text{toGate}, \text{outDir}) \lor \text{InnerGate}(\text{ow}, \text{toGate}, \text{inDir}) ) \\
\text{CREATELINK}(\text{ow}, \text{fromGate}, \text{toGate}, \text{nd}, \text{cp.s-Signal-identifier-set}, \text{cd}) \\
\text{endchoose} \\
\text{endchoose}
\end{align*}
\]

where

\[
\begin{align*}
\text{OuterGate}(\text{ow}: \text{AGENT}, \text{g}: \text{GATE}, \text{dir}: \text{DIRECTION}): \text{BOOLEAN} & = \text{def} \\
\text{g.myAgent} & = \text{ow.owner} \land \text{g.direction} = \text{dir} \\
\text{InnerGate}(\text{ow}: \text{AGENT}, \text{g}: \text{GATE}, \text{dir}: \text{DIRECTION}): \text{BOOLEAN} & = \text{def} \\
\text{g.myAgent.owner} & = \text{ow} \land \text{g.direction} = \text{dir}
\end{align*}
\]

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 \text{myAgent} and \text{direction} defined on gates.

### 2.3.1.7 Link Creation and Removal

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

\[
\text{CREATEALLLINKS}(\text{ow}: \text{AGENT}) \equiv \\
\text{do forall g}: \text{g} \in \text{ow.ingates} \\
\text{CREATELINK}(\text{ow}, \text{g}, \text{ow.inport}, \text{NODELAY}, \text{g.nodeAS1.s-In-signal-identifier-set}, \text{undefined}) \\
\text{enddo}
\]

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 \text{ingates}) with the input port of an agent.

\[
\text{CREATELINK}(\text{ow}: \text{AGENT}, \text{fromGate}: \text{GATE}, \text{toGate}: \text{GATE}, \text{nd}: \text{NODELAY}, \text{w}: \text{In-signal-identifier-set}, \\
\text{cd}: \text{Channel-definition}) \equiv \\
\text{extend LINK with l} \\
\text{l.nodeAS1} := \text{cd} \\
\text{l.owner} := \text{ow} \\
\text{l.from} := \text{fromGate} \\
\text{l.to} := \text{toGate} \\
\text{l.noDelay} := \text{nd} \\
\text{l.with} := \text{w} \\
\text{l.program} := \text{LINK-PROGRAM}
\]

\text{LINK-PROGRAM:}

\text{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. \text{LINK-PROGRAM} defines the dynamic behaviour of link agents.

\[
\text{REMOVEALLLINKS}(\text{ow}: \text{AGENT}) \equiv \\
\text{do forall l}: \text{l} \in \text{LINK} \land \text{l.owner} = \text{ow} \\
\text{REMOVELINK(l)} \\
\text{enddo}
\]
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 ≡
    if atd.s-Agent-type-identifier = undefined then
      atd.s-Variable-definition-set
    else
      atd.s-Variable-definition-set ∪
      atd.s-Agent-type-identifier.idToNodeASI.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 ≡
    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.idToNodeASI.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

\[
\text{collectAllVariableDefinitions}(pd: \text{Procedure-definition}): \text{Variable-definition-set} = \begin{cases} 
\text{def} & \text{if } pd.s.-\text{Procedure-identifier} = \text{undefined} \\
\text{pd.s.-Variable-definition-set} & \text{else}
\end{cases}
\]

\[
\text{collectAllProcedureFPars}(pd: \text{Procedure-definition}): \text{Procedure-formal-parameter*} = \begin{cases} 
\text{def} & \text{if } pd.s.-\text{Procedure-identifier} = \text{undefined} \\
\text{pd.s.-Procedure-formal-parameter*} & \text{else}
\end{cases}
\]

endwhere

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

\[
\text{CREATECOMPOUNDNODEVARIABLES}(sa: \text{SDLAGENT}, scope: \text{SCOPE}) \equiv
\begin{align*}
\text{extend STATEID with sid} \\
\text{sa.currentStateId} & := \text{sid} \\
\text{scopeName}(\text{Self}, \text{sid}) & := \text{scope.s.-Connector-name} \\
\text{scopeContinueLabel}(\text{Self}, \text{sid}) & := \text{scope.s.-CONTINUELABEL} \\
\text{scopeStepLabel}(\text{Self}, \text{sid}) & := \text{scope.s.-STEPLABEL} \\
sa.stateAgent.state & := \text{initAgentState}(sa.stateAgent.state, \text{sid}, \\
\text{sa.currentStateId}, \text{scope.s.-Variable-definition-set})
\end{align*}
\]

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.

\[
\text{CREATESTATEMACHINE}(smd: \text{State-machine-definition}) \equiv
\text{CREATETOPSTATEPARTITION}(smd)
\]

When an SDL agent is created, the macro \text{CREATESTATEMACHINE} is applied with the effect that the root node (\text{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 \text{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.

\[
\text{INITSTATEMACHINE} \equiv
\begin{align*}
\text{if } \text{Self.stateNodesToBeCreated} \neq \emptyset & \text{ then} \\
\text{CREATESTATENODE} \\
\text{elseif } \text{Self.statePartitionsToBeCreated} \neq \emptyset & \text{ then} \\
\text{CREATESTATEPARTITION} \\
\text{elseif } \text{Self.ehNodesToBeCreated} \neq \emptyset & \text{ then} \\
\text{CREATEEXCEPTIONHANDLERENODE} \\
\text{elseif } \text{Self.stateNodesToBeSpecialised} \neq \emptyset & \text{ then} \\
\text{CREATEINHERITEDSTATE} & \text{ // these are composite states!} \\
\text{elseif } \text{Self.stateNodesToBeRefined} \neq \emptyset & \text{ then}
\end{align*}
\]
CREATESTATEREFINEMENT
else
    Self.agentMode2 := initialisationFinished
endif

Nodes to be created are kept in the agent’s state components \texttt{stateNodesToBeCreated}, \texttt{statePartitionsToBeCreated}, \texttt{stateNodesToBeSpecialised}, and \texttt{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.

\[
\text{CREATEPROCEDUREGRAPH}(pg:\texttt{Procedure-graph},cl:\texttt{CONTINUELABEL}) \equiv \\
\text{CREATEPROCEDURESTATENODE}(pg,cl)
\]

When a procedure is called, the macro \texttt{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 \texttt{INITPROCEDUREGRAPH}.

\[
\text{INITPROCEDUREGRAPH} \equiv \\
\text{if } Self\cdot\texttt{stateNodesToBeCreated} \neq \emptyset \text{ then } \\
\text{CREATESTATENODE} \\
\text{elseif } Self\cdot\texttt{statePartitionsToBeCreated} \neq \emptyset \text{ then } \\
\text{CREATESTATEPARTITION} \\
\text{elseif } Self\cdot\texttt{ehNodesToBeCreated} \neq \emptyset \text{ then } \\
\text{CREATEEXCEPTIONHANDLENODE} \\
\text{elseif } Self\cdot\texttt{stateNodesToBeSpecialised} \neq \emptyset \text{ then } \\
\text{CREATEINHERITEDSTATE} \\
\text{elseif } Self\cdot\texttt{stateNodesToBeRefined} \neq \emptyset \text{ then } \\
\text{CREATESTATEREFINEMENT} \\
\text{else} \\
    Self\cdot\texttt{agentMode4} := \texttt{initialisationFinished} \\
\text{endif}
\]

Nodes to be created are kept in the agent’s state components \texttt{stateNodesToBeCreated}, \texttt{statePartitionsToBeCreated}, \texttt{stateNodesToBeSpecialised}, and \texttt{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 \texttt{STATENODE}. A macro is defined to handle the creation of state nodes. State partitions are also modelled as elements of the domain \texttt{STATENODE}, but are not treated in this clause.

\[
\text{CREATESTATENODE} \equiv \\
\text{choose } snd: snd \in \texttt{Self\cdotstateNodesToBeCreated} \\
\text{Self\cdotstateNodesToBeCreated} := \texttt{Self\cdotstateNodesToBeCreated} \setminus \{snd\} \\
\text{extend } \texttt{STATENODE} \text{ with } sn \\
    sn\cdot\texttt{nodeAS1} := snd \quad \text{used, e.g., as argument for startLabel} \\
    sn\cdot\texttt{owner} := \texttt{Self} \\
    sn\cdot\texttt{parentStateNode} := \texttt{Self\cdotcurrentParentStateNode} \\
    sn\cdot\texttt{stateNodeKind} := \texttt{stateNode} \\
    sn\cdot\texttt{stateName} := \texttt{snd\cdotState-name} \\
    sn\cdot\texttt{stateTransitions} := \texttt{snd\cdotgetStateTransitions} \\
    sn\cdot\texttt{startTransitions} := \emptyset \quad \text{updated if the state node is refined} \\
    \texttt{Self\cdotstateNodesToBeRefined} := \texttt{Self\cdotstateNodesToBeRefined} \cup \{sn\} \\
    \texttt{Self\cdotstateNodesToBeSpecialised} := \texttt{Self\cdotstateNodesToBeSpecialised} \cup \{sn\} \\
\text{endextend} \\
\text{endchoose}
\]
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 $\text{stateNodesToBeCreated}$. 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 := ∅ // 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 = 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 \textit{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.

\textbf{CREATE\textit{TOPSTATEPARTITION}}(smd.\textit{State-machine-definition}) = \\
\textbf{extend} \textit{STATENODE with} \textit{sn} \\
\textit{sn}.\textit{nodeAS1} := \textit{smd} \\
\textit{sn}.\textit{owner} := \textit{Self} \\
\textit{Self}.\textit{topStateNode} := \textit{sn} \\
\textit{sn}.\textit{parentStateNode} := \textit{undefined} \\
\textit{sn}.\textit{stateNodeKind} := \textit{statePartition} \\
\textit{sn}.\textit{stateName} := \textit{smd}.\textit{s-State-name} \\
\textit{sn}.\textit{stateTransitions} := \emptyset \\
\textit{sn}.\textit{startTransitions} := \emptyset \\
\begin{verbatim}
if \textit{smd} \neq \textit{undefined} then
    \textit{Self}.\textit{stateNodesToBeRefined} := \{\textit{sn}\} \\
    \textit{Self}.\textit{stateNodesToBeSpecialised} := \{\textit{sn}\}
else
    \textit{Self}.\textit{stateNodesToBeRefined} := \emptyset \\
    \textit{Self}.\textit{stateNodesToBeSpecialised} := \emptyset
end\textbf{if}
\end{verbatim}
\textit{Self}.\textit{stateNodesToBeCreated} := \emptyset \\
\textit{Self}.\textit{statePartitionsToBeCreated} := \emptyset \\
\textit{Self}.\textit{stateNodesToBeSpecialised} := \emptyset \\
\textbf{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 \textbf{CREATE\textit{TOPSTATEPARTITION}}. 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.

\textbf{CREATE\textit{STATEPARTITION}} = \\
\textbf{choose} \textit{spd}: \textit{spd} \in \textit{Self}.\textit{statePartitionsToBeCreated} \\
\textit{Self}.\textit{statePartitionsToBeCreated} := \textit{Self}.\textit{statePartitionsToBeCreated} \setminus \{\textit{spd}\} \\
\textbf{extend} \textit{STATENODE with} \textit{sn} \\
\textit{sn}.\textit{nodeAS1} := \textit{spd} \quad // used, e.g., as argument for \textit{startLabel} \\
\textit{sn}.\textit{owner} := \textit{Self} \\
\textit{sn}.\textit{parentStateNode} := \textit{Self}.\textit{currentParentStateNode} \\
\textit{sn}.\textit{stateNodeKind} := \textit{statePartition} \\
\textit{sn}.\textit{stateName} := \textit{spd}.\textit{s-Name} \\
\textit{sn}.\textit{stateTransitions} := \emptyset \\
\textit{sn}.\textit{startTransitions} := \emptyset \\
\begin{verbatim}
if \textit{cd} \in \textit{Entry-connection-definition} then
    \textit{entryConnection}(%s-Outer-entry-point, %s-Inner-entry-point) := %s-Inner-entry-point
else
    \textit{cd} \in \textit{Exit-connection-definition} then
    \textit{exitConnection}(%s-Inner-exit-point, %s-Outer-exit-point) := %s-Outer-exit-point
end\textbf{if}
\end{verbatim}
\textit{Self}.\textit{stateNodesToBeRefined} := \textit{Self}.\textit{stateNodesToBeRefined} \cup \{\textit{sn}\} \\
\textit{Self}.\textit{stateNodesToBeSpecialised} := \textit{Self}.\textit{stateNodesToBeSpecialised} \cup \{\textit{sn}\}
\textbf{endextend}
\textbf{end\textbf{choose}}

(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 \textit{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.

\[
\text{CREATEINHERITEDSTATE} \equiv \\
\text{choose sns: sns } \in \text{Self.stateNodesToBeSpecialised} \\
\text{Self.stateNodesToBeSpecialised } := \text{Self.stateNodesToBeSpecialised } \setminus \{\text{sns}\} \\
\text{let cstd } = \text{sns.nodeAS1.s-Composite-state-type-identifier.idToNodeAS1 in} \\
\text{if cstd.s-Composite-state-type-identifier } \neq \text{undefined then} \\
\text{extend STANODE with sn} \\
\text{sn.nodeAS1 } := \text{cstd.s-Composite-state-type-identifier.idToNodeAS1} \\
\text{sn.owner } := \text{Self} \\
\text{sn.parentStateNode } := \text{sns.parentStateNode} \\
\text{sn.stateNodeKind } := \text{sns.stateNodeKind} \\
\text{sn.stateName } := \text{sns.stateName} \\
\text{sn.stateTransitions } := \emptyset \\
\text{sn.startTransitions } := \emptyset \quad \// \text{updated if the state node is refined} \\
\text{sns.inheritedStateNode } := \text{sn} \\
\text{Self.stateNodesToBeRefined } := \text{Self.stateNodesToBeRefined } \cup \{\text{sn}\} \\
\text{Self.stateNodesToBeSpecialised } := \text{Self.stateNodesToBeSpecialised } \cup \{\text{sn}\} \\
\text{endextend} \\
\text{else} \\
\text{sns.inheritedStateNode } := \text{undefined} \\
\text{endif} \\
\text{endlet} \\
\text{endchoose}
\]

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.

\[
\text{CREATESTATEREFINEMENT} = \\
\text{choose snr: snr } \in \text{Self.stateNodesToBeRefined} \\
\text{Self.stateNodesToBeRefined } := \text{Self.stateNodesToBeRefined } \setminus \{\text{snr}\} \\
\text{Self.currentParentStateNode } := \text{snr} \\
\text{if snr.nodeAS1 } \in \text{Procedure-graph then} \\
\text{CREATEPROCEDUREVARIABLES(Self, snr, snr.nodeAS1)} \\
\text{CREATEPROCEDUREGRAPHNODES(snr, snr.nodeAS1)} \\
\text{elseif snr.nodeAS1.s-Composite-state-type-identifier } \neq \text{undefined then} \\
\text{CREATECOMPOSITESTATEVARIABLES(Self, snr, snr.nodeAS1.s-Composite-state-type-identifier)} \\
\text{CREATECOMPOSITESTATE(snr, snr.nodeAS1.s-Composite-state-type-identifier.idToNodeAS1)} \\
\text{else} \\
\text{snr.stateNodeRefinement } := \text{undefined} \\
\text{endif} \\
\text{endchoose}
\]

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)
elsif 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.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.
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.

```
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
```

**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 =
   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.

![Diagram of SDL agent activity phases: selecting transition (level 3)](image)

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

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.

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.

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.

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.

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.
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.

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.

```plaintext
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
  S ELEXCEPTIONSTARTPHASE
elseif Self.agentMode4 = selectionPhase then
  S ELEXCEPTIONSELECTIONPHASE
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.

**S ELEXCEPTIONSTARTPHASE ≡**

```
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`.

**S ELEXCEPTIONSELECTIONPHASE ≡**

```
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] )
```
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
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
endif
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
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
endif
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 :=

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)](image)

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.

```
SELECTPRIORITYINPUT ≡
  if Self.agentMode4 = startPhase then
    SELF.PRIORITYINPUTSTARTPHASE
  elseif Self.agentMode4 = selectionPhase then
    SELF.PRIORITYINPUTSELECTIONPHASE
  elseif Self.agentMode4 = selectSpontaneous then
    SELECTSPONTANEOUS
  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.

```
SEL.PRIORITYINPUTSTARTPHASE ≡
  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
```

```
SELF.PRIORITYINPUTSELECTIONPHASE ≡
  if Self.stateNodeChecked = undefined then
    NEXTSTATE_NODE_TO_BE_CHECKED
  elseif Self.Spontaneous then
    Self.agentMode4 := selectSpontaneous
```

```
else
let
\[ t = \text{take}\{ tr \in \text{Self.stateNodeChecked}.\text{stateTransitions}.\text{priorityInputTransitions} : \]
\[ \text{tr.s-SIGNAL} = \text{Self.signalChecked.signalType} \}\} \text{ in} \]
 if \( t \neq \text{undefined} \) then
\[
\text{Self.currentSignalInst} := \text{Self.signalChecked} \\
\text{Self.sender} := \text{Self.signalChecked.signalSender} \\
\text{DELETE(Self.signalChecked, Self.inport)} \\
\text{TRANSITIONFOUND}(t) \\
\text{else} \]
\[
\text{Self.stateNodeChecked} := \text{undefined} \\
\text{endif} \\
\text{endlet} \\
\text{endif} \\
\text{endif} \\
\text{where} \\
\text{NEXTSTATENODETOBECHECKED} \equiv \\
\text{if} \ \text{Self.stateNodesToBeChecked} \neq \emptyset \text{ then} \\
\text{if} \ \text{Self.stateNodeChecked} = \text{undefined} \text{ then} \\
\text{SELECTNEXTSTATENODE} \\
\text{else} \\
\text{CHECKFORINHERITEDSTATENODES} \\
\text{endif} \\
\text{else} \\
\text{NEXTSIGNALTOBECHECKED} \\
\text{Self.stateNodesToBeChecked} := \text{collectCurrentSubStates(\{Self.topStateNode\})} \\
\text{Self.stateNodeChecked} := \text{undefined} \\
\text{endif} \\
\text{SELECTNEXTSTATENODE} = \\
\text{let} \ sn = \text{Self.stateNodesToBeChecked}.\text{selectNextStateNode in} \\
\text{if} \ \text{sn.stateNodeKind} = \text{procedureNode} \text{ then} \\
\text{Self.stateNodesToBeChecked} := \text{Self.stateNodesToBeChecked} \setminus \\
\text{collectCurrentSubStates(\{sn.getPreviousStatePartition\})} \\
\text{else if} \ \text{sn.stateNodeKind} = \text{statePartition} \text{ then} \\
\text{Self.stateNodesToBeChecked} := \text{Self.stateNodesToBeChecked} \setminus \{sn\} \\
\text{else if} \ \text{sn.stateNodeKind} = \text{stateNode} \text{ then} \\
\text{Self.stateNodeChecked} := \text{sn} \\
\text{Self.stateNodesToBeChecked} := \text{Self.stateNodesToBeChecked} \setminus \{sn\} \\
\text{endif} \\
\text{endlet} \\
\text{CHECKFORINHERITEDSTATENODES} = \\
\text{Self.stateNodeChecked} := \text{undefined} \\
\text{let} \ sn1 = \text{Self.stateNodeChecked in} \\
\text{if} \ \text{Self.signalChecked.signalType} \in \\
\{\in \text{Signal-identifier} \mid \text{in} \in \text{sn1.nodeAS1.s-Input-node-set} \} \cup \\
\text{sn1.nodeAS1.s-Save-signalset} \text{ then} \\
\text{Self.stateNodesToBeChecked} := \text{Self.stateNodesToBeChecked} \setminus \\
\{\text{sn2} \in \text{Self.stateNodesToBeChecked} \mid \text{InheritsFrom}(\text{sn1, sn2})\} \\
\text{endif} \\
\text{endlet} \\
\text{NEXTSIGNALTOBECHECKED} = \\
\text{let} \ si = \text{nextSignal(Self.signalChecked, Self.inputPortChecked) in} \\
\text{if} \ si \neq \text{undefined} \text{ then} \\
\text{Self.signalChecked} := \text{si} \\
\text{else} \\
\text{Self.agentMode3} := \text{selectInput} \\
\text{Self.agentMode4} := \text{startPhase} \\
\text{RETURNEXECRIGHT}
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)](image)

```
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})
```

68 ITU-T Z.100/Annex F3 (01/2000)
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
    if t.s-LABEL ≠ undefined then
      EvaluateEnablingCondition(t)
    else
      Self.currentSignalInst := Self.signalChecked
      Self.sender := Self.signalChecked.signalSender
      DELETE(Self.signalChecked.signalSender)
      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.signalSender)
    endif
  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
  else
    self.stateNodeChecked := sn
  endif
```
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.
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 a continuous signal consists of the subphases shown in the state diagram in Figure F3-11. The control is identical to the selection of an ordinary input.

![Activity phases of SDL agents: selecting continuous signals (level 4)](image)

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

```plaintext
SELECTCONTINUOUS ≡
  if Self.agentMode4 = startPhase then
    SELECTCONTINUOSEVALUATIONPHASE
  elseif Self.agentMode4 = selectionPhase then
    SELECTCONTINUOUSELECTIONPHASE
  elseif Self.agentMode4 = evaluationPhase then
    SELECTCONTINUOSELECPHASE
  elseif Self.agentMode4 = selectSpontaneous then
    SELECTCONTINUOSSPONTANEOUS
  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.

```plaintext
SELECTCONTINUOSSSTARTPHASE ≡
  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.

```plaintext
SELECTCONTINUOSSSELECTIONPHASE ≡
  if Self.stateNodeChecked = undefined then
    NEXTSTATENODETOBECHECKED
  elseif Self.Spontaneous then
    Self.agentMode4 := selectSpontaneous
  else
    let t = selectContinuousSignal(Self.transitionsToBeChecked, Self.continuousPriorities) in
```
if \( t \neq \text{undefined} \) then
   \( \text{Self}.\text{transitionsToBeChecked} \leftarrow \text{Self}.\text{transitionsToBeChecked} \setminus \{t\} \)
if \( t.\text{s-LABEL} \neq \text{undefined} \) then
   \( \text{EvaluateEnablingCondition}(t) \)
else
   \( \text{TransitionFound}(t) \)
endif
else
   \( \text{NextStateNodeToBeChecked} \)
endif
endlet
endif

where
\[
\text{EvaluateEnablingCondition}(t; \text{TRANSITION}) \equiv \\
\text{Self}.\text{transitionChecked} \leftarrow t \\
\text{Self}.\text{currentStateId} \leftarrow \text{Self}.\text{stateNodeChecked}.\text{parentStateNode}.\text{stateId} \\
\text{Self}.\text{currentLabel} \leftarrow t.\text{s-LABEL} \\
\text{Self}.\text{agentMode4} \leftarrow \text{evaluationPhase}
\]

\( \text{NextStateNodeToBeChecked} \equiv \\
\begin{cases} \\
\text{if} \ \text{Self}.\text{stateNodesToBeChecked} \neq \emptyset \text{ then} \\
\text{if} \ \text{Self}.\text{stateNodeChecked} = \text{undefined} \text{ then} \\
\text{SelectNextStateNode} \\
\text{else} \\
\text{CheckForInheritedStateNodes} \\
\end{cases} \\
\text{else} \\
\text{Self}.\text{agentMode3} \leftarrow \text{startSelection} \\
\text{RETURNEXECRIGHT}
\]

\( \text{SelectNextStateNode} \equiv \\
\begin{cases} \\
\text{let} \sn = \text{Self}.\text{stateNodesToBeChecked}.\text{selectNextStateNode} \text{ in} \\
\text{if} \sn.\text{stateNodeKind} = \text{procedureNode} \text{ then} \\
\text{Self}.\text{stateNodesToBeChecked} \leftarrow \text{Self}.\text{stateNodesToBeChecked} \setminus \\
\text{collectCurrentSubStates}(\text{sn}.\text{getPreviousStatePartition}) \\
// \text{only state partitions of the state machine to be considered here} \\
\text{elseif} \sn.\text{stateNodeKind} = \text{statePartition} \text{ then} \\
\text{Self}.\text{stateNodesToBeChecked} \leftarrow \text{Self}.\text{stateNodesToBeChecked} \setminus \{\sn\} \\
\text{elseif} \sn.\text{stateNodeKind} = \text{stateNode} \text{ then} \\
\text{Self}.\text{stateNodeChecked} \leftarrow \sn \\
\text{Self}.\text{stateNodesToBeChecked} \leftarrow \text{Self}.\text{stateNodesToBeChecked} \setminus \{\sn\} \\
\text{Self}.\text{transitionsToBeChecked} \leftarrow \text{sn1}.\text{stateTransitions}.\text{continuousSignalTransitions} \\
\text{Self}.\text{continuousPriorities} \leftarrow \emptyset \\
\end{cases} \\
\end{cases}
\]

\( \text{CheckForInheritedStateNodes} \equiv \\
\begin{cases} \\
\text{let} \sn = \text{Self}.\text{stateNodeChecked} \text{ in} \\
\text{let} \sn1 = \text{selectInheritedStateNode}(\sn, \text{Self}.\text{stateNodesToBeChecked}) \text{ in} \\
\text{if} \sn1 \neq \text{undefined} \text{ then} \\
\text{Self}.\text{stateNodesToBeChecked} \leftarrow \text{Self}.\text{stateNodesToBeChecked} \setminus \{\sn1\} \\
\text{Self}.\text{stateNodeChecked} \leftarrow \sn1 \\
\text{Self}.\text{transitionsToBeChecked} \leftarrow \\
\text{sn1}.\text{stateTransitions}.\text{continuousSignalTransitions} \\
\text{Self}.\text{continuousPriorities} \leftarrow \text{Self}.\text{continuousPriorities} \cup \\
\{\text{t.s-NAT} | \text{t} \in \text{sn}.\text{stateTransitions}.\text{continuousSignalTransitions}\} \\
\text{else} \\
\text{Self}.\text{stateNodeChecked} \leftarrow \text{undefined} \\
\end{cases} \\
\end{cases}
\]
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.

\[
\text{SELCONTINUOUSEVALUATIONPHASE} \equiv \\
\text{if } \text{Self.currentLabel} \neq \text{undefined} \text{ then} \\
\quad \text{choose } b : b \in \text{behaviour} \land b.s\text{-LABEL} = \text{Self.currentLabel} \\
\quad \text{EVAL}(b.s\text{-PRIMITIVE}) \\
\text{endchoose} \\
\text{elseif } \text{semvalue(value(Self.transitionChecked.s\text{-LABEL},Self))} \text{ then} \\
\text{TRANITIONFOUND(Self.transitionChecked)} \\
\text{else} \\
\text{Self.agentMode4 := selectionPhase} \\
\text{endif}
\]

For each continuous signal, the continuous expression has to be evaluated. As this evaluation consists of several actions in general, another agent mode, \textit{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)](image)

Since any time the agent mode \textit{selectSpontaneous} is entered, only one spontaneous transition is checked, there are only two sub modes (\textit{agentMode5}) as shown in the diagram in Figure F3-12.

\[
\text{SELECTSPONTANEUS} \equiv \\
\text{if } \text{Self.agentMode5 = selectionPhase} \text{ then} \\
\quad \text{SELSPONTANEOUSSELECTIONPHASE} \\
\text{elseif } \text{Self.agentMode5 = evaluationPhase} \text{ then} \\
\quad \text{SELSPONTANEOUSEVALUATIONPHASE} \\
\text{endif}
\]

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

\[
\text{SELSPONTANEOUSSELECTIONPHASE} \equiv \\
\text{if } \text{Self.stateNodeChecked.stateTransitions.spontaneousTransitions} \neq \emptyset \text{ then} \\
\quad \text{choose } t : t \in \text{Self.stateNodeChecked.stateTransitions.spontaneousTransitions} \\
\quad \text{if } t.s\text{-LABEL} \neq \text{undefined} \text{ then} \\
\quad \quad \text{EVALUATEENABLINGCONDITION}(t)
\]
else
    \text{\small \textit{Self}:sender := Self.self}
    \text{\small TRANSITIONFOUND}(t)
endif
endchoose
endif

where
\text{\small E VALUATEENABLINGCONDITION}(t:\textit{TRANSITION}) \equiv
    \text{\small Self.transitionChecked} := t
    \text{\small Self.currentStateId} := \text{\small Self.stateNodeChecked}.parentStateNode.stateId
    \text{\small Self.currentLabel} := \text{\small t.s-LABEL}
    \text{\small Self.agentMode5 := evaluationPhase}
endwhere

For a given state node, an arbitrary spontaneous transition is selected, and it is checked whether this transition is fireable.

\text{\small SELFSPONTANEouseEVALUATIONPHASE} \equiv
    \text{\small if \text{\small Self}.currentLabel \neq \text{\small undefined} then}
        \text{\small choose } b: b \in \text{\small behaviour} \land b.s-LABEL = \text{\small Self.currentLabel}
        \text{\small EVAL}(b.s-\text{\small PRIMITIVE})
    \text{\small endchoose}
    \text{\small elseif \text{\small semvalue}(value(\text{\small Self}.transitionChecked.s-LABEL,Self)) then}
        \text{\small Self.sender := Self.self}
        \text{\small TRANSITIONFOUND}(\text{\small Self}.transitionChecked)
    \text{\small else}
        \text{\small Self.agentMode4 := selectionPhase}
    \text{\small 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, \textit{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.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{f3-13.png}
\caption{Activity phases of SDL agents: firing transitions (level 3)}
\end{figure}
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.

![Activity phases of SDL agents: entering state node (level 4)](image)

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.

\[
\text{ENTERSTATENODES} \equiv \text{ENTERPHASE} = \begin{cases} 
\text{if } \text{Self} \cdot \text{stateNodesToBeEntered} \neq \emptyset \text{ then} & \begin{cases} 
\text{choose } snwen: \text{snwen} \in \text{Self} \cdot \text{stateNodesToBeEntered} \\
\text{snwen} \cdot \text{STATENODE.currentSubStates} := \emptyset \\
\text{snwen} \cdot \text{STATENODE.currentExitPoints} := \emptyset \\
\text{if } \text{snwen} \cdot \text{STATENODE.parentStateNode} \neq \text{undefined} \text{ then} & \begin{cases} 
\text{snwen} \cdot \text{STATENODE.parentStateNode.currentSubStates} := \text{snwen} \cdot \text{STATENODE.parentStateNode.currentSubStates} \cup \{ \text{snwen} \cdot \text{STATENODE} \}
\end{cases} \\
\end{cases} \\
\end{cases} \\
\text{endif} \\
\text{else} \\
\text{Self} \cdot \text{agentMode4} := \text{enteringFinished} \\
\text{endif}
\]

where

\[
\text{REFINEMENTUNDEF(snwen:STATENODEWITHENTRYPOINT)} \equiv \begin{cases} 
\text{if } \text{snwen} \cdot \text{STATENODE.inheritedStateNode} \neq \text{undefined} \text{ then} & \begin{cases} 
\text{refinement possibly inherited} \\
\text{Self} \cdot \text{stateNodesToBeEntered} := \text{Self} \cdot \text{stateNodesToBeEntered} \setminus \{ \text{snwen} \} \cup \\
\{ \text{mk-STATENODEWITHENTRYPOINT}(\text{snwen} \cdot \text{STATENODE.inheritedStateNode}, \\
\text{snwen} \cdot \text{STATEENTRYPOINT}) \}
\end{cases} \\
\text{else} & \\
\text{Self} \cdot \text{stateNodesToBeEntered} := \text{Self} \cdot \text{stateNodesToBeEntered} \setminus \{ \text{snwen} \}
\end{cases}
\]

\[
\text{REFINEMENTSTATEAGGRNODE(snwen:STATENODEWITHENTRYPOINT)} \equiv \begin{cases} 
\text{Self} \cdot \text{stateNodesToBeEntered} := \text{Self} \cdot \text{stateNodesToBeEntered} \setminus \{ \text{snwen} \} \cup \\
\{ \text{snwen} \in \text{STATENODEWITHENTRYPOINT}: \text{snwen} = \text{mk-STATENODEWITHENTRYPOINT}(sp, \\
\text{entryConnection}(\text{snwen} \cdot \text{STATEENTRYPOINT}, sp)) \land \\
sp \in \text{snwen} \cdot \text{STATENODE.statePartitionSet} \}
\text{let } cstd = \text{snwen} \cdot \text{STATENODE.nodeAS1.s-Composite-state-type-identifier.idToNodeAS1 in} & \\
\text{if } cstd.s-\text{State-aggregation-node.s-Entry-procedure-definition} \neq \text{undefined} \text{ then} & \\
\text{CREATEPROCEDURE(cstd.s-\text{State-aggregation-node.s-Entry-procedure-definition}, } \\
\text{undefined})
\end{cases}
\]

\[
\text{REFINEMENTCOMPSTATENODE(snwen:STATENODEWITHENTRYPOINT)} \equiv \begin{cases} 
\text{Self} \cdot \text{stateNodesToBeEntered} := \text{Self} \cdot \text{stateNodesToBeEntered} \setminus \{ \text{snwen} \} \land \\
\text{let } cstd = \text{snwen} \cdot \text{STATENODE.nodeAS1.s-Composite-state-type-identifier.idToNodeAS1 in} & \\
\text{if } cstd.s-\text{Composite-state-graph.s-Entry-procedure-definition} \neq \text{undefined} \text{ then} & \\
\text{CREATEPROCEDURE(cstd.s-\text{Composite-state-graph.s-Entry-procedure-definition}, } \\
\text{undefined})
\end{cases}
\]

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.

\[
\text{ENTERSTATENODESENTERINGFINISHED} \equiv \\
\text{Self.agentMode2 := selectingTransition} \\
\text{Self.agentMode3 := startSelection} \\
\text{RETURNEXECRIGHT}
\]

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

\[
\text{LEAVESTATENODES} \equiv \\
\text{if Self.agentMode4 = leavePhase then} \\
\text{LEAVESTATENODESLEAVEPHASE} \\
\text{elseif Self.agentMode4 = leavingFinished then} \\
\text{LEAVESTATENODESLEAVINGFINISHED} \\
\text{endif}
\]

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.

![Activity phases of SDL agents: leaving state node (level 4)](image)

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

\section*{LEAVESTATENODESLEAVINGFINISHED}

\begin{verbatim}
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
\end{verbatim}

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.

\subsection*{2.3.2.18 Exiting of Composite States}

\section*{EXITCOMPOSITESTATE}

\begin{verbatim}
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
endif
else
    Self.currentExitStateNodes := ∅
    Self.agentMode2 := selectingTransition
    Self.agentMode3 := startPhase
endif
\end{verbatim}

\subsection*{2.3.2.19 Stopping Agent Execution}

An agent ceases to exist as soon as all contained agents have been removed.

\section*{STOPPHASE}

\begin{verbatim}
if ∀sas ∈ SDLAGENTSET: (sas.owner = Self ⇒ ¬∃sa ∈ SDLAGENT: sa.owner = sas) then
\end{verbatim}
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: DefinitionASI × 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.

\[
\text{functional}(\text{procedure}, \text{values}) = \text{def} \\
\text{procedure}.s\text{-Qualifier}.head \in \text{Package-qualifier} \\
\land \text{procedure}.s\text{-Qualifier}.head.s\text{-Package-name}.s\text{-TOKEN} = \text{“predefined”}
\]

\[
\text{intype}(\text{procedure}; \text{PROCEDURE}, \text{name}; \text{NAME}); \text{BOOLEAN} = \text{def} \\
\text{procedure}.s\text{-Qualifier}.last.s\text{-Data-type-name} = \text{name}
\]

\[
\text{compute}(\text{procedure}, \text{values }) = \text{def} \\
\text{if intype (procedure, IntegerType) then computeInteger(procedure, values)} \\
\text{elseif intype (procedure, BooleanType) then computeBoolean(procedure, values)} \\
\text{elseif intype (procedure, CharacterType) then computeChar(procedure, values)} \\
\text{elseif intype (procedure, RealType) then computeReal(procedure, values)} \\
\text{elseif intype (procedure, DurationType) then computeDuration(procedure, values)} \\
\text{elseif intype (procedure, TimeType) then computeTime(procedure, values)} \\
\text{elseif intype (procedure, Stringtype) then computeString(procedure, values)} \\
\text{elseif intype (procedure, Arraytype) then computeArray(procedure, values)} \\
\text{elseif intype (procedure, Powersettype) then computePowerset(procedure, values)} \\
\text{elseif intype (procedure, Bagtype) then computeBag(procedure, values)} \\
\text{else undefined}
\]
computeSideEffects(procedure, values, state, id) =def
if isstructureoperator(procedure) then
  structComputeObjects(procedure, values, id)
else
  objectvalues
endif

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

\[ \text{matchingoperator} : \text{PROCEDURE} \times \text{NAME} \times \text{IDENTIFIER}^* \rightarrow \text{BOOLEAN} \]

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

\[ \text{compareTail}(t1 : \text{TOKEN}, t2 : \text{TOKEN}) : \text{TOKEN} = \text{def} \]
\[ \begin{align*}
&\text{if } t1.\text{length} < t2.\text{length} \text{ then } \text{False} \\
&\text{elseif } t1.\text{length} > t2.\text{length} \text{ then } \text{compareTail}(t1.\text{tail}, t2) \\
&\text{else } t1 = t2
\end{align*} \]

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

\[ \text{definingSort}(p : \text{PROCEDURE}) : \text{Identifier} = \text{def} \]
\[ p.\text{parentAS1}.\text{nodeAS1ToId} \]

The function procName computes the token of an operator.

\[ \text{procName}(p : \text{PROCEDURE}) : \text{TOKEN} = \text{def} \]
\[ p.\text{s-Name}.s-TOKEN \]

### 3.1.1 Well-known definitions

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

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

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

- **OutOfRange**: Identifier =def mk-Identifier(<mk-Package-qualifier("predefined")>,"OutOfRange")
- **InvalidReference**: Identifier =def mk-Identifier(<mk-Package-qualifier("predefined")>,"InvalidReference")
- **UndefinedVariable**: Identifier =def mk-Identifier(<mk-Package-qualifier("predefined")>,"UndefinedVariable")
- **UndefinedField**: Identifier =def mk-Identifier(<mk-Package-qualifier("predefined")>,"UndefinedField")
- **InvalidIndex**: Identifier =def mk-Identifier(<mk-Package-qualifier("predefined")>,"InvalidIndex")
- **DivisionByZero**: Identifier =def mk-Identifier(<mk-Package-qualifier("predefined")>,"DivisionByZero")
- **Empty**: Identifier =def mk-Identifier(<mk-Package-qualifier("predefined")>,"Empty")
To raise an exception, the function \texttt{raise} can be used. The predefined exceptions are distinguished from normal return values as they are Identifiers.

\[
\text{raise}(\texttt{ex}) : \text{Identifier} = \text{def} \quad \text{ex}
\]

The following operation signatures are also well-known:

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

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

\[
\text{sdlTrue: Literal-signature} = \text{def} \\
\text{mk-Literal-signature}((<\text{mk-Package-qualifier("predefined")}, <\text{mk-Data-type-qualifier("Boolean")}>, \text{"True"}, \text{BooleanType}, \text{undefined}>)
\]

\subsection{3.1.2 Boolean}

The function \texttt{computeBoolean} determines the value of an application of a predefined Boolean operator.

\[
\text{SDLBOOLEAN} = \text{def} \text{BOOLEAN} \times \text{Identifier}
\]

\texttt{computeBoolean(procedure: \text{PROCEDURE}, values: \text{VALUE}*): \text{VALUE} = \text{def}}

\[
\text{let restype} = \text{definingSort(procedure) in}
\text{if matchingoperator(procedure, \"not\", \text{BooleanType}) then}
\text{mk-SDLBOOLEAN}((\neg\text{values.head.semvalue}, \text{restype})
\text{elseif matchingoperator(procedure, \"and\", <\text{BooleanType}, \text{BooleanType}>) then}
\text{mk-SDLBOOLEAN}\left(\text{values.head.semvalue} \land \text{values.tail.head.semvalue}, \text{restype}\right)
\text{elseif matchingoperator(procedure, \"or\", <\text{BooleanType}, \text{BooleanType}>) then}
\text{mk-SDLBOOLEAN}\left(\text{values.head.semvalue} \lor \text{value.tail.head.semvalue}, \text{restype}\right)
\text{elseif matchingoperator(procedure, \"xor\", <\text{BooleanType}, \text{BooleanType}>) then}
\text{mk-SDLBOOLEAN}\left(\neg(\text{values.head.semvalue} !\text{values.tail.head.semvalue}), \text{restype}\right)
\text{elseif matchingoperator(procedure, \"=>\", <\text{BooleanType}, \text{BooleanType}>) then}
\text{mk-SDLBOOLEAN}\left(\text{values.head.semvalue} \Rightarrow \text{values.tail.head.semvalue}, \text{restype}\right)
\text{endlet}
\]

\[
\text{semvalue (v:SDLBOOLEAN): \text{BOOLEAN} = def v.s-BOOLEAN}
\]

\subsection{3.1.3 Integer}

\[
\text{SDLINTEGER} = \text{def} \text{NAT} \times \text{Identifier}
\]

\texttt{computeInteger (procedure: \text{PROCEDURE}, values: \text{VALUE}*): \text{VALUE} = \text{def}}

\[
\text{let restype} = \text{definingSort(procedure) in}
\text{if procedure \in \text{Literal} then}
\text{integerLiteral(0, procedure.spelling, restype)}
\text{elseif procedure.s-Name = \"-\" \land values.length = 1 then}
\text{mk-SDLINTEGER}(0 - \text{values.head.s-NAT}, \text{restype})
\text{elseif procedure, s-Name \in \{\"+\", \"-\", \"*\", \"/\", \"mod\", \"rem\", \"<\", \">\", \"<=\", \">=\", \"power\}\ then}
\text{let val1 = values[1].s-NAT, val2 = values[2].s-NAT in}
\text{case procedure.procName in}
\text{\"+\": mk-SDLINTEGER (val1+val2, restype)}
\text{\"-\": mk-SDLINTEGER (val1 - val2, restype)}
\text{\"*\": mk-SDLINTEGER (val1 * val2, restype)}
\text{\"\ haciendo el \}: 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 \texttt{NAT} associated with a single character in the range “0” to “9”.

\begin{verbatim}
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
\end{verbatim}

The function \texttt{integerLiteral} returns the \texttt{SDLINTEGER} value for a real literal.

\begin{verbatim}
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
\end{verbatim}

The function \texttt{intDiv} returns the result of integer-dividing its arguments.

\begin{verbatim}
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)
\end{verbatim}
The function `intMod` returns the result of the integer-modulo operation.

\[
\text{intMod}(a, b) = \begin{cases} 
  \text{intRem}(a, b) & \text{if } a \geq 0 \land b > 0 \\
  \text{intMod}(a, -b) & \text{if } b < 0 \\
  \text{intRem}(a, b) & \text{if } a < 0 \land b > 0 \land \text{intRem}(a, b) = 0 \\
  b + \text{intRem}(a, b) & \text{if } a < 0 \land b > 0 \land \text{intRem}(a, b) < 0
\end{cases}
\]

The function `intRem` returns the result of the integer-remainder operation.

\[
\text{intRem}(a, b) = a - b \times \text{intDiv}(a, b)
\]

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

\[
\text{intPower}(a, b) = \begin{cases} 
  1 & \text{if } b = 0 \\
  a \times \text{intPower}(a, b-1) & \text{if } b > 0 \\
  \text{intDiv}(\text{intPower}(a, b+1), b) & \text{else}
\end{cases}
\]

### 3.1.4 Character

Character values are represented by their name.

\[
\text{SDLCHARACTER} = \text{NAME} \times \text{Identifier}
\]

\[
\text{computeChar } (\text{procedure: PROCEDURE, values: VALUE}^\ast) : \text{VALUE}^\ast = \begin{cases} 
  \text{mk-SDLCHARACTER}(\text{procedure}.\text{procName}, \text{restype}) & \text{if } \text{procedure} \in \text{Literal} \\
  \text{mk-SDLCHARACTER}(\text{charValue}(\text{values}.\text{head}.\text{s-Name}), \text{restype}) & \text{if } \text{procedure}.\text{procName} = \text{“num”} \\
  \text{mk-SDLCHARACTER}(\text{charChr}(\text{intvalue}(\text{values}.\text{head})), \text{restype}) & \text{if } \text{procedure}.\text{procName} = \text{“chr”}
\end{cases}
\]

The function `charvalue` returns the numeral value of the character.

\[
\text{charValue}(\text{ch: Name}) : \text{Name} = \begin{cases} 
  \text{take}(\{L.s-\text{Constant-expression} \mid L \in \text{literals}: L.s-\text{Literal-name} = \text{ch}\}) & \text{let literals = CharacterType.s-Literal-signature-set in}
\end{cases}
\]

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

\[
\text{charChr}(a) : \text{Name} = \begin{cases} 
  \text{charChr}(a-128) & \text{if } a > 128 \\
  \text{charChr}(a+128) & \text{if } a < 0 \\
  \text{mk-SDLINTEGER}(a, \text{IntegerType}) & \text{let literals = CharacterType.s-Literal-signature-set, result = \text{mk-SDLINTEGER}(a, \text{IntegerType}) in}
\end{cases}
\]

### 3.1.5 Real

The predefined type Real is represented as a rational number, with numerator and denominator.
\[ \text{SDLREAL} \triangleq \text{NAT} \times \text{NAT} \times \text{Identifier} \]

\[
\text{compu} \text{teReal} (\text{procedure}: \text{PROCEDURE}, \text{values}: \text{VALUE*}): \text{VALUE} \triangleq \text{def}
\]

\[
\begin{align*}
\text{let} & \quad \text{restype} = \text{definingSort(\text{procedure})} \text{ in} \\
\text{if} & \quad \text{procedure} \in \text{Literal} \text{ then} \\
& \quad \text{realLiteral}(0, 1, \text{procedure.spelling}, \text{restype}) \\
\text{elseif} & \quad \text{procedure.procName} = "-" \land \text{values.length} = 1 \text{ then} \\
& \quad \text{mk-SDLREAL}(0 - \text{values.head.s-NAT}, \text{values.head.s2-NAT}) \\
\text{elseif} & \quad \text{procedure.procName} \in \{ +, -, \times, \div, \min, \max, \lt, \gt, \leq, \geq, \lt=, \gt= \} \text{ then} \\
& \quad \text{let} \quad \text{num1} = \text{values}[1].\text{s-NAT}, \text{den1} = \text{values}[1].\text{s2-NAT}, \\
& \quad \text{num2} = \text{values}[2].\text{s-NAT}, \text{den2} = \text{values}[2].\text{s2-NAT} \text{ in} \\
& \quad \text{case} \quad \text{procedure.procName} \quad \text{in} \\
& \quad \quad \text{"+"} \Rightarrow \text{mk-SDLREAL(num1*den2 + num2*den1, den1*den2, restype)} \\
& \quad \quad \text{"-"} \Rightarrow \text{mk-SDLREAL(num1*den2 - num2*den1, den1*den2, restype)} \\
& \quad \quad \text{"\times"} \Rightarrow \text{mk-SDLREAL(num1*den2, den1*den2, restype)} \\
& \quad \quad \text{"\div"} \Rightarrow \text{mk-SDLREAL(num1*den2, den1*den2, restype)} \\
& \quad \text{endcase} \\
\text{else} & \quad \text{\text{mk-SDLREAL(num1*num2, den1*den2, restype)}} \\
\text{endif} \\
\text{elseif} & \quad \text{procedure} = \text{"float"} \text{ then} \\
& \quad \text{\text{mk-SDLREAL(intvalue(values.head), 1, restype)}} \\
\text{elseif} & \quad \text{procedure} = \text{"fix"} \text{ then} \\
& \quad \text{\text{mk-SDLInt(computeFix(values.head.s-NAT, values.head.s2-NAT), IntegerType)}}
\end{align*}
\]

The function \text{realLiteral} returns the SDLREAL value for a real literal.

\[
\text{realLiteral(num: \text{NAT}, den: \text{NAT}, proc: \text{Name}, type: \text{Identifier})}: \text{SDLREAL} \triangleq \text{def}
\]

\[
\begin{align*}
\text{if} & \quad \text{proc} = \text{empty} \text{ then} \\
& \quad \text{mk-SDLREAL(num, den, type)} \\
\text{elseif} & \quad \text{proc.head} = \text{"."} \text{ then} \\
& \quad \text{realLiteral(num*10, den*10, proc.tail)} \\
\text{elseif} & \quad \text{den} = 1 \text{ then} \\
& \quad \text{realLiteral(num*10 + numberValue(proc.head), den, type)} \\
\text{else} & \quad \text{realLiteral(num*10 + numberValue(proc.head), den, type)}
\end{align*}
\]

The function \text{computeFix} returns the NAT value given numerator and denominator.

\[
\text{computeFix(num: \text{NAT}, den: \text{NAT})}: \text{NAT} \triangleq \text{def}
\]

\[
\begin{align*}
\text{if} & \quad \text{num} < 0 \text{ then} \\
& \quad - \text{computeFix(- num, den)} - 1 \\
\text{elseif} & \quad \text{num} < \text{den} \text{ then} \\
& \quad 0 \\
\text{else} & \quad \text{\text{computeFix (num - den, den) + 1}}
\end{align*}
\]

### 3.1.6 Duration

The domain \text{SDLDuration} is based on the domain \text{SDLREAL}.

\[
\text{SDLDuration} \triangleq \text{def} \text{SDLREAL}
\]

\[
\text{computeDuration (procedure: \text{PROCEDURE}, values: \text{VALUE*})}: \text{VALUE} \triangleq \text{def}
\]

\[
\text{computeReal (procedure: \text{PROCEDURE}, values: \text{VALUE*})}
\]

84 ITU-T Z.100/Annex F3 (01/2000)
3.1.7 Time

The domain SDLTIME is based on the domain SDLREAL.

\[ \text{SDLTIME} \triangleq \text{SDLREAL} \]

\[ \text{computeTime}(\text{procedure} : \text{PROCEDURE}, \text{values} : \text{VALUE}*): \text{VALUE} = \text{def} \]

\[ \text{let restype} = \text{definingSort}(\text{procedure}) \text{ in} \]

\[ \text{if procedure} \in \text{Literal} \text{ then} \]

\[ \text{realLiteral}(0.1, \text{procedure.spelling}, \text{restype}) \]

\[ \text{case procedure.procName in} \]

| “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) endif |

3.1.8 String

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

\[ \text{SDLSTRING} \triangleq \text{VALUE} * \times \text{Identifier} \]

\[ \text{computeString}(\text{procedure} : \text{PROCEDURE}, \text{values} : \text{VALUE}*): \text{VALUE} = \text{def} \]

\[ \text{let restype} = \text{definingSort}(\text{procedure}) \text{ in} \]

\[ \text{case procedure.procName in} \]

| “emptystring” => mk-SDLSTRING(\text{<}, 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 |

\[ \text{if index} < 0 \lor \text{index} > \text{string.length then} \]

\[ \text{raise(InvalidIndex)} \]

\[ \text{else} \]

\[ \text{string[index]} \]

| “modify” => let index = values[2].s-NAT in |

\[ \text{let val = substr(values[1], s-VALUE-seq, 1, index) ∩ values[3]} \]

\[ \text{substr(values[1].s-VALUE-seq, index+1, values[1].s-VALUE-seq.length - index in)} \]

\[ \text{if InvalidIndex in val then raise(InvalidIndex)} \]

\[ \text{else} \]

\[ \text{mk-SDLSTRING(val, restype)} \]

endif |

\[ “substring” => \text{let val = substr(values[1], values[2].s-NAT, values[3].s-NAT in)} \]

\[ \text{if InvalidIndex in val then raise(InvalidIndex)} \]
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 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) >
else 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)
else name = “extract” then
let v = take({, 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
endif
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 =_{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] ≤ values[2], s-VALUE-set, BooleanType)
| “<” => mk-SDLBOOLEAN (values[1] < values[2], s-VALUE-set, BooleanType)
| “and” => mk-SDLBOOLEAN (values[1] ∩ values[2], s-VALUE-set, BooleanType)
| “or” => mk-SDLBOOLEAN (values[1] ∪ values[2], s-VALUE-set, restype)
| “length” => mk-SDLINTEGER (0 ≤ |values[1]| ≤ IntegerType)
| “take” => if values[1].s-VALUE-set = ∅ then raise(Empty)
else values[1]. s-VALUE-set.take
endcase

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

SDLBAG =_{def} FREQUENCY-set × Identifier

FREQUENCY =_{def} VALUE × NAT

computeBag (procedure: PROCEDURE, values: VALUE*): VALUE =_{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 (baginc(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[1], values[2]), BooleanType)
| “and” => mk-SDLBOOLEAN (¬ baginbag(values[2], values[1]), BooleanType)
| “or” => mk-SDLBOOLEAN (baginbag(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 =_{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

baginc(item: VALUE, bag: SDLBAG): FREQUENCY-set =_{def}
if bagcount(item, bag) ≠ 0 then
| if elem.s-VALUE = item then mk-FREQUENCY(item, elem.s-NAT+1) else elem endif
| elem ∈ bag.s-FREQUENCY-set)
else
bag.s-FREQUENCY-set ∪ {mk-FREQUENCY (item, 1)}

bagdel(item: VALUE, bag: SDLBAG): FREQUENCY-set =_{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
|
\[ \text{bag} \cdot \text{s-FREQUENCY-set} \setminus \{ \text{mk-FREQUENCY(item, 1)} \} \]

\[ \forall \text{elem} \in \text{smaller.s-FREQUENCY-set}: \text{bagcount}((\text{elem}.\text{s-VALUE}, \text{larger}) < \text{elem}.\text{s-NAT}) \]

\[ \text{baginbag}(\text{smaller}: \text{SDLBAG}, \text{larger}: \text{SDLBAG}): \text{BOOLEAN} = \text{def} \]

\[ \text{bagand}(a: \text{SDLBAG}, b: \text{SDLBAG}): \text{FREQUENCY-set} = \text{def} \]

\[ \text{bagor}(a: \text{SDLBAG}, b: \text{SDLBAG}): \text{FREQUENCY-set} = \text{def} \]

\[ \text{baglength}(a: \text{SDLBAG}): \text{NAT} = \text{def} \]

### 3.2 Pid Types

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

\[ \text{PID} = \text{def} (\text{SDLAGENT} \times \text{Interface-definition}) \cup \{ \text{null} \} \]

\[ \text{static null}: \rightarrow X \]

### 3.3 Constructed Types

#### 3.3.1 Structures

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

\[ \text{SDLSTRUCTURE} = \text{def} \text{FIELD-set} \times \text{Identifier} \]

\[ \text{FIELD} = \text{def} \text{Name} \times \text{VALUE} \]

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

\[ \text{specialName}(\text{procedure}: \text{PROCEDURE}, \text{suffix}: \text{TOKEN}): \text{BOOLEAN} = \text{def} \]

\[ \text{structExtract}(\text{fieldname}: \text{Name}, \text{fields}: \text{SDLSTRUCTURE}): \text{VALUE} = \text{def} \]

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

\[ \text{structExtract}(\text{fieldname}: \text{Name}, \text{fields}: \text{SDLSTRUCTURE}): \text{VALUE} = \text{def} \]

---

88 ITU-T Z.100/Annex F3 (01/2000)
The function \texttt{structModify} returns a new structure with one field changed.

\begin{verbatim}
structModify(fieldname: Name, fields: SDLSTRUCTURE, value: VALUE): SDLSTRUCTURE = _def
{ f.s-VALUE | f ∈ fields: f.s-Name ≠ fieldname} ⊕
{ mk-Field(fieldname, value) }
\end{verbatim}

\begin{verbatim}
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
else 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
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}

### 3.3.2 Literals

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

\begin{verbatim}
SDLLITERALS =_def Literal-signature × Identifier
\end{verbatim}

\begin{verbatim}
computeLiteral(procedure:PROCEDURE, values:VALUE*): VALUE = _def
let restype = definingSort(procedure) in
define 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
\end{verbatim}
literalMinimum(s: Literal-signature-set): Constant-expression = def
take( \{ s \in s \land \forall s2 \in s: s2.s-Constant-expression > s.s-Constant-expression \})

literalMaximum(s: Literal-signature-set): Constant-expression = def
take( \{ s \in s \land \forall s2 \in s: s2.s-Constant-expression > s.s-Constant-expression \})

literalSucc(s: Literal-signature-set, val: Literal-signature): Constant-expression = def
take( \{ s1 \in s \land s1.s-Constant-expression = val.s-Constant-expression.semvalue + 1 \})

literalPred(s: Literal-signature-set, val: Literal-signature): Constant-expression = def
take( \{ s1 \in s \land 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.

\[ \text{OBJECT} = \text{OBJECTIDENTIFIER} \]

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

\[ \text{OBJECTVALUE} = \text{OBJECT} \times \text{VALUE} \]

The function objectValue retrieves the value associated with an object:

\[ \text{objectValue}(\text{state: STATE}, \text{object: OBJECT}): \text{VALUE} = \text{def} \]
\[ \{ o.s-VALUE | o \in \text{state.s-ObjectValue-set}: o.s-OBJECT = \text{object} \} . \text{take} \]

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

\[ \text{modifyObject}(\text{state: STATE}, \text{object: OBJECT}, \text{value: VALUE}): \text{STATE} = \text{def} \]
\[ \text{let newsub = initDeclarations(newid, declarations), newsuper = mk-SUPERSTATE(id, newid) in} \]
\[ \text{mk-STATE}(\text{state.s-NAMEDVALUE-set} \cup \text{newsub}, \text{state.s-SuperState-set} \cup \text{newsuper, state.s-OBJECTVALUE-set}) \]

3.5 State Access

The domain \text{STATE} consists of sub states (associations of values for a specific \text{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.

\[ \text{STATE} = \text{NAMEDVALUE-set} \times \text{SUPERSTATE-set} \times \text{OBJECTVALUE-set} \]

\[ \text{NAMEDVALUE} = \text{def} \text{STATEID} \times \text{Variable-identifier} \times \text{BOUNDVALUE} \]

\[ \text{BOUNDVALUE} = \text{def} \text{VALUE} \cup \text{Variable-identifier} \]

\[ \text{SUPERSTATE} = \text{def} \text{STATEID} \times \text{STATEID} \]

\[ \text{initAgentState(state: STATE, newid: STATEID, id: STATEID, declarations: DECLARATION): STATE} = \text{def} \]
\[ \text{let newsub = initDeclarations(newid, declarations), newsuper = mk-SUPERSTATE(id, newid) in} \]
\[ \text{mk-STATE}(\text{state.s-NAMEDVALUE-set} \cup \text{newsub}, \text{state.s-SUPERSTATE-set} \cup \text{newsuper, state.s-OBJECTVALUE-set}) \]

\[ \text{initProcedureState(state: STATE, newid: STATEID, id: STATEID, declarations: DECLARATION-set,} \]
The function \( \text{setValue} \) puts a sequence of parameter values into a named values set for a given state id.

\[
\text{setValue}(\text{namedvalues}: \text{NAMEDVALUE-set}, \text{id}: \text{STATEID}, \text{declarations}: \text{DECLARATION*}, \text{values}: \text{VALUE*}, \text{variables}: \text{Variable-definition*}) = \text{def}
\]

\[
\text{if values = empty} \quad \emptyset
\]

\[
\text{else}
\]

\[
\text{let varname = decls.head.s-Parameter.s-Variable-name in}
\]

\[
\text{case decls.head.s-PARAMETERKIND in}
\]

\[
\text{in} = \Rightarrow \text{setValue}(\text{namedvalues}, \text{varname}, \text{values.head}, \text{decls.tail}, \text{values.tail}, \text{variables.tail})
\]

\[
\text{out} = \Rightarrow \text{setValue}(\text{namedvalues}, \text{decls.tail}, \text{values.tail}, \text{variables.tail})
\]

\[
\text{inout} = \Rightarrow \text{setValue}(\text{namedvalues}, \text{varname}, \text{values.head})
\]

\[
\text{endcase}
\]

\[
\text{endlet}
\]

The function \( \text{update} \) modifies a binding of a name to a value.

\[
\text{update}(\text{name}: \text{Identifier}, \text{value}: \text{VALUE}, \text{state}: \text{STATE}, \text{id}: \text{STATEID}) = \text{def}
\]

\[
\text{let val = getValue(state.s-NAMEDVALUE-set, id, name) in}
\]

\[
\text{if val = empty} \quad \text{empty}
\]

\[
\text{elseif val.take \in VALUE} \quad \text{val.take}
\]

\[
\text{else}
\]

\[
\text{value(val, state, caller(state, id))}
\]

The function \( \text{getValue} \) returns the association between \text{id} and \text{varname} in \text{namedvalues}.

\[
\text{getValue}(\text{namedvalues}: \text{NAMEDVALUE-set}, \text{id}: \text{STATEID}, \text{varname}: \text{Identifier}) = \text{def}
\]

\[
\text{if b.s-BOUNDVALUE | b \in namedvalues:}
\]

\[
\text{b.s-STATEID = id} \land b.s-Variable-identifier = varname
\]

The function \( \text{eval} \) 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.

\[
\text{eval}(\text{name}: \text{Identifier}, \text{state}: \text{STATE}, \text{id}: \text{STATEID}): \text{VALUE} = \text{def}
\]

\[
\text{let val = getValue(state.s-NAMEDVALUE-set, id, name) in}
\]

\[
\text{if val = empty} \quad \text{empty}
\]

\[
\text{elseif val.take \in VALUE} \quad \text{val.take}
\]

\[
\text{else}
\]

\[
\text{value(val, state, caller(state, id))}
\]

The function \( \text{assignValues} \) puts a sequence of parameter values into a named values set for a given state id.

\[
\text{assignValues}(\text{namedvalues}: \text{NAMEDVALUE-set, id}: \text{STATEID}, \text{declarations}: \text{DECLARATION*}, \text{values}: \text{VALUE*}, \text{variables}: \text{Variable-definition*}) = \text{def}
\]

\[
\text{if values = empty} \quad \emptyset
\]

\[
\text{else}
\]

\[
\text{let varname = decls.head.s-Parameter.s-Variable-name in}
\]

\[
\text{case decls.head.s-PARAMETERKIND in}
\]

\[
\text{in} = \Rightarrow \text{assignValues}(\text{setValue}(\text{namedvalues}, \text{varname}, \text{values.head}, \text{decls.tail}, \text{values.tail}, \text{variables.tail}))
\]

\[
\text{out} = \Rightarrow \text{assignValues}(\text{namedvalues}, \text{decls.tail}, \text{values.tail}, \text{variables.tail})
\]

\[
\text{inout} = \Rightarrow \text{assignValues}(\text{setValue}(\text{namedvalues}, \text{varname}, \text{values.head}))
\]

\[
\text{endcase}
\]

\[
\text{endlet}
\]
else if \text{val}.\text{take} \in \text{VALUE} then
\begin{verbatim}
  \text{mk-STATE}(\text{setValue}(\text{state}.\text{s-NAMEDVALUE-set}, \text{id}, \text{name}, \text{value}),
  \text{state}.\text{s-SUPERSTATE-set}, \text{state}.\text{s-OBJECTVALUE-set})
\end{verbatim}
\end{verbatim}
else
\begin{verbatim}
  \text{update}(\text{val}.\text{take}, \text{value}, \text{state}, \text{id})
\end{verbatim}
\end{verbatim}

The function \text{assign} modifies the variable with the given name in the state/id association to the given value.

\begin{verbatim}
\text{assign}(\text{variablename}: \text{Variable-identifier}, \text{value}: \text{VALUE}, \text{state}: \text{STATE}, \text{id}: \text{STATEID}) \equiv \
\text{if isObjectVariable(\text{variablename}) then} \\
\begin{verbatim}
  \text{if value} \in \text{OBJECT} then \\
  \begin{verbatim}
    \text{if isCompatibleTo1}(\text{variable}.\text{variableSort}, \text{value}.\text{valuesort}) then \\
    \begin{verbatim}
      \text{update}(\text{variablename}, \text{value}, \text{state}, \text{id})
    \end{verbatim}
  \end{verbatim}
  \begin{verbatim}
    \text{else} \\
    \begin{verbatim}
      \text{update}(\text{variablename}, \text{variable}.\text{variabletype}.\text{nullvalue}, \text{state}, \text{id})
    \end{verbatim}
  \end{verbatim}
\end{verbatim}
\begin{verbatim}
\text{endif}
\end{verbatim}
\begin{verbatim}
\text{else} \\
\begin{verbatim}
  \text{if isValueVariable(\text{variablename}) then} \\
  \begin{verbatim}
    \text{if value} \in \text{OBJECT} then \\
    \begin{verbatim}
      \text{if value} = \text{null} then \text{raise(InvalidReference)} \\
      \begin{verbatim}
        \text{else} \\
        \begin{verbatim}
          \text{update}(\text{variablename}, \text{value}.\text{value}, \text{state}, \text{id})
        \end{verbatim}
      \end{verbatim}
    \end{verbatim}
  \end{verbatim}
  \begin{verbatim}
    \text{else} \\
    \begin{verbatim}
      \text{update}(\text{variablename}, \text{value}, \text{state}, \text{id})
    \end{verbatim}
  \end{verbatim}
\end{verbatim}
\begin{verbatim}
\text{else} \\
\begin{verbatim}
  // Pid variable \\
  \text{if compatiblepid(\text{variable}.\text{variablesort}, \text{value}.\text{interface}) then} \\
  \begin{verbatim}
    \text{update}(\text{variablename}, \text{value}, \text{state}, \text{id})
  \end{verbatim}
  \begin{verbatim}
    \text{else} \\
    \begin{verbatim}
      \text{update}(\text{variablename}, \text{null}, \text{state}, \text{id})
    \end{verbatim}
  \end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}
\end{verbatim}

The function \text{caller} returns the state id that caused this state id to exist.

\begin{verbatim}
\text{caller}(\text{state}: \text{STATE}, \text{id}: \text{STATEID}) \equiv \
\begin{verbatim}
\text{take}(\{ \text{s}.\text{s-STATEID} \mid \text{s} \in \text{state}.\text{s-SUPERSTATE}: \text{s}.\text{s2-STATEID} = \text{id} \})
\end{verbatim}
\end{verbatim}

The function \text{variableSort} returns the sort for a given variable identifier.

\begin{verbatim}
\text{variableSort}(\text{variablename}: \text{Variable-identifier}): \text{Data-type-definition} \equiv \
\text{variablename}.\text{idToNodeAS1}.\text{s-Sort-reference-identifier}.\text{idToNodeAS1}
\end{verbatim}

The predicate \text{isValueVariable} holds if the \text{variablename} refers to a variable of a value type.

\begin{verbatim}
\text{isValueVariable}(\text{variablename}: \text{Variable-identifier}): \text{BOOLEAN} \equiv \
\text{variablename}.\text{variableSort} \in \text{Value-data-type-definition}
\end{verbatim}

The predicate \text{isObjectVariable} holds if the \text{variablename} refers to a variable of a object type.

\begin{verbatim}
\text{isObjectVariable}(\text{variablename}: \text{Variable-identifier}): \text{BOOLEAN} \equiv \
\text{variablename}.\text{variableSort} \in \text{Object-data-type-definition}
\end{verbatim}

The predicates \text{assignCopies} and \text{assignClones} determine whether an assignment will invoke assign or clone, respectively.

\begin{verbatim}
\text{assignCopies}(\text{variablename}: \text{Variable-identifier}, \text{value}: \text{VALUE}): \text{BOOLEAN} \equiv \
\text{isValueVariable(\text{variablename})}
\end{verbatim}
\begin{verbatim}
\text{assignClones}(\text{variablename}: \text{Variable-identifier}, \text{value}: \text{VALUE}): \text{BOOLEAN} \equiv \
\text{isObjectVariable(\text{variablename})} \land \text{value}.\text{type} \in \text{Value-data-type-definition}
\end{verbatim}

### 3.6 Specialisation

The function \text{dynamicType} determines the dynamic type of a value.

\begin{verbatim}
\text{dynamicType}(v: \text{VALUE}): \text{Identifier} \equiv \
\begin{verbatim}
\end{verbatim}
\end{verbatim}
case v in
| SDLBOOLEAN (*, t)   => t  
| SDLINTEGER (*, t)  => t  
| SDLCHARACTER(*, t) => t  
| SDLREAL(*, *, t)   => t  
| SDSLSTRING(*, t)   => t  
| SDSLITERALS(*, 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.

\[
\text{dispatch}(\text{procedure}:\text{PROCEDURE}, \text{values}:\text{VALUE}^*) : \text{Identifier} = \text{def}
\]

if procedure ∈ Static-operation-signature then
    procedure.s-Procedure-identifier
else if 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.

\[
\text{allVirtualArgsSet}(\text{args}:\text{Formal-argument}^*, \text{values}:\text{VALUE}^*) : \text{BOOLEAN} = \text{def}
\]

if args = empty then
    True
else if 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.

\[
\text{allDynamicCandidates}(\text{procedure}:\text{PROCEDURE}) : \text{BOOLEAN} = \text{def}
\]

\{ p | p ∈ Dynamic-operation-signature:  \text{p.s-Operation-name} = \text{procedure.s-Operation-name} \}

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

\[
\text{matchingCandidates}(\text{procedures}:\text{PROCEDURE-set}, \text{values}:\text{VALUE}^*) : \text{PROCEDURE-set} = \text{def}
\]

\{ p | p ∈ procedures: isSignatureCompatible(p.s-Formal-argument, dynamicTypes(values)) \}

The function `bestMatch` returns the most specialized signature.

\[
\text{bestMatch}(\text{procedures}:\text{PROCEDURE-set}) : \text{PROCEDURE} = \text{def}
\]

take(\{ p | p ∈ procedures:  \forall q ∈ procedures: \text{isSignatureCompatible}(p.s-Formal-argument-seq, q.s-Formal-argument-seq) } )

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

\[
\text{isSignatureCompatible}(p:\text{Formal-argument}^*, q:\text{Formal-argument}^*) : \text{BOOLEAN} = \text{def}
\]

if p = empty then
True
else
   isSortCompatible(p.head.s-Argument, q.head.s-Argument) ∧
   isSignatureCompatible(p.tail, q.tail)
endif

dynamicTypes(values:VALUE*): Identifier* = 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 = 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 = 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 = 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.
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.
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
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 l1-2. Sub-labels, that appear during the compilation of expressions are represented in the form l1(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-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 |
| Assignment-attempt |
| Informal-text |
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
Active, 13, 34; defined at, 13
activeHandler, 13, 14, 28, 29, 67; defined at, 21
agentMode1, 28, 29, 49, 50, 51, 52; defined at, 20
agentMode2, 14, 28, 29, 35, 37, 51, 52, 58, 63, 64, 65, 66, 79, 81, 82; defined at, 20
agentMode3, 14, 28, 29, 37, 38, 53, 64, 65, 66, 70, 71, 72, 73, 74, 76, 78, 79, 81, 82; defined at, 20
agentMode4, 28, 29, 38, 53, 58, 64, 66, 67, 68, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82; defined at, 20
agentMode5, 28, 29, 77, 78; defined at, 20
allDynamicCandidates, 97; defined at, 97
allVirtualArgsSet, 97; defined at, 97
Applicable, 12, 32; defined at, 12
Arraytype, 83; defined at, 84
arrival, 9, 10, 13, 63; defined at, 9
assign, 23, 96; defined at, 23
assignClones, 23, 96; defined at, 23
assignCopies, 23, 96; defined at, 23
assignValues, 95; defined at, 95
bagand, 91; defined at, 92
bagcount, 91, 92; defined at, 91
bagdel, 91; defined at, 91
baginbag, 91; defined at, 92
bagincl, 91; defined at, 91
baglength, 92; defined at, 92
bagor, 91; defined at, 92
Bagtype, 83; defined at, 84
behaviour, 24, 49, 74, 77, 78, 79; defined at, 24
bestMatch, 97; defined at, 97
BooleanType, 83, 85, 91, 93, 98; defined at, 84
caller, 37, 38, 95, 96; defined at, 96
callingProcedureNode, 29, 51, 59; defined at, 16
CharacterType, 83, 87; defined at, 84
charChr, 87; defined at, 87
charValue, 87; defined at, 87
clock: defined at, 6
collectCurrentSubStates, 15, 38, 68, 70, 71, 72, 73, 74, 75, 76; defined at, 15
compareTail, 84, 92; defined at, 84
compile, 40, 41, 42, 43, 45, 49; defined at, 40
compileExpr, 40, 41, 42, 43, 44, 45, 46, 47; defined at, 40
compute, 27, 36, 83; defined at, 24
computeArray, 83; defined at, 90
computeBag, 83; defined at, 91
computeBoolean, 83, 85; defined at, 85
computeChar, 83; defined at, 87
computeDuration, 83; defined at, 88
computeFix, 88; defined at, 88
computeInteger, 83; defined at, 88
computeLiteral: defined at, 93
computePowerset, 83; defined at, 91
computeReal, 83, 88, 89; defined at, 88
computeSideEffects, 24, 84; defined at, 24
computeString, 83; defined at, 89
computeStruct: defined at, 93
computeTime, 83; defined at, 89
conditionItemCheck, 98; defined at, 98
continueLabel, 28, 29, 38, 39, 82; defined at, 20
continuousPriorities, 75, 76; defined at, 20
continuousSignalTransitions, 76; defined at, 22
currentConnector, 37, 51, 66, 69; defined at, 19
currentExceptionInst, 14, 51, 66, 68; defined at, 19
currentExitPoints, 68, 80, 82; defined at, 15
currentExitStateNodes, 66, 69, 82; defined at, 19
currentLabel, 20, 25, 27, 28, 29, 30, 31, 33, 34, 36, 37, 38, 39, 53, 64, 65, 66, 73, 74, 76, 77, 78, 79, 82, 83; defined at, 20
currentParentStateNode, 28, 29, 38, 39, 58, 59, 60, 61, 62, 65, 66, 67, 68, 69; defined at, 21
currentProcedureStateNode, 53, 59; defined at, 21
currentSignalInst, 18, 30, 51, 66, 71, 73, 74; defined at, 18
currentStartNodes, 51, 66, 69; defined at, 19
currentStateId, 28, 29, 30, 37, 38, 57, 65, 66, 73, 76, 78; defined at, 20
currentSubStates, 15, 68, 80, 81, 82; defined at, 15
currentTime: defined at, 6
definingSort, 84, 85, 87, 88, 89, 90, 91, 93; defined at, 84
delay, 11, 12; defined at, 11
delete, 10, 13; defined at, 10
direction, 21, 54, 55; defined at, 9
DirectlyInheritsFrom, 15, 16, 69, 70; defined at, 15
DirectlyRefinedBy, 15; defined at, 15
dispatch, 27, 97; defined at, 24
DivisionByZero, 85, 86, 88; defined at, 84
duration, 13; defined at, 13
DurationType, 83, 89; defined at, 84
dynamicType, 96, 98; defined at, 96
dynamicTypes, 97; defined at, 98
ehNodesToBeCreated, 57, 58, 59, 62; defined at, 19
ehParentStateNodeChecked, 67, 68; defined at, 20
empty, 10, 12, 25, 31, 32, 40, 42, 43, 44, 45, 46, 47, 48, 54, 63, 67, 70, 72, 86, 88, 91, 92, 95, 97; defined at, 6

ITU-T Z.100/Annex F3 (01/2000) 109
AgentMode, 20; defined at, 8
signalType, 8, 11, 12, 68, 71, 74; defined at, 8
signalValues: defined at, 8
specialName, 92, 93; defined at, 92
Spontaneous, 18, 70, 72, 73, 75; defined at, 18
spontaneousTransitions, 77; defined at, 22
startLabel, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 58, 60; defined at, 47
startNodeChecked, 69; defined at, 19
startTransitions, 58, 59, 60, 61, 62, 69; defined at, 14
state, 18, 22, 27, 28, 30, 37, 38, 56, 57; defined at, 18
stateAgent, 18, 27, 30, 37, 38, 51, 56, 57, 63, 64; defined at, 18
stateId, 56, 65, 66, 73, 76, 78; defined at, 14
stateLabel, 15, 38, 42, 58, 59, 60, 61, 67; defined at, 14
stateNodeChecked, 37, 65, 66, 69, 70, 71, 73, 74, 75, 76, 77, 78; defined at, 19
stateNodeKind, 16, 38, 58, 59, 60, 61, 67, 68, 71, 73, 74, 76, 82; defined at, 14
stateNodeRefinement, 61, 62, 68, 80, 81; defined at, 14
stateNodesToBeChecked, 70, 71, 72, 73, 74, 75, 76; defined at, 19
stateNodesToBeCreated, 57, 58, 59, 60, 62; defined at, 19
stateNodesToBeEntered, 38, 53, 64, 80, 81; defined at, 20
stateNodesToBeLeft, 38, 81, 82; defined at, 20
stateNodesToBeRefined, 57, 58, 59, 60, 61; defined at, 19
stateNodesToBeSpecialised, 57, 58, 59, 60, 61; defined at, 19
stateNodeToBeExited, 29, 82; defined at, 20
statePartitionSet, 60, 62, 80; defined at, 14
statePartitionsToBeCreated, 57, 58, 59, 60, 62; defined at, 19
stateTransitions, 58, 59, 60, 61, 68, 69, 71, 74, 76, 77; defined at, 14
StringType, 83; defined at, 84
structComputeObjects, 84; defined at, 93
structExtract, 92, 93; defined at, 92
structModify, 93; defined at, 93
structPartitionSet, 60, 64, 70, 71, 72, 73, 75; defined at, 19
to, 11, 12, 55; defined at, 11
toArg, 8, 11, 12, 32, 63; defined at, 8
topStateId, 56; defined at, 18
topStateMachine, 57, 60, 64, 70, 71, 72, 73, 75; defined at, 19
toSet, 41, 42, 43, 45; defined at, 6
transitionChecked, 73, 74, 76, 77, 78; defined at, 19
transitionsToBeChecked, 73, 74, 75, 76; defined at, 19
UndefinedField, 92; defined at, 84
UndefinedVariable: defined at, 84
uniqueLabel, 40, 41, 42, 43, 44, 45, 46, 47, 48, 103; defined at, 40
update, 95, 96; defined at, 95
value, 25, 27, 29, 30, 31, 32, 33, 34, 36, 40, 74, 77, 78, 95; defined at, 25
values, 25, 27, 31, 33, 34, 35; defined at, 25
variableSort, 96; defined at, 96
viaArg, 8, 11, 12, 32; defined at, 8
with, 11, 12, 55; defined at, 11

II.2 Domains

Action, 24, 25, 26; defined at, 25
Agent, 9, 11, 12, 14, 18, 20, 21, 49, 51, 53, 54, 55; defined at, 6
AgentMode, 20; defined at, 17
Answer, 31, 43, 46; defined at, 36
AssignParameters, 25, 26, 30, 41; defined at, 30
 Behaviour, 23, 24, 40, 42, 43; defined at, 24
Boolean, 12, 13, 15, 18, 20, 21, 23, 24, 29, 55, 64, 83, 84, 85, 92, 96, 97, 98; defined at, 6
BoundValue, 94, 95; defined at, 94
Break, 25, 26, 37, 43, 45; defined at, 37
Call, 25, 26, 28, 44, 47; defined at, 28
Contiue, 25, 26, 38, 43; defined at, 37
ContinueLabel, 19, 20, 25, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 53, 57, 58, 59; defined at, 25
Create, 25, 26, 32, 44; defined at, 32
Decision, 25, 26, 31, 43, 46; defined at, 31
Declaration, 22, 94, 95; defined at, 22
Direction, 9, 55; defined at, 9
Duration, 11, 13; defined at, 11
EnterStateNode, 25, 26, 38, 42, 65; defined at, 38
Equality, 25, 26, 30, 46; defined at, 30

Exception, 7, 8, 13, 14, 23; defined at, 13
ExceptionInst, 8, 14, 19; defined at, 13
ExceptionScope, 13, 14, 20, 21, 28, 29, 34, 43, 67; defined at, 13
Field, 92, 93; defined at, 92
FreeAction, 14, 40, 65; defined at, 21
Frequency, 91, 92; defined at, 91
Gate, 9, 10, 11, 12, 18, 21, 53, 54, 55; defined at, 9
Label, 16, 20, 21, 22, 24, 25, 36, 40, 43, 47, 65, 66, 73, 74, 76, 77, 78, 83; defined at, 40
LeafStateNode, 25, 26, 38, 42, 65; defined at, 38
Link, 11, 12, 14, 18, 55, 56; defined at, 11
NamedValue, 94, 95, 96; defined at, 94
Nat, 11, 15, 20, 21, 22, 30, 40, 76, 83, 85, 86, 87, 88, 89, 90, 91, 92, 106, 108; defined at, 6
Object, 23, 93, 94, 96; defined at, 94
ObjectIdentifier, 24, 94; defined at, 24
ObjectValue, 23, 24, 94, 96; defined at, 94
OperationApplication, 25, 26, 27, 45, 46; defined at, 27
Output, 25, 26, 31, 44; defined at, 31
PId, 8, 12, 13, 18, 23, 32, 51, 63, 92, 97; defined at, 92
PlainSignal, 7, 8; defined at, 7
PlainSignalInst, 7, 8, 32; defined at, 7
II.3 AS1 Nonterminals

Action-return-node, 42, 48, 109; defined at, 109
Active-expression, 103, 111; defined at, 111
Agent-definition, 18, 49, 50, 51, 100, 101, 102, 106; defined at, 106
Agent-formal-parameter, 106, 110; defined at, 106
Agent-identifier, 12, 32, 109; defined at, 105
Agent-kind, 33, 50, 51, 64, 100, 101, 102, 106; defined at, 106
Agent-name, 100, 101, 102, 106; defined at, 105
Agent-qualifier, 105; defined at, 106
Agent-type-definition, 32, 49, 51, 53, 54, 56, 100, 101, 102, 106; defined at, 106
Agent-type-identifier, 49, 50, 51, 53, 54, 56, 100, 101, 102, 106; defined at, 105
Agent-type-name, 100, 101, 102, 106; defined at, 105
Agent-type-qualifier, 105; defined at, 106
Alternative-expression, 112; defined at, 112
Any-expression, 46, 48, 112; defined at, 112
Argument, 111; defined at, 111
Assignment, 44, 47, 102, 103, 109; defined at, 44, 47, 109
Assignment-attempt, 44, 47, 109; defined at, 109
Boolean-expression, 108, 112; defined at, 108
Break-node, 43, 48, 109; defined at, 110
Call-node, 44, 47, 108; defined at, 109
Channel-definition, 21, 54, 55, 100, 101, 106; defined at, 107
Channel-identifier, 109; defined at, 105
Channel-name, 100, 101, 107; defined at, 105
Channel-path, 21, 54, 55, 100, 101, 107; defined at, 107
Closed-range, 45, 48, 98, 111; defined at, 111
Composite-state-formal-parameter, 110; defined at, 110
Composite-state-graph, 62, 80, 81, 110; defined at, 110
Composite-state-type-definition, 42, 56, 62, 106, 107, 110; defined at, 110
Composite-state-type-identifier, 61, 80, 81, 108, 110; defined at, 105
Compound-node, 44, 47, 108; defined at, 109
Conditional-expression, 46, 48, 111; defined at, 112
Condition-item, 98, 111; defined at, 111
Condition-definition, 60, 110; defined at, 110
Connect-node, 39, 42, 44, 47, 108; defined at, 110
Connector-name, 14, 19, 21, 36, 37, 38, 40, 57, 69, 108, 109, 110; defined at, 105
Consequence-expression, 112; defined at, 112
Constant-expression, 87, 94, 95, 102, 103, 107, 111; defined at, 111
Continue-node, 43, 48, 109; defined at, 110
Continuous-expression, 39, 108; defined at, 108
Continuous-signal, 39, 42, 47, 108; defined at, 108
Create-request-node, 44, 47, 108; defined at, 109
Dash-nextstate, 42, 48, 109; defined at, 109
Data-type-definition, 84, 96, 106, 107, 110; defined at, 111
Data-type-identifier, 102, 111; defined at, 105
Data-type-name, 83, 106; defined at, 105
Data-type-qualifier, 85, 106; defined at, 106
Decision-answer, 48, 110; defined at, 110
Decision-node, 40, 43, 44, 48, 108; defined at, 110
Decision-question, 110; defined at, 110
DefinitionAS0: defined at, 6
DefinitionAS1, 7, 18, 40, 43, 47, 83; defined at, 6
Destination-gate, 21, 55, 100, 101, 107; defined at, 107
Direct-via, 109; defined at, 109
Dynamic-operation-signature, 24, 111; defined at, 111
Else-answer, 110; defined at, 110
Else-handle-node, 41, 43, 47, 107; defined at, 107
Entry-connection-definition, 60, 110; defined at, 110
Entry-procedure-definition, 80, 110; defined at, 110
Equality-expression, 46, 48, 111; defined at, 112
Exception-definition, 13, 106, 110; defined at, 107
Exception-handler-name, 13, 20, 21, 34, 59, 107; defined at, 105
Exception-handler-node, 19, 39, 41, 62, 108, 109; defined at, 107
Exception-identifier, 35, 39, 107, 110; defined at, 105
Exception-name, 107; defined at, 105
Exit-connection-definition, 60, 110; defined at, 110
Exit-procedure-definition, 81, 110; defined at, 110
Expanded-sort-identifier, 105; defined at, 105
Expression, 102, 103, 108, 109, 110, 112; defined at, 111
First-operand, 112; defined at, 112
Formal-argument, 97, 111; defined at, 111
Free-action, 40, 42, 47, 108; defined at, 108
Gate-definition, 53, 100, 101, 102, 106, 110; defined at, 107
Gate-identifier, 100, 101, 107, 109; defined at, 105
Gate-name, 100, 101, 102, 107; defined at, 105
Graph-node, 43, 44, 47, 102, 103, 108, 109; defined at, 108
Handle-node, 39, 41, 43, 47, 83, 107; defined at, 107
Identifier, 7, 8, 12, 13, 22, 24, 44, 48, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 95, 96, 97, 98, 105; defined at, 105
Imperative-expression, 111; defined at, 111
Informal-text, 109, 110; defined at, 106
Init-graph-node, 109; defined at, 109
Initial-number, 51, 100, 101, 102, 106; defined at, 106
Inner-entry-point, 60, 110; defined at, 110
Inner-exit-point, 60, 110; defined at, 110
Inout-parameter, 107; defined at, 107
In-parameter, 44, 47, 107; defined at, 107
Input-node, 39, 41, 43, 47, 71, 74, 103, 108; defined at, 108
In-signal-identifier, 54, 55, 100, 101, 102, 107; defined at, 107
Interface-definition, 32, 51, 92, 111; defined at, 111
Interface-name, 106; defined at, 105
Interface-qualifier, 106; defined at, 106
Join-node, 43, 48, 109; defined at, 109
Literal, 46, 48, 85, 87, 88, 89, 111; defined at, 85, 87, 93, 94, 105, 111, 112
Maximum-number, 32, 100, 101, 106; defined at, 106
II.4 Macros

Assign, 30; defined at, 23
CreateAgent, 32, 33, 50, 51; defined at, 51
CreateAgentSet, 49; defined at, 49
CreateAgentVariables, 52; defined at, 56
CreateAllAgents, 50, 51; defined at, 51
CreateAllAgentSets, 52; defined at, 49
CreateAllChannels, 52; defined at, 54
CreateAllGates, 49, 50; defined at, 53
CreateAllLinks, 52; defined at, 55
CreateChannel, 54; defined at, 54
CreateChannelPath, 54; defined at, 55
CreateCompositeState, 61; defined at, 62
CreateCompositeStateGraph, 62; defined at, 62
CreateCompositeStateVariables, 61; defined at, 56
CreateCompoundNodeVariables, 37; defined at, 57
CreateExceptionHandlerNode, 57, 58; defined at, 59
CreateGate, 53; defined at, 53
CreateInheritedState, 57, 58; defined at, 61
CreateInputPort, 51; defined at, 54
CreateLink, 55; defined at, 55
CreateProcedure, 27, 28, 52, 80, 81; defined at, 53
CreateProcedureGraph, 53, 58; defined at, 58
CreateProcedureGraphNodes, 61; defined at, 62
CreateProcedureStateNode, 58; defined at, 59
CreateProcedureVariables, 59, 61; defined at, 56
CreateStateAggregationNode, 62; defined at, 62
CreateStateMachine, 52, 57; defined at, 52
CreateStatePartition, 57, 58; defined at, 57
CreateStateRefinement, 58, 61; defined at, 61
CreateStateTransitionGraph, 62; defined at, 62
CreateTopStatePartition, 57, 60; defined at, 60
Delete, 10, 12, 13, 63, 71, 73, 74; defined at, 10
DeleteSignals, 62; defined at, 63
EnterStateNodes, 79; defined at, 79
EnterStateNodesEnteringFinished, 79; defined at, 81
EnterStateNodesEnterPhase, 79; defined at, 80
EnterStateNodesStartTimePhase, 79; defined at, 79
Eval, 26, 74, 77, 78, 79; defined at, 26
EvalAnyValue, 26; defined at, 36
EvalAssignParameters, 26; defined at, 30
EvalBreak, 26; defined at, 37
EvalCall, 26, 28; defined at, 28
EvalContinue, 26; defined at, 38
EvalCreate, 26; defined at, 32
EvalDecision, 26, 31; defined at, 31
EvalEnterStateNode, 26; defined at, 38
EvalEquality, 26; defined at, 30
EvalExitCompositeState, 29; defined at, 29
EvalExitProcedure, 29; defined at, 29
EvalExitStateNode, 26; defined at, 38
EvalOperationApplication, 26; defined at, 27
EvalOutput, 26, 31; defined at, 31
EvalRaise, 26, 35; defined at, 35
EvalReset, 26, 33; defined at, 34
EvalReturn, 26; defined at, 29
EvalScope, 26; defined at, 37
EvalSet, 26, 33; defined at, 33
EvalSetHandler, 26, 34; defined at, 34
EvalSetRangeCheckValue, 26; defined at, 36
EvalSkip, 26; defined at, 37
EvalStop, 26, 35; defined at, 35
EvalSystemValue, 26; defined at, 36
EvalTask, 26; defined at, 30
EvalTimerActive, 26, 34; defined at, 34
EvalVar, 26, 27; defined at, 27
ExecAgent, 52, 63; defined at, 63
ExecAgentSet, 50; defined at, 62
ExecutionStartPhase, 63; defined at, 64
ExitCompositeState, 79; defined at, 82
ExitTransitionFound, 69; defined at, 65
FireAction, 78; defined at, 79
FireTransition, 63; defined at, 78
ForwardSignal, 55; defined at, 12
FreeActionFound, 69; defined at, 65
GetExecRight, 52; defined at, 63
HandleNodeFound, 68; defined at, 66
InitAgent, 52; defined at, 52
InitAgentSet, 50; defined at, 50
InitProcedure, 79; defined at, 53
InitProcedureGraph, 53, 58; defined at, 58
InitStateMachine, 52, 57; defined at, 57
Insert, 10, 12, 32, 63; defined at, 10
LeaveStateNodes, 79; defined at, 81
LeaveStateNodesLeavePhase, 81; defined at, 81
LeaveStateNodesLeavingFinished, 81; defined at, 82
RaiseException, 31, 32, 35; defined at, 14
RemoveAgent, 83; defined at, 52
RemoveAgentSet, 50; defined at, 50
RemoveAllAgentSets, 83; defined at, 50
RemoveAllLinks, 52; defined at, 55
RemoveLink, 55; defined at, 56
ResetExceptionHandler, 28, 66; defined at, 14
ResetTimer, 33, 34; defined at, 13
RestoreExceptionScopes, 29, 68; defined at, 29
RestoreProcedureControlBlock, 29; defined at, 29
ReturnExecRight, 65, 66, 70, 71, 73, 74, 76, 79, 81; defined at, 64
SaveExceptionScopes, 59; defined at, 28
SaveProcedureControlBlock, 59; defined at, 28
SelContinuousEvaluationPhase, 75; defined at, 77
SelContinuousSelectionPhase, 75; defined at, 75
SelContinuousStartPhase, 75; defined at, 75
SelectContinuous, 65; defined at, 67
SelectException, 65; defined at, 67
SelectExitTransition, 65; defined at, 69
SelectFreeAction, 65; defined at, 69
SelectInput, 65; defined at, 72
SelectPriorityInput, 65; defined at, 70
SelectSpontaneous, 70, 72, 75; defined at, 77
SelectStartTransition, 65; defined at, 69
SelectTransition, 63; defined at, 64
SelectTransitionStartPhase, 64; defined at, 66
SelExceptionSelectionPhase, 67; defined at, 67
SelExceptionStartPhase, 67; defined at, 67
SelInputEvaluationPhase, 72; defined at, 74
SelInputSelectionPhase, 72; defined at, 73
SelStartPhase, 72; defined at, 72
SelPriorityInputSelectionPhase: defined at, 70
SelPriorityInputStartPhase, 70; defined at, 70
SelSpontaneousEvaluationPhase, 77; defined at, 78
SelSpontaneousSelectionPhase, 77; defined at, 77
SetExceptionHandler, 34; defined at, 13
SetTimer, 33; defined at, 13

SignalOutput, 31; defined at, 32
StartTransitionFound, 69; defined at, 65
StopPhase, 63; defined at, 82
TransitionFound, 71, 73, 74, 76, 77, 78; defined at, 65
UndefinedBehaviour, 68; defined at, 39

II.5 Programs
Agent-Program, 51; defined at, 52
Agent-Set-Program, 49, 50; defined at, 50

Link-Program, 11, 55; defined at, 55
Undefined-Behaviour-Program, 39; defined at, 39
## SERIES OF ITU-T RECOMMENDATIONS

<table>
<thead>
<tr>
<th>Series</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Organization of the work of ITU-T</td>
</tr>
<tr>
<td>B</td>
<td>Means of expression: definitions, symbols, classification</td>
</tr>
<tr>
<td>C</td>
<td>General telecommunication statistics</td>
</tr>
<tr>
<td>D</td>
<td>General tariff principles</td>
</tr>
<tr>
<td>E</td>
<td>Overall network operation, telephone service, service operation and human factors</td>
</tr>
<tr>
<td>F</td>
<td>Non-telephone telecommunication services</td>
</tr>
<tr>
<td>G</td>
<td>Transmission systems and media, digital systems and networks</td>
</tr>
<tr>
<td>H</td>
<td>Audiovisual and multimedia systems</td>
</tr>
<tr>
<td>I</td>
<td>Integrated services digital network</td>
</tr>
<tr>
<td>J</td>
<td>Cable networks and transmission of television, sound programme and other multimedia signals</td>
</tr>
<tr>
<td>K</td>
<td>Protection against interference</td>
</tr>
<tr>
<td>L</td>
<td>Construction, installation and protection of cables and other elements of outside plant</td>
</tr>
<tr>
<td>M</td>
<td>TMN and network maintenance: international transmission systems, telephone circuits, telegraphy, facsimile and leased circuits</td>
</tr>
<tr>
<td>N</td>
<td>Maintenance: international sound programme and television transmission circuits</td>
</tr>
<tr>
<td>O</td>
<td>Specifications of measuring equipment</td>
</tr>
<tr>
<td>P</td>
<td>Telephone transmission quality, telephone installations, local line networks</td>
</tr>
<tr>
<td>Q</td>
<td>Switching and signalling</td>
</tr>
<tr>
<td>R</td>
<td>Telegraph transmission</td>
</tr>
<tr>
<td>S</td>
<td>Telegraph services terminal equipment</td>
</tr>
<tr>
<td>T</td>
<td>Terminals for telematic services</td>
</tr>
<tr>
<td>U</td>
<td>Telegraph switching</td>
</tr>
<tr>
<td>V</td>
<td>Data communication over the telephone network</td>
</tr>
<tr>
<td>X</td>
<td>Data networks and open system communications</td>
</tr>
<tr>
<td>Y</td>
<td>Global information infrastructure and Internet protocol aspects</td>
</tr>
<tr>
<td>Z</td>
<td>Languages and general software aspects for telecommunication systems</td>
</tr>
</tbody>
</table>