|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Fond-Rec_e | | **International Telecommunication Union** | | |
|  | |  | | |
| **ITU-T** | **Z.100** | |
| TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU | | **Annex F3**  (11/2018) |
|  | SERIES Z: LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS  Formal description techniques (FDT) – Specification and Description Language (SDL) | | | |
|  | Specification and Description Language – Overview of SDL‑2010  **Annex F3: SDL‑2010 formal definition: Dynamic semantics** | | | |
|  | Recommendation ITU‑T Z.100  – Annex F3 | | | |



ITU-T Z-SERIES RECOMMENDATIONS

**LANGUAGES AND GENERAL SOFTWARE ASPECTS FOR TELECOMMUNICATION SYSTEMS**

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

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

|  |
| --- |
| Recommendation ITU-T Z.100  Specification and Description Language – Overview of SDL‑2010  Annex F3  SDL‑2010 formal definition: Dynamic semantics |
| Summary  This annex defines the SDL‑2010 dynamic semantics. |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| History   |  |  |  |  |  | | --- | --- | --- | --- | --- | | Edition | Recommendation | Approval | Study Group | Unique ID[[1]](#footnote-2)\* | | 1.0 | ITU-T Z.100 | 1984-10-19 |  | [11.1002/1000/2222](http://handle.itu.int/11.1002/1000/2222) | | 1.1 | ITU-T Z.100 Annex A | 1984-10-19 |  | [11.1002/1000/6664](http://handle.itu.int/11.1002/1000/6664) | | 1.2 | ITU-T Z.100 Annex B | 1984-10-19 |  | [11.1002/1000/6665](http://handle.itu.int/11.1002/1000/6665) | | 1.3 | ITU-T Z.100 Annex C1 | 1984-10-19 |  | [11.1002/1000/6666](http://handle.itu.int/11.1002/1000/6666) | | 1.4 | ITU-T Z.100 Annex C2 | 1984-10-19 |  | [11.1002/1000/6667](http://handle.itu.int/11.1002/1000/6667) | | 1.5 | ITU-T Z.100 Annex D | 1984-10-19 |  | [11.1002/1000/6668](http://handle.itu.int/11.1002/1000/6668) | | 2.0 | ITU-T Z.100 | 1987-09-30 | X | [11.1002/1000/10954](http://handle.itu.int/11.1002/1000/10954) | | 2.1 | ITU-T Z.100 Annex A | 1988-11-25 |  | [11.1002/1000/6669](http://handle.itu.int/11.1002/1000/6669) | | 2.2 | ITU-T Z.100 Annex B | 1988-11-25 |  | [11.1002/1000/6670](http://handle.itu.int/11.1002/1000/6670) | | 2.3 | ITU-T Z.100 Annex C1 | 1988-11-25 |  | [11.1002/1000/6671](http://handle.itu.int/11.1002/1000/6671) | | 2.4 | ITU-T Z.100 Annex C2 | 1988-11-25 |  | [11.1002/1000/6672](http://handle.itu.int/11.1002/1000/6672) | | 2.5 | ITU-T Z.100 Annex D | 1988-11-25 | X | [11.1002/1000/3646](http://handle.itu.int/11.1002/1000/3646) | | 2.6 | ITU-T Z.100 Annex E | 1988-11-25 |  | [11.1002/1000/6673](http://handle.itu.int/11.1002/1000/6673) | | 2.7 | ITU-T Z.100 Annex F1 | 1988-11-25 | X | [11.1002/1000/3647](http://handle.itu.int/11.1002/1000/3647) | | 2.8 | ITU-T Z.100 Annex F2 | 1988-11-25 | X | [11.1002/1000/3648](http://handle.itu.int/11.1002/1000/3648) | | 2.9 | ITU-T Z.100 Annex F3 | 1988-11-25 | X | [11.1002/1000/3649](http://handle.itu.int/11.1002/1000/3649) | | 3.0 | ITU-T Z.100 | 1988-11-25 |  | [11.1002/1000/3153](http://handle.itu.int/11.1002/1000/3153) | | 3.1 | ITU-T Z.100 Annex C | 1993-03-12 | X | [11.1002/1000/3155](http://handle.itu.int/11.1002/1000/3155) | | 3.2 | ITU-T Z.100 Annex D | 1993-03-12 | X | [11.1002/1000/3156](http://handle.itu.int/11.1002/1000/3156) | | 3.3 | ITU-T Z.100 Annex F1 | 1993-03-12 | X | [11.1002/1000/3157](http://handle.itu.int/11.1002/1000/3157) | | 3.4 | ITU-T Z.100 Annex F2 | 1993-03-12 | X | [11.1002/1000/3158](http://handle.itu.int/11.1002/1000/3158) | | 3.5 | ITU-T Z.100 Annex F3 | 1993-03-12 | X | [11.1002/1000/3159](http://handle.itu.int/11.1002/1000/3159) | | 3.6 | ITU-T Z.100 App. I | 1993-03-12 | X | [11.1002/1000/3160](http://handle.itu.int/11.1002/1000/3160) | | 3.7 | ITU-T Z.100 App. II | 1993-03-12 | X | [11.1002/1000/3161](http://handle.itu.int/11.1002/1000/3161) | | 4.0 | ITU-T Z.100 | 1993-03-12 | X | [11.1002/1000/3154](http://handle.itu.int/11.1002/1000/3154) | | 4.1 | ITU-T Z.100 (1993) Add. 1 | 1996-10-18 | 10 | [11.1002/1000/3917](http://handle.itu.int/11.1002/1000/3917) | | 5.0 | ITU-T Z.100 | 1999-11-19 | 10 | [11.1002/1000/4764](http://handle.itu.int/11.1002/1000/4764) | | 5.1 | ITU-T Z.100 (1999) Cor. 1 | 2001-10-29 | 17 | [11.1002/1000/5567](http://handle.itu.int/11.1002/1000/5567) | | 6.0 | ITU-T Z.100 | 2002-08-06 | 17 | [11.1002/1000/6029](http://handle.itu.int/11.1002/1000/6029) | | 6.1 | ITU-T Z.100 (2002) Amd. 1 | 2003-10-29 | 17 | [11.1002/1000/7091](http://handle.itu.int/11.1002/1000/7091) | | 6.2 | ITU-T Z.100 (2002) Cor. 1 | 2004-08-29 | 17 | [11.1002/1000/356](http://handle.itu.int/11.1002/1000/356) | | 7.0 | ITU-T Z.100 | 2007-11-13 | 17 | [11.1002/1000/9262](http://handle.itu.int/11.1002/1000/9262) | | 8.0 | ITU-T Z.100 | 2011-12-22 | 17 | [11.1002/1000/11387](http://handle.itu.int/11.1002/1000/11387) | | 8.1 | ITU-T Z.100 Annex F1 | 2000-11-24 | 10 | [11.1002/1000/5239](http://handle.itu.int/11.1002/1000/5239) | | 8.2 | ITU-T Z.100 Annex F2 | 2000-11-24 | 10 | [11.1002/1000/5576](http://handle.itu.int/11.1002/1000/5576) | | 8.3 | ITU-T Z.100 Annex F3 | 2000-11-24 | 10 | [11.1002/1000/5577](http://handle.itu.int/11.1002/1000/5577) | | 8.4 | ITU-T Z.100 Annex F1 | 2015-01-13 | 17 | [11.1002/1000/12354](http://handle.itu.int/11.1002/1000/12354) | | 8.5 | ITU-T Z.100 Annex F2 | 2015-01-13 | 17 | [11.1002/1000/12355](http://handle.itu.int/11.1002/1000/12355) | | 8.6 | ITU-T Z.100 Annex F3 | 2015-01-13 | 17 | [11.1002/1000/12356](http://handle.itu.int/11.1002/1000/12356) | | 9.0 | ITU-T Z.100 | 2016-04-29 | 17 | [11.1002/1000/12846](http://handle.itu.int/11.1002/1000/12846) | | 9.1 | ITU-T Z.100 Annex F1 | 2016-10-29 | 17 | [11.1002/1000/13040](http://handle.itu.int/11.1002/1000/13040) | | 9.2 | ITU-T Z.100 Annex F2 | 2016-10-29 | 17 | [11.1002/1000/13041](http://handle.itu.int/11.1002/1000/13041) | | 9.3 | ITU-T Z.100 Annex F3 | 2016-10-29 | 17 | [11.1002/1000/13042](http://handle.itu.int/11.1002/1000/13042) | | 9.4 | ITU-T Z.100 Annex F1 | 2018-11-13 | 17 | [11.1002/1000/13732](http://handle.itu.int/11.1002/1000/13732) | | 9.5 | ITU-T Z.100 Annex F2 | 2018-11-13 | 17 | [11.1002/1000/13733](http://handle.itu.int/11.1002/1000/13733) | | 9.6 | ITU-T Z.100 Annex F3 | 2018-11-13 | 17 | [11.1002/1000/13734](http://handle.itu.int/11.1002/1000/13734) | |

|  |
| --- |
| Keywords  Specification and Description Language, SDL‑2010, formal definition, Dynamic semantics, Behaviour semantics, SDL‑2010 abstract machine, SAM, Compilation function, SAM programs, Data semantics. |

FOREWORD

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

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

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

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

NOTE

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

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

INTELLECTUAL PROPERTY RIGHTS

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

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

© ITU 2019

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

**Table of Contents**

Page

Annex F3 – SDL‑2010 formal definition: Dynamic semantics 1

F3.1 General information 1

F3.2 Behaviour semantics 2

F3.3 Data semantics 76

Appendix I to Annex F3 – List of abstract syntax grammar rules used 95

Recommendation ITU-T Z.100

Specification and Description Language – Overview of SDL‑2010

Annex F3  
  
SDL‑2010 formal definition: Dynamic semantics

## F3.1 General information

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

### F3.1.1 Definitions from Annex F1

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

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

The domains Agent, Boolean, DefinitionAS1, Nat, Real,Time, Token, X.

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

The operation symbols \*, +, -set, -seq, =, ≠, ∧, ∨, ⇒, ⇔, ¬, ∃, ∀, >, ≥, <, ≤, +, -, \*, /, in, ×, ⁀, ∪, ∩, \, ∈, ∉, ⊆, ⊂, | |, U, ∅, mk**-**, s-, s1-, s2-, s3-.

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

### F3.1.2 Definitions from Annex F2

EntityDefinition1: the union of all the entity definitions in AS1. It is therefore a subset of DefinitionAS1.

EntityDefinition1=def Agent-definition

∪ Agent-type-definition

∪ Channel-definition

∪ Composite-state-type-definition

∪ Data-type-definition

∪ Gate-definition

∪ Literal-signature

∪ Operation-signature

∪ Package-definition

∪ Procedure-definition

∪ Signal-definition

∪ State-node

∪ Syntype-definition

∪ Timer-definition

∪ Variable-definition

EntityKind1: the set of all the entity kinds in AS1.

EntityKind1 =def {agent, agent type, package, state, state type, procedure, variable,

signal, timer, channel, gate, sort, exception, literal, operation}

Given an Identifier, the corresponding EntityDefinition1 is retrieved using the function idToNodeAS1:

idToNodeAS1(id: Identifier): [EntityDefinition1]=def

getEntityDefinition1(id, idKind1(id) )

where:

function getEntityDefinition1 from Annex F2 gets the entity definition for an identifier:

getEntityDefinition1: Identifier, EntityKind1→EntityDefinition1

and function idKind1 from Annex F2 is used determine the kind of the entity from the identifier:

idKind1:Identifier→EntityKind1

Given an EntityDefinition1, the corresponding Identifier is retrieved using the function identifier1 from Annex F2:

identifier1: EntityDefinition1→ Identifier

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

isSuperType: EntityDefinition1 × EntityDefinition1 → Boolean

### F3.1.3 Status of Annex F3 (this annex)

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

The work on Annex F between the (10/2016) and this edition has focused on the static semantics in Annex F2, therefore there has not been much change in this edition of Annex F3 and further study is needed (denoted by "Further study needed …" items the text below). When interpreting the abstract grammar, the difference between SDL‑2000 and SDL‑2010 is not very significant, because many of added SDL‑2010 features are either in *Model* clauses that disappear in transformations in Annex F2, or map from the SDL-2010 concrete grammar (from AS0 in Annex F2) to abstract grammar that is unchanged from SDL‑2000. Therefore the work to update Annex F3 for SDL‑2010 should be relatively simple compared to updating Annex F2. In addition to **REF** *Aggregation‑kind* some other features are state timers, gates on input nodes, handling of availability time, encode/decode with expressions/output. In the handling of inputs some of the description of the dynamic semantics is still based on SDL‑2000 and not updated to SDL‑2010. Checks need to be made that

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

## F3.2 Behaviour semantics

This clause defines the following parts of the dynamic semantics:

• the SAM (SDL‑2010 Abstract Machine): clause F3.2.1;

• the compilation function: clause F3.2.2; and

• SAM programs: clause F3.2.3.

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

### F3.2.1 SDL‑2010 abstract machine definition (SAM)

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

• signal flow related definitions: clause F3.2.1.1;

• SDL‑2010 agent-related definitions: clause F3.2.1.2;

• the interface to the data semantics: clause F3.2.1.3; and

• behaviour primitives: clause F3.2.1.4.

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

#### F3.2.1.1 Signal flow model

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

##### F3.2.1.1.1 Signals

PlainSignal represents the set of *signal types* as declared by an SDL‑2010 specification.

PlainSignal =def Identifier ∪ NONE

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

Signal =def PlainSignal ∪ Timer

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

shared domain PlainSignalInst

initially PlainSignalInst = ∅

shared plainSignalType: PlainSignalInst → PlainSignal

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

SignalInst =def PlainSignalInst ∪ TimerInst

signalType(si:SignalInst): Signal =def

if si ∈ PlainSignalInst then si.plainSignalType

elseif si ∈ TimerInst then si.s-Timer

endif

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

shared plainSignalSender: PlainSignalInst → Pid

signalSender(si:SignalInst): Pid=def

if si ∈ PlainSignalInst then si.plainSignalSender

elseif si ∈ TimerInst then si.s-Pid

endif

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

shared plainSignalValues: PlainSignalInst → Value\*

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

ViaArg =def Identifier-set

ToArg =def Pid ∪ Identifier

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

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

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

shared toArg: PlainSignalInst → [ToArg]

shared viaArg: PlainSignalInst→ ViaArg

##### F3.2.1.1.2 Gates

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

controlled domain Gate

initially Gate = ∅

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

Direction =def { inDir, outDir }

controlled direction: Gate → Direction

controlled myAgent: Gate → Agent

Global system time

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

monitored now: → Time

There are two integrity constraints on the behaviour of now:

1. now values change monotonically, increasing over ASM runs;

2. now values do not increase as long as a signal is in transit on a non-delaying channel.

Discrete delay model

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

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

shared arrival: SignalInst → Time

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

shared schedule: Gate → SignalInst\*

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

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

Waiting signals

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

queue(g: Gate): SignalInst\* =def < si in g.schedule: (now ≥ si.arrival) >

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

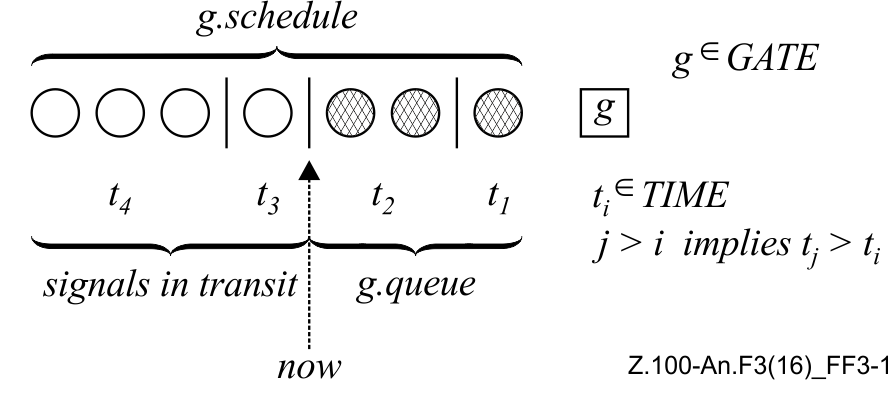


Figure F3-1 – Signal instances at a gate

Operations on schedules

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

insert(si:SignalInst, t:Time, siSeq:SignalInst\*): SignalInst\* =def

if siSeq = empty then

< si > ⁀ siSeq

elseif t < siSeq.head.arrival

then < si > ⁀ siSeq

else < siSeq.head > ⁀ insert(si, t, siSeq.tail)

endif

The function insert defines the result of inserting some signal instance si with the intended arrival time t into a finite signal instance list siSeq, representing (for example) the schedule of a gate. Analogously, a function delete is used to remove a signal from a finite signal instance list siSeq.

delete(si:SignalInst, siSeq:SignalInst\*): SignalInst\* =def

if siSeq = empty then empty

elseif siSeq.head = si then siSeq.tail

else < siSeq.head > ⁀ delete(si, siSeq.tail)

endif

The macros Insert and Delete update the schedule of a gate g by assigning some new signal list to g.schedule.

Insert(si:SignalInst, t:Time, g:Gate) ≡

g.schedule := insert(si,t,g.schedule)

si.arrival := t+si.delay

Delete(si:SignalInst, g:Gate) ≡

g.schedule := delete(si,g.schedule)

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

nextSignal(si: SignalInst, siSeq:SignalInst\*): [SignalInst] =def

if siSeq = empty then undefined

elseif siSeq.head = si then

if siSeq.tail = empty then undefined

else siSeq.tail.head

endif

else nextSignal(si, siSeq.tail)

endif

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

selectContinuousSignal(tSet: SemTransition-set, nSet: Nat-set): [SemTransition] =def

if ∀t1 ∈ tSet: t1.s-Nat ∈ nSet then undefined

else take({t ∈ tSet: t.s-Nat ∉ nSet ∧ ∀t1 ∈ tSet: (t1.s-Nat ∉ nSet ⇒ t.s-Nat ≤ t1.s-Nat)})

endif

##### F3.2.1.1.3 Channels

Channels, as declared in a given SDL‑2010 specification, consist of either one or two unidirectional *channel paths.* In the SAM model, each channel path is identified with an object of a derived domain Link. The elements of Link are SAM agents, such that their behaviour is defined through Link-Program.

Link =def Agent

LinkSeq =def Link\*

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

controlled with: Link → Signal-set

controlled from: Link → [ Gate ] // need to have optional result here, because function is also called within allConnections with general Agent

controlled to: Link → Gate

controlled noDelay: Link → [NODELAY]

Signal delays

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

Signal delays are modelled through a monitored function delay stating the dependency on external conditions and events. In a given SAM state, delay associates finite time intervals from a domain Duration to the elements of Link, where the duration of a particular signal delay appears to be chosen non-deterministically.

Duration =def Real

monitored delay: Link → Duration

****Integrity constraints****

There are two important integrity constraints on the function delay:

1. Taking into account that there are also non-delaying channels, the only admissible value for non-delaying channel paths is 0.

2. For every link agent *l*, the value of (now + l.delay) increases monotonically (with respect to now).

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

Channel behaviour

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

But, how does a link agent l know whether it is applicable to a signal si? Now, this decision does of course depend on the values of si.toArg, si.viaArg, si.signalType and l.with. In other words, l is a legal choice for the transportation of si only, if the following two conditions hold: (1) si.signalType ∈ l.with and (2) there exists an applicable path connecting l.to to some final destination that matches with the address information and the path constraints of si. Abstractly, this decision can be expressed using a predicate applicable, defined in clause F3.2.1.1.4. The domain ToArg is defined in clause F3.2.1.1.1.

##### F3.2.1.1.4 Reachability

When signals are sent, it has to be determined whether there currently is an applicable communication path: a path consisting of a sequence of links that can transfer the signal, and that satisfies further constraints as specified by the optional to- and via-arguments. The predicate applicable formally states all conditions that must be satisfied.

applicable(s: Signal, toArg: [ ToArg ], viaArg: ViaArg, g: Gate, l: [Link]): Boolean =def

∃ commPath ∈ allConnections (g):

(∀ ll ∈ commPath: s ∈ ll.with ∧ ll.owner ≠ undefined) ∧

if commPath = empty then

l = undefined ∧ ((g.direction = outDir) ⇒

(toArg = undefined ∧ s ∈ g.gateAS1.s-Out-signal-identifier-set)) ∧

((g.direction = inDir) ⇒ (validDestinationGate(g, toArg) ∧ // to self

s ∈ g.gateAS1.s-In-signal-identifier-set)) ∧ viaArg = ∅

else

if l ≠ undefined then commPath.head = l else true endif ∧

¬ ∃ll ∈ Link: (ll.from = commPath.last.to ∧ s ∈ ll.with) ∧ // the path is complete

viaArg ⊆ commPath.commPathIds ∧ validDestinationGate(commPath.last.to, toArg)

endif

// Further study needed because this function is difficult to understand.

// Some additional annotation and/or description of what it does might help.

// toArg: [ ToArg ] is Pid ∪ Identifier

// and corresponds to Signal-destination in an Output-node.

// It needs to checked that THIS and *Agent-identifier* *Destination-number*

// are converted to *Pid* before *applicable* is called.

validDestinationGate(g: Gate, toArg: [ ToArg ]): Boolean =def

if toArg ∈ Agent-identifier then

g.myAgent.agentAS1.identifier1 = toArg else true endif ∧

if toArg ∈ Pid∧ toArg ≠ nullPid then

∃sa ∈ Agent: (sa.owner = g.myAgent ∧ sa.selfPid = toArg) else true

endif

allConnections(g: Gate): LinkSeq-set =def

**U** ({ { < l > ⁀ list | list ∈ allConnections(l.to) } | l ∈ Link: l.from = g }) ∪

{ empty }

commPathIds(lSeq: Link\*): Identifier-set =def

{ g.gateAS1.identifier1 | g ∈ Gate: ∃l ∈ lSeq: (g = l.from ∨ g = l.to) } ∪

{ l.agentAS1.identifier1 | l ∈Link: (l ∈ lSeq) }

##### F3.2.1.1.5 Timers

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

Timer =def Identifier

TimerInst =def Pid × Timer × Value\*

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

Active timers

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

active(tmi:TimerInst): Boolean =def tmi ∈ Self.inport.schedule

Timer operations

The macros below model the SDL‑2010 actions Set-nodeand Reset-node on timers as executed by a corresponding SDL‑2010 agent. A static function (duration) is used to represent default duration values as defined by an SDL‑2010 specification under consideration.

static duration: Timer → Duration

SetTimer(tm:Timer, vSeq :Value\*, t:[Time]) ≡

let tmi = mk-TimerInst(Self.selfPid, tm, vSeq ) in

if t = undefined then

Self.inport.schedule := insert(tmi, now + tm.duration, delete(tmi, Self.inport.schedule))

tmi.arrival := now + tm.duration

else

Self.inport.schedule := insert(tmi, t, delete(tmi, Self.inport.schedule))

tmi.arrival := t

endif

endlet

ResetTimer(tm:Timer, vSeq :Value\*) ≡

let tmi = mk**-**TimerInst(Self.selfPid, tm, vSeq ) in

if active(tmi) then

Delete(tmi, Self.inport)

endif

endlet

##### F3.2.1.1.6 Exceptions

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

Exception =def Exception-identifier

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

#### F3.2.1.2 SDL‑2010 agents

In this clause, the domain Agent is further refined to consist of three basically different types of agents, namely: link agent instances (modelled by the domain Link, see clause F3.2.1.1.3), SDL‑2010 agent instances, and SDL‑2010 agent set instances (modelled by the derived domains SdlAgent and SdlAgentSet, respectively).

SdlAgent =def Agent

SdlAgentSet =def Agent-set

Initially, there is only a single agent system denoting a distinguished SDL‑2010 agent set instance of the domain SdlAgentSet.

static system: → SdlAgentSet

initially Agent = { system }

controlled agentSetPids: SdlAgentSet → Pid\*

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

##### F3.2.1.2.1 State machine

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

controlled domain StateNode

initially StateNode = ∅

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

StateNodeKind =def { stateNode, statePartition, procedureNode}

StateNodeRefinementKind =def { compositeStateGraph, stateAggregationNode}

StateEntryPoint =def [ State-entry-point-name ]

StateExitPoint =def State-exit-point-name ∪ DEFAULT

StateNodeWithEntryPoint =def StateNode × (StateEntryPoint ∪ HISTORY)

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.

controlled stateNodeKind: StateNode → StateNodeKind

controlled stateNodeRefinement: StateNode → [StateNodeRefinementKind]

controlled stateName: StateNode → State-name

controlled stateId: StateNode → StateId

controlled inheritedStateNode: StateNode → [StateNode]

controlled parentStateNode: StateNode → [StateNode]

controlled stateTransitions: StateNode → SemTransition-set

controlled startTransitions: StateNode → StartTransition-set

controlled freeActions: StateNode → FreeAction-set

controlled statePartitionSet: StateNode → StateNode-set

The stateNodeRefinement of a StateNode for a basic state is undefined.

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

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

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

controlled currentSubStates: StateNode → StateNode-set

controlled previousSubStates: StateNode → StateNode-set

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

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

collectCurrentSubStates(sn: StateNode): StateNode-set =def

{sn} ∪ **U** ({collectCurrentSubStates(x) | x ∈ sn.currentSubStates ∪ sn.inheritedStateNodes})

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

controlled currentExitPoints: StateNode → StateExitPoint-set

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

inheritsFrom(sn1: StateNode, sn2: StateNode): Boolean =def

if sn2.parentStateNode = undefined then false

elseif sn1.parentStateNode = undefined then false

else

sn2.parentStateNode ∈ sn1.parentStateNode.inheritedStateNodes ∧

sn1.stateName ≠ sn2.stateName

endif

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

directlyInheritsFrom(sn1: StateNode, sn2: StateNode): Boolean =def

inheritsFrom(sn1, sn2) ∧

(¬ ∃snx ∈ StateNode:

inheritsFrom(sn1, snx)∧ inheritsFrom(snx, sn2))

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

directlyRefinedBy(sn1: StateNode, sn2: StateNode): Boolean =def

sn2.parentStateNode = sn1

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

directlyInheritsFromOrRefinedBy(sn1: StateNode, sn2: StateNode): Boolean =def

directlyRefinedBy(sn1, sn2) ∨ directlyInheritsFrom(sn1, sn2)

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

inheritsFromOrRefinedBy(sn1: StateNode, sn2: StateNode): Boolean =def

directlyInheritsFromOrRefinedBy(sn1, sn2) ∨

(∃ sn3 ∈ { sn ∈ StateNode: directlyInheritsFromOrRefinedBy (sn1, sn) }:

(inheritsFromOrRefinedBy(sn3, sn2)))

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

selectNextStateNode(snSet: StateNode-set): [StateNode] =def

let sn = take({sn1 ∈ snSet: (¬ ∃sn2 ∈ snSet: inheritsFromOrRefinedBy(sn1, sn2))}) in

if sn = undefined then undefined

elseif ∃sn1 ∈ snSet: directlyInheritsFrom(sn1, sn) ∨ sn = sn1.inheritedStateNode then

selectNextStateNode(snSet \ {sn})

else sn

endif

endlet

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

inheritedStateNodes(sn: StateNode): StateNode-set =def

if sn.inheritedStateNode = undefined then ∅

else {sn.inheritedStateNode} ∪ sn.inheritedStateNode.inheritedStateNodes

endif

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

parentStateNodes(sn: StateNode): StateNode-set =def

if sn.parentStateNode = undefined then ∅

else {sn.parentStateNode} ∪ sn.parentStateNode.parentStateNodes

endif

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

mostSpecialisedStateNode(sn:StateNode): StateNode =def

let sn1 = take({sn2 ∈ StateNode: inheritsFrom(sn2, sn)}) in

if sn1 = undefined then sn else sn1.mostSpecialisedStateNode endif

endlet

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

selectInheritedStateNode(sn: StateNode, snSet: StateNode-set): [StateNode ]=def

take({sn1 ∈ snSet: directlyInheritsFrom(sn,sn1)})

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

getPreviousStatePartition(sn: StateNode): StateNode =def

if sn.stateNodeKind = statePartition ∧

¬ ∃sn1 ∈ sn.parentStateNodes: sn1.stateNodeKind = procedureNode

then sn.mostSpecialisedStateNode

else getPreviousStatePartition(sn.parentStateNode)

endif

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

controlled resultLabel: StateNode → Label

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

controlled callingProcedureNode: (Agent ∪ StateNode) → [StateNode]

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

controlled entryConnection: StateEntryPoint × StateNode → [StateEntryPoint]

controlled exitConnection: StateExitPoint × StateNode → StateExitPoint

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

##### F3.2.1.2.2 Agent modes

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

AgentMode =def {

initialisation, // agent mode 1

execution, // agent mode 1

selectingTransition, // agent mode 2

firingTransition, // agent mode 2

stopping, // agent mode 2

initialising1, // agent mode 2, 4

initialising2, // agent mode 2

initialisingStateMachine, // agent mode 2

initialisingProcedureGraph, // agent mode 4

initialisationFinished, // agent mode 2, 4

startSelection, // agent mode 3

selectFreeAction, // agent mode 3

selectExitTransition, // agent mode 3

selectStartTransition, // agent mode 3

selectPriorityInput, // agent mode 3

selectInput, // agent mode 3

selectContinuous, // agent mode 3

startPhase, // agent mode 2, 4

selectionPhase, // agent mode 4, 5

evaluationPhase, // agent mode 4, 5

selectSpontaneous, // agent mode 4

leavingStateNode, // agent mode 3

firingAction, // agent mode 3, 4

enteringStateNode, // agent mode 3

exitingCompositeState, // agent mode 3

initialisingProcedure, // agent mode 3

enterPhase, // agent mode 4

enteringFinished, // agent mode 4

leavePhase, // agent mode 4

leavingFinished} // agent mode 4

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

##### F3.2.1.2.3 Agent control block

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

controlled owner: Agent ∪ StateNode ∪ Link → [Agent]

Hierarchical system structure is modelled by means of a function owner defined on agents, and on state nodes (see clause F3.2.1.2.1), expressing *structural relations* between them and their constituent components. More specifically, an agent set instance is considered as owner of all those agent instances currently contained in the set; an agent instance owns its substructure, consisting of agent set instances. Similarly, a composite state node owns the state nodes or state partitions forming the refinement.

controlled agentAS1: Agent → Agent-definition

controlled channelAS1: Agent → [Channel-definition]

controlled gateAS1: Gate → [Gate-definition]

controlled stateAS1: StateNode → State-node

controlled procedureAS1: StateNode → Procedure-definition

controlled stateDefinitionAS1: StateNode → Composite-state-type-definition

controlled partitionAS1: StateNode → [State-partition]

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

isAgentSet(ag: Agent): Boolean =def ag.program = Agent-Set-Program

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

controlled selfPid: SdlAgent → Pid

controlled sender: SdlAgent → Pid

controlled parent: SdlAgent → [Pid]

controlled offspring: SdlAgent → Pid

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

controlled state: SdlAgent → State

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

controlled stateAgent: SdlAgent → SdlAgent

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

controlled topStateId: SdlAgent → StateId

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

controlled isActive: SdlAgent → [SdlAgent]

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

monitored spontaneous: Agent → Boolean

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

controlled inport: SdlAgent → Gate

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

controlled currentSignalInst: SdlAgent → [SignalInst]

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

controlled topStateNode: SdlAgent → StateNode

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

controlled currentStartNodes: SdlAgent → StateNodeWithEntryPoint-set

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

controlled currentExitStateNodes: SdlAgent → StateNodeWithExitPoint-set

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

controlled currentConnector: SdlAgent → [StateNodeWithConnector]

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

controlled scopeName: SdlAgent × StateId → Connector-name

controlled scopeContinueLabel: SdlAgent × StateId → ContinueLabel

controlled scopeStepLabel: SdlAgent × StateId → StepLabel

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

InitStateMachine/InitProcedureGraph control block

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

controlled stateNodesToBeCreated: SdlAgent → State-node-set

controlled statePartitionsToBeCreated: SdlAgent → State-partition-set

controlled stateNodesToBeRefined: SdlAgent → StateNode-set

controlled stateNodesToBeSpecialised: SdlAgent → StateNode-set

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

Selection control block

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

controlled inputPortChecked: SdlAgent → SignalInst\*

controlled stateNodesToBeChecked: SdlAgent → StateNode-set

controlled stateNodeChecked: SdlAgent → [StateNode]

controlled startNodeChecked: SdlAgent → StateNodeWithEntryPoint

controlled exitNodeChecked: SdlAgent → StateNodeWithExitPoint

controlled transitionsToBeChecked: SdlAgent → SemTransition-set

controlled transitionChecked: SdlAgent → SemTransition

controlled signalChecked: SdlAgent → SignalInst

controlled SignalSaved: SdlAgent → Boolean

controlled continuousPriorities: SdlAgent → Nat-set

Enter/Leave/ExitStateNode control block

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

controlled stateNodesToBeEntered: SdlAgent → StateNodeWithEntryPoint-set

controlled stateNodesToBeLeft: SdlAgent → StateNode-set

controlled stateNodeToBeExited: SdlAgent → [StateNodeWithExitPoint]

Procedure control block

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

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

controlled agentMode1: Agent ∪ StateNode → AgentMode

controlled agentMode2: Agent ∪ StateNode → AgentMode

controlled agentMode3: Agent ∪ StateNode → AgentMode

controlled agentMode4: Agent ∪ StateNode → AgentMode

controlled agentMode5: Agent ∪ StateNode → AgentMode

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

controlled currentStateId: SdlAgent ∪ StateNode → StateId

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

controlled currentLabel: SdlAgent ∪ StateNode → [Label]

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

controlled continueLabel: SdlAgent ∪ StateNode → [ContinueLabel]

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

controlled currentParentStateNode: SdlAgent ∪ StateNode → StateNode

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

controlled previousStateNode: SdlAgent ∪ StateNode → StateNode

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

controlled currentProcedureStateNode: SdlAgent ∪ StateNode → StateNode

The currentProcedureStateNode function refers to the current procedure state node.

##### F3.2.1.2.4 Agent connections

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

gateUnconnected(g:Gate):Boolean =def

let myDef: Agent-type-definition = g.myAgent.agentAS1.s-Agent-type-identifier.idToNodeAS1 in

∀cd ∈ myDef.s-Channel-definition-set: ∀cp ∈ cd.s-Channel-path-set:

(g.gateAS1 ≠ cp.s-Originating-gate.idToNodeAS1 ∧

g.gateAS1 ≠ cp.s-Destination-gate.idToNodeAS1)

endlet

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

ingates(a: 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

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

##### F3.2.1.2.5 Agent behaviour

For the transitions of agents, a tuple domain is introduced, consisting of the signal type, the start label for any firing conditions, a priority value, and the start label of the transition actions. Further study needed to determine if an optional identifier of a *Gate* for which *Gate*.*direction* = *inDir* ∧ *Gate*.*agent* is the local agent: that is, for an input with a gate specified for the signal. Additionally, state exit points may be given. Depending on the kind of transition, some of these components may be unspecified. For instance, in case of a non-priority input transition without an enabling condition, there is no priority and no firing transition.

SemTransition=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:

priorityInputTransitions(tSet:SemTransition-set): SemTransition-set =def

{ t ∈ tSet: t.s-Signal ≠ NONE ∧ t.s-Label = undefined ∧ t.s-Nat ≠ undefined }

inputTransitions(tSet:SemTransition-set): SemTransition-set =def

{ t ∈ tSet: t.s-Signal ≠ NONE ∧ t.s-Nat = undefined }

continuousSignalTransitions(tSet:SemTransition-set): SemTransition-set =def

{ t ∈ tSet: t.s-Signal = NONE ∧ t.s-Label ≠ undefined ∧ t.s-Nat ≠ undefined }

spontaneousTransitions(tSet:SemTransition-set): SemTransition-set =def

{ t ∈ tSet: t.s-Signal = NONE ∧ t.s-Nat = undefined ∧ t.s-StateExitPoint = undefined }

exitTransitions(tSet:SemTransition-set): SemTransition-set =def

{ t ∈ tSet: t.s-StateExitPoint ≠ undefined }

#### F3.2.1.3 Interface to the data type part

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

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

##### F3.2.1.3.1 Functions provided by the data type part

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

derived domain State

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

The parameters of initAgentState are:

• State of the outermost process agent (undefined if the outermost process agent is being created)

• State ID of the new state

• State ID of the super state of the new state (undefined for the outermost agent)

• Declarations of the agent

The additional parameter for initProcedureState is

• List of parameter values and variable names

controlled domain StateId

Declaration=def Procedure-formal-parameter ∪ Variable-definition

initAgentState: [State] × StateId × [StateId] × Declaration-set→ State

initProcedureState: State × StateId× StateId × Declaration-set × Declaration\* × Value\* × Variable-identifier\* → State

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

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

Value =def SDLInteger ∪ SDLBoolean ∪ SDLReal ∪ SDLCharacter ∪ SDLString

∪ Pid∪ SDLLiterals ∪ SDLStructure ∪ SDLArray ∪ SDLPowerset

∪ SDLBag ∪ SDLTime ∪ SDLDuration

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

StateOrException =def State ∪ Exception

ValueOrException =def Value ∪ Exception

The data type part has to provide functions that model how assignments are performed, namely

assign: Variable-identifier × Value × State × StateId → StateOrException

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

Assign(variableName: Variable-identifier, value: Value, state: State, id: StateId) ≡

Self.stateAgent.state:= assign(variableName, value, state, id)

Assignments are the only way to change the state.

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

eval: Variable-identifier × State× StateId →Value

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

In order to handle expressions, the concurrent semantics provides a domain for procedure bodies, which is also used for method and operator bodies. The data part, in return, provides a static domain Procedure for procedures (definitions) and a function dispatch for procedure instances.

Procedure =def Static-operation-signature ∪ Literal-signature

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

dispatch: Procedure × Value\* → Identifier

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

functional: Procedure × Value\* → Boolean

compute: Procedure × Value\* → ValueOrException

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

derived domain SDLBoolean

derived domain SDLInteger

derived semvalueBool: SDLBoolean → Boolean

derived semvalueInt: SDLInteger→ Nat

derived semvalueRealNum: SDLReal→ Nat

derived semvalueRealDen: SDLReal→ Nat

derived semvalueReal: SDLReal → Real

##### F3.2.1.3.2 Functions used by the data type part

The following special points are worth noting:

• If two processes have part of their state in common (which could be possible due to the reference nature of the new data type part), there are no semantic problems in the concurrency part, as all state changes are automatically synchronized by the underlying ASM semantics.

• The values for the predefined variables of a process such as SENDER, PARENT, OFFSPRING, SELF, as well as the value of NOW are provided by the concurrency part.

#### F3.2.1.4 Behaviour primitives

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

behaviour: Behaviour =def rootNodeAS1.compile

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

StartLabel =def Label

Behaviour =def Primitive-set

Primitive =def Label × Action

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

##### F3.2.1.4.1 Action evaluation

Explanation

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

Representation

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

Action =def Var ∪ OperationApplication ∪ Call ∪ Return ∪ Task ∪ AssignParameters ∪ Equality ∪ Decision ∪ Output ∪ Create ∪ Set ∪ Reset ∪ TimerActive ∪ TimerRemaining ∪ Stop ∪ SystemValue ∪ AnyValue ∪ SetRangeCheckValue ∪ Scope ∪ Skip ∪ Break ∪ Continue ∪ EnterStateNode ∪ LeaveStateNode

Domains

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

ContinueLabel =def Label

ValueLabel =def Label

Functions

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

controlled value: ValueLabel × SdlAgent → Value

values(lSeq: ValueLabel\*, sa: SdlAgent): Value\* =def

if lSeq = empty then empty

else < value(lSeq.head,sa) > ⁀ values(lSeq.tail,sa)

endif

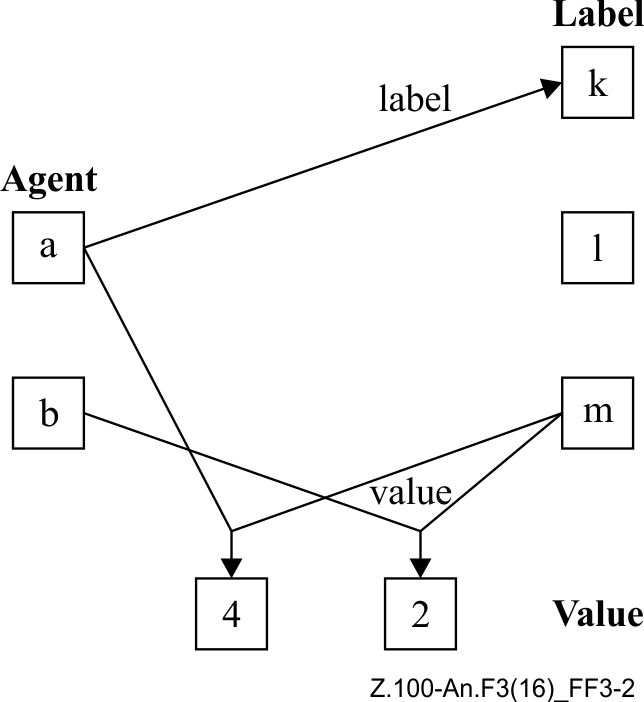


Figure F3-2 – Agents, labels and values

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

Behaviour

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

Eval(a:Action) ≡

if a ∈ Var then EvalVar(a)

elseif a ∈ OperationApplication then EvalOperationApplication(a)

elseif a ∈ Call then EvalCall(a)

elseif a ∈ Return then EvalReturn(a)

elseif a ∈ Task then EvalTask(a)

elseif a ∈ AssignParameters then EvalAssignParameters(a)

elseif a ∈ Equality then EvalEquality(a)

elseif a ∈ Decision then EvalDecision(a)

elseif a ∈ Output then EvalOutput(a) // this is the only place EvalOutput is used

elseif a ∈ Create then EvalCreate(a)

elseif a ∈ Set then EvalSet(a)

elseif a ∈ Reset then EvalReset(a)

elseif a ∈ TimerActive then EvalTimerActive(a)

elseif a ∈ Stop then EvalStop(a)

elseif a ∈ SystemValue then EvalSystemValue(a)

elseif a ∈ AnyValue then EvalAnyValue(a)

elseif a ∈ SetRangeCheckValue then EvalSetRangeCheckValue(a)

elseif a ∈ Scope then EvalScope(a)

elseif a ∈ Skip then EvalSkip(a)

elseif a ∈ Break then EvalBreak(a)

elseif a ∈ Continue then EvalContinue(a)

elseif a ∈ EnterStateNode then EvalEnterStateNode(a)

elseif a ∈ LeaveStateNode then EvalLeaveStateNode(a)

endif

##### F3.2.1.4.2 Primitive Var

Explanation

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

Representation

The domain Var is defined as a Cartesian product of the domain Variable-identifier of variable names and domain ContinueLabel of labels.

Var =def Variable-identifier × ContinueLabel

Behaviour

If the value of a variable in the current state of the executing agent is undefined, the UndefinedVariable exception is raised. Otherwise the value of a variable in the current state of the executing agent is determined by function eval and is written at Self.currentLabel. In order to avoid conflicts with other agents, the function value takes a further argument of type Agent, which identifies the owner of the value. Additionally, the label which determines the next rule to be fired is set to the given continue label.

EvalVar(a:Var) ≡

if eval(a.s-Variable-identifier, Self.stateAgent.state, Self.currentStateId) = undefined then

raise(UndefinedVariable)

else

value(Self.currentLabel, Self) := eval(a.s-Variable-identifier,

Self.stateAgent.state, Self.currentStateId)

Self.currentLabel := a.s-ContinueLabel

endif

Reference sections

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

##### F3.2.1.4.3 Primitive OperationApplication

Explanation

The OperationApplication primitive models the application of operators. Procedures without procedure body are called functional or predefined procedures. In this sense, all built-in operators such as +, - on the set of integers are predefined procedures. A predefined procedure is executed by function compute: a non-functional operation, which is handled with function dispatch that determines (depending on the current values) the correct procedure identifier.

Representation

OperationApplication =def Procedure × ValueLabel\* × ContinueLabel

Behaviour

EvalOperationApplication(a:OperationApplication) ≡

if functional(a.s-Procedure, values(a.s-ValueLabel-seq, Self)) then

value(Self.currentLabel, Self):= compute(a.s-Procedure, values(a.s-ValueLabel-seq, Self))

Self.currentLabel:= a.s-ContinueLabel

else

let pd: Procedure-definition = idToNodeAS1(

dispatch(a.s-Procedure, values(a.s-ValueLabel-seq, Self))) in

CreateProcedure(pd, Self.currentLabel, a.s-ContinueLabel)

endlet

endif

Reference sections

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

##### F3.2.1.4.4 Primitive Call

Explanation

The call primitive models procedure calls, or method invocations. It is used within the evaluation of expressions and actions. An action of type Call is defined as a tuple consisting of an identifier of the called procedure, a sequence of value labels and variable identifiers, and a continue label. In-parameters are represented by value labels, in/out-parameters by variable identifiers. The macro EvalCall creates a new context (e.g., new local scope for variables, for names of its states and connectors) and saves the old context, which in turn is restored by the corresponding return.

Representation

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

CallParam =def ValueLabel ∪ Variable-identifier

Call =def Procedure-identifier × CallParam\* × ValueLabel × ContinueLabel

Behaviour

EvalCall(a:Call) ≡

let pd: Procedure-definition = a.s-Procedure-identifier.idToNodeAS1 in

CreateProcedure(pd, a.s-ValueLabel, a.s-ContinueLabel)

endlet

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

SaveProcedureControlBlock(sn:StateNode, cl:ContinueLabel) ≡

sn.agentMode1 := Self.agentMode1

sn.agentMode2 := Self.agentMode2

sn.agentMode3 := Self.agentMode3

sn.agentMode4 := Self.agentMode4

sn.agentMode5 := Self.agentMode5

sn.currentStateId := Self.currentStateId

sn.currentLabel := Self.currentLabel

sn.continueLabel := cl

sn.currentParentStateNode := Self.currentParentStateNode

sn.previousStateNode := Self.previousStateNode

sn.callingProcedureNode := Self.callingProcedureNode

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

Reference sections

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

##### F3.2.1.4.5 Primitive Return

Explanation

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

Representation

Return =def () × (ValueLabel ∪ StateExitPoint)

Behaviour

EvalReturn(a: Return) ≡

if a.s-implicit ∈ ValueLabel then

EvalExitProcedure(a.s-implicit )

else

EvalExitCompositeState(a.s-implicit)

endif

EvalExitProcedure(vl: ValueLabel) ≡

value(Self.callingProcedureNode.resultLabel, Self) := value(vl, Self)

RestoreProcedureControlBlock(Self.callingProcedureNode)

EvalExitCompositeState(sep: StateExitPoint) ≡

Self.stateNodeToBeExited :=

mk**-**StateNodeWithExitPoint(Self.currentParentStateNode, sep)

Self.agentMode3 := exitingCompositeState

RestoreProcedureControlBlock(sn:StateNode) ≡

Self.agentMode1 := sn.agentMode1

Self.agentMode2 := sn.agentMode2

Self.agentMode3 := sn.agentMode3

Self.agentMode4 := sn.agentMode4

Self.agentMode5 := sn.agentMode5

Self.currentStateId := sn.currentStateId

Self.currentLabel := sn.continueLabel

Self.continueLabel := sn.continueLabel

Self.currentParentStateNode := sn.currentParentStateNode

Self.previousStateNode := sn.previousStateNode

Self.callingProcedureNode := sn.callingProcedureNode

Reference sections

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

##### F3.2.1.4.6 Primitive Task

Explanation

The Task primitive is used for the evaluation of assignments. An action of type Task is defined as a tuple consisting of a variable name, a value label and a continue label. The variable name becomes as value within the state of the executing agent the value stored at value label.

Representation

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

Task =def Variable-identifier × ValueLabel × Boolean × ContinueLabel

Behaviour

The assignment is mainly realized by means of macro Assign. Within the state of the executing agent the corresponding variable is set to the value stored at value label.

EvalTask(a:Task) ≡

Assign(a.s-Variable-identifier, value*(*a.s-ValueLabel, Self), Self.stateAgent.state,

Self.currentStateId)

Self.currentLabel := a.s-ContinueLabel

Reference Sections

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

##### F3.2.1.4.7 Primitive AssignParameters

Explanation

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

Representation

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

AssignParameters =def Variable-identifier × Nat × ContinueLabel

Behaviour

EvalAssignParameters(a:AssignParameters) ≡

let v = Self.currentSignalInst.plainSignalValues[a.s-Nat] in

Assign(a.s-Variable-identifier, v, Self.stateAgent.state, Self.currentStateId)

endlet

Self.currentLabel := a.s-ContinueLabel

Reference sections

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

##### F3.2.1.4.8 Primitive Equality

Explanation

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

Representation

Equality =def ValueLabel × ValueLabel × ContinueLabel × Boolean

The Boolean is true for equality and false for negative equality.

Behaviour

EvalEquality (a:Equality) ≡

if ( a.s- Boolean ∧ (value(a.s-ValueLabel, Self) = value(a.s2-ValueLabel, Self))) ∨

(¬ a.s- Boolean ∧ ¬(value(a.s-ValueLabel, Self) = value(a.s2-ValueLabel, Self))) then

value(a.s-ContinueLabel, Self) := mk**-**SDLBoolean(true, BooleanType)

else

value(a.s-ContinueLabel, Self) := mk**-**SDLBoolean(false, BooleanType)

endif

Self.currentLabel := a.s-ContinueLabel

Reference sections

No references.

##### F3.2.1.4.9 Primitive Decision

Explanation

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

Representation

A decision in Decision consists of a value label and a set of answer. An answer in Answer is a tuple consisting of a value label and a continue label.

Decision =def ValueLabel × Answer-set × [ContinueLabel]

Answer =def ValueLabel × ContinueLabel

Behaviour

Macro EvalDecision chooses an answer such that the decision-value given by the corresponding value label coincides with the answer-value.

EvalDecision(d:Decision) ≡

if value(d.s-ValueLabel, Self) ∈ { value(an.s-ValueLabel, Self) | an ∈ d.s-Answer-set } then

choose an: an ∈ d.s-Answer-set ∧

value(d.s-ValueLabel, Self) = value(an.s-ValueLabel, Self)

Self.currentLabel := an.s-ContinueLabel

endchoose

elseif d.s-ContinueLabel ≠ undefined then

Self.currentLabel := d.s-ContinueLabel

else raise(NoMatchingAnswer)

endif

Reference sections

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

##### F3.2.1.4.10 Primitive Output

Explanation

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

Representation

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

Output =def Signal × ValueLabel\* × ValueLabel × ValueLabel × [ValueLabel] × ViaArg × ContinueLabel

Further study needed to ensure *Expression* or *Encoded-expression* is handled in an Output.

Behaviour

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

EvalOutput(a:Output) ≡ // this invoked invoked only from Eval in F3.2.1.4.1 Action evaluation

SignalOutput(a.s-Signal, values(a.s-ValueLabel-seq, Self),

a.s1-ValueLabel, // activation delay

a.s2-ValueLabel, // signal priority

if a.s3-ValueLabel = undefined then undefined else value(a.s3-ValueLabel, Self) endif,

a.s-ViaArg)

Self.currentLabel := a.s-ContinueLabel

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

• possible destinations,

• potential receivers and

• admissible paths,

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

SignalOutput(s:Signal, vSeq:Value\*, delay:Duration, priority:Nat,

toArg:[ToArg], viaArg:ViaArg) ≡

let invReference = (if toArg ∈ Pid then

s.idToNodeAS1 ∉ toArg.s-Interface-definition.s-Signal-definition-set

else false endif)

in

if invReference then

raise(InvalidReference)

else

choose g: g ∈ (Self.outgates ∪ Self.ingates) ∧ applicable(s, toArg, viaArg, g, undefined)

extend PlainSignalInst with si

si.plainSignalType:= s

si.plainSignalValues := vSeq

si.delay = delay

si.priority = priority

si.toArg := toArg

si.viaArg := viaArg

si.plainSignalSender := Self.selfPid

Insert(si, now, g)

endextend

endchoose

endif

endlet

Reference sections

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

##### F3.2.1.4.11 Primitive Create

Explanation

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

Representation

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

Create =def Agent-identifier × ValueLabel\* × ContinueLabel

Behaviour

EvalCreate(a:Create) ≡

let sas = take({as ∈ SdlAgentSet: as.agentAS1 = a.s-Agent-identifier.idToNodeAS1 }) in

if sas.agentAS1.s-Number-of-instances.s-Maximum-number ≠ undefined then

let n = |{ sa ∈ SdlAgent: sa.owner = sas }| in

if n < sas.agentAS1.s-Number-of-instances.s-Maximum-number then

CreateAgent(sas, Self, sas.agentAS1)

else

Self.offspring := nullPid

endif

endlet

else

CreateAgent(sas, Self, sas.agentAS1)

endif

endlet

Self.currentLabel := a.s-ContinueLabel

Reference sections

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

##### F3.2.1.4.12 Primitive Set

Explanation

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

Representation

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

Set =def TimeLabel × Timer × ValueLabel\* × ContinueLabel

Domains

TimeLabel =def ValueLabel

Behaviour

Macro EvalSet defines the setting of a timer by macro SetTimer.

EvalSet(a:Set) ≡

SetTimer(a.s-Timer, values(a.s-ValueLabel-seq, Self), semvalueReal(value(a.s-TimeLabel,Self)))

Self.currentLabel := a.s-ContinueLabel

Reference sections

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

##### F3.2.1.4.13 Primitive Reset

Explanation

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

Representation

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

Reset =def Timer × ValueLabel\* × ContinueLabel

Behaviour

Macro EvalReset specifies a reset of a timer with macro ResetTimer.

EvalReset(a:Reset) ≡

ResetTimer(a.s-Timer, values( a.s-ValueLabel-seq, Self))

Self.currentLabel := a.s-ContinueLabel

Reference sections

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

##### F3.2.1.4.14 Primitive TimerActive and Primitive TimerRemaining

Explanation

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

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

Representation

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

TimerActive =def Timer × ValueLabel\* × ContinueLabel

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

TimerRemaining =def Timer × ValueLabel\* × ContinueLabel

Behaviour

Macro EvalTimerActive specifies the evaluation of a timer active expression.

EvalTimerActive(t:TimerActive) ≡

let tmi = mk**-**TimerInst(Self.selfPid, t.s-Timer, values( t.s-ValueLabel-seq, Self) ) in

value(Self.currentLabel, Self) := mk**-**SDLBoolean(active(tmi), BooleanType)

Self.currentLabel := t.s-ContinueLabel

endlet

Macro EvalTimerRemaining specifies the evaluation of a timer remaining duration.

EvalTimerRemaining(t:TimerRemaining) ≡

let tmi = mk**-**TimerInst(Self.selfPid, t.s-Timer, values( t.s-ValueLabel-seq, Self) ) in

if active(tmi) then

value(Self.currentLabel, Self) := mk**-**SDLDuration ( tmi.arrival- now, DurationType)

else

mk**-**SDLDuration ( 0.0, DurationType)

endif

Self.currentLabel := t.s-ContinueLabel

endlet

Reference sections

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

##### F3.2.1.4.15 Primitive Raise (SDL‑2000 feature)

Explanation

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

Reference sections

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

##### F3.2.1.4.16 Primitive Stop

Explanation

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

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

Further study needed to ensure #set\_ and #get procedures (see F2.2.5.1.3) are handled.

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

Representation

Stop =def ()

Behaviour

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

EvalStop(a:Stop) ≡

if Self.agentAS1.s-Number-of-instances.s-Lower-bound <

|{ sa ∈ SdlAgent: sa.agentAS1 = Self.agentAS1 }|

then

Self.agentMode2 := stopping

else

raise(OutOfRange)

endif

Reference sections

Clause F3.2.3.2.18.

##### F3.2.1.4.17 Primitive SystemValue

Explanation

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

Representation

SystemValue =def ValueKind × ContinueLabel

ValueKind =def { kNow, kSelf, kParent, kOffspring, kSender,kActiveAgents}

Behaviour

EvalSystemValue(a: SystemValue) ≡

value(Self.currentLabel, Self) :=

case a.s-ValueKind of

| kNow => **mk-**SDLTime(now, TimeType)

| kSelf=> Self.selfPid

| kParent=> Self.parent

| kOffspring=> Self.offspring

| kSender=> Self.sender

| kActiveAgents=>

**mk-**SDLInteger(|{ sa ∈ SdlAgent: sa.agentAS1 = Self.agentAS1 }|, IntegerType)

endcase

Self.currentLabel := a.s-ContinueLabel

##### F3.2.1.4.18 Primitive AnyValue

Explanation

The AnyValue primitive computes the any expression.

Representation

AnyValue =def Sort-identifier × ContinueLabel

Behaviour

EvalAnyValue(a: AnyValue) ≡

value(Self.currentLabel, Self) := selectAnyValue( a.s-Sort-identifier)

Self.currentLabel := a.s-ContinueLabel

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

selectAnyValue(id: Sort-identifier): Value =def

if id.idToNodeAS1 ∈ Interface-definition then nullPid

else take( {v | v ∈ Value ∧ v.sort =id } )

endif

##### F3.2.1.4.19 Primitive SetRangeCheckLabel

Explanation

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

Representation

SetRangeCheckValue =def ValueLabel × ContinueLabel

static rangeCheckValue: → Label

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

Behaviour

EvalSetRangeCheckValue(a: SetRangeCheckValue) ≡

value(rangeCheckValue, Self) := value(a.s-ValueLabel, Self)

Self.currentLabel := a.s-ContinueLabel

##### F3.2.1.4.20 Primitive Scope

Explanation

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

Representation

Scope =def Connector-name × Variable-definition-set × StartLabel × StepLabel × ContinueLabel

StepLabel =def Label

Behaviour

EvalScope(a:Scope) ≡

CreateCompoundNodeVariables(Self, a)

Self.currentLabel := a.s-StartLabel

Reference sections

See also clause F3.2.3.1.8.

##### F3.2.1.4.21 Primitive Skip

Explanation

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

Representation

Skip =def () × (Connector-name ∪ ContinueLabel)

Behaviour

EvalSkip(a:Skip) ≡

if a.s-implicit ∈ Connector-name then

Self.stateNodeChecked := Self.currentParentStateNode

Self.currentConnector := **mk-**StateNodeWithConnector(Self.currentParentStateNode, a.s-implicit)

Self.agentMode2 := selectingTransition

Self.agentMode3 := startSelection

else

Self.currentLabel := a.s-implicit

endif

Reference sections

Clause F3.2.3.2.8.

##### F3.2.1.4.22 Primitive Break

Explanation

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

Representation

Break =def () × (Connector-name)

Behaviour

EvalBreak(a:Break) ≡

if scopeName(Self, Self.currentStateId) = a.s-Connector-name then

Self.currentLabel := scopeContinueLabel(Self, Self.currentStateId)

endif

Self.currentStateId := caller(Self.stateAgent.state, Self.currentStateId)

##### F3.2.1.4.23 Primitive Continue

Explanation

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

Representation

Continue =def () × (Connector-name)

Behaviour

EvalContinue(a:Continue) ≡

if scopeName(Self, Self.currentStateId) = a.s-Connector-name then

Self.currentLabel := scopeStepLabel(Self, Self.currentStateId)

else

Self.currentStateId := caller(Self.stateAgent.state, Self.currentStateId)

endif

##### F3.2.1.4.24 Primitive EnterStateNode

Explanation

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

Representation

EnterStateNode =def ( State-name ∪ HISTORY ) × StateEntryPoint × ValueLabel\*

Behaviour

EvalEnterStateNode(a:EnterStateNode) ≡

let enterName: (State-name ∪ HISTORY) = a.s-implicit in

if enterName = HISTORY then

Self.stateNodesToBeEntered :=

{mk**-**StateNodeWithEntryPoint(Self.previousStateNode, HISTORY)}

else

choose sn: sn ∈ StateNode ∧ sn.stateName = enterName∧

sn.stateNodeKind = stateNode ∧ sn.parentStateNode = Self.currentParentStateNode

Self.stateNodesToBeEntered :=

{mk**-**StateNodeWithEntryPoint(sn, a.s-StateEntryPoint)}

endchoose

endif

Self.agentMode3 := enteringStateNode

Self.agentMode4 := startPhase

Self.currentLabel := undefined

Self.continueLabel := undefined

endlet

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

Reference sections

See also clause F3.2.3.2.15.

##### F3.2.1.4.25 Primitive LeaveStateNode

Explanation

State nodes are left at the start of transitions.

Representation

LeaveStateNode =def State-name × ContinueLabel

Behaviour

EvalLeaveStateNode(a:LeaveStateNode) ≡

choose sn: sn ∈ StateNode ∧ sn.stateName = a.s-State-name ∧

sn.stateNodeKind = stateNode ∧ sn.parentStateNode = Self.currentParentStateNode

// assertion: sn = Self.previousStateNode

Self.stateNodesToBeLeft := collectCurrentSubStates(sn)

endchoose

Self.agentMode3 := leavingStateNode

Self.agentMode4 := leavePhase

Self.currentLabel := undefined

Self.continueLabel := a.s-ContinueLabel

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

Reference sections

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

#### F3.2.1.5 Undefined behaviour

Undefined behaviour is represented by the following program:

UndefinedBehaviour ≡

Self.program := Undefined-Behaviour-Program

Undefined-Behaviour-Program:

// the contents of this program is not defined

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

This local undefinedness condition is in fact global as the program Undefined-Behaviour-Program could involve setting program for all agents.

### F3.2.2 Compilation function

The following two functions form the interface between the compilation and the dynamic semantics. For all the behaviour parts that involve transitions, the corresponding runtime representation of the transitions is generated.

Further study needed in getStateTransitions for the optional inclusion of SemTransition for an optional *State-timer*, and an extra parameter on SemTransition for the optional gate of signal inputs, but will be undefined for the alternatives with NONE instead of Signal-identifier.

getStateTransitions(s: State-node): SemTransition-set =def

{ mk**-**SemTransition(i.s-Signal-identifier,

if i.s-Provided-expression = undefined then

undefined

else

i.s-Provided-expression.startLabel

endif,

i.s-Priority-name,

i.s-Transition.startLabel,

undefined)

| i ∈ s.s-Input-node-set } ∪

{ mk**-**SemTransition(NONE, sp.s-Provided-expression.startLabel,

undefined, sp.s-Transition.startLabel, undefined)

| sp ∈ s.s-Spontaneous-transition-set } ∪

{ mk**-**SemTransition(NONE, c.s-Continuous-expression.startLabel,

c.s-Priority-name, c.s-Transition.startLabel, undefined)

| c ∈ s.s-Continuous-signal-set } ∪

{ mk**-**SemTransition(NONE, undefined, undefined, c.s-Transition.startLabel,

if c.s-State-exit-point-name = undefined then DEFAULT else c.s-State-exit-point-name endif)

| c ∈ s.s-Connect-node-set }

getStateStartTransitions(sn: State-start-node): StartTransition=def

mk**-**StartTransition(sn.s-Transition.startLabel, sn.s-State-entry-point-name)

getNamedStartTransitions(sn: Named-start-node): StartTransition=def

mk**-**StartTransition(sn.s-Transition.startLabel, sn.s-State-entry-point-name)

getProcStartTransitions(sn: Procedure-start-node): StartTransition=def

mk**-**StartTransition(sn.s-Transition.startLabel, undefined)

getStartTransitions(s: (State-start-node ∪ Named-start-node ∪ Procedure-start-node)-set):

StartTransition-set =def

{ if sn ∈ State-start-node then getStateStartTransitions(sn)

elseif sn ∈Named-start-node then getNamedStartTransitions(sn)

elseif sn ∈ Procedure-start-node then getProcStartTransitions(sn)

endif | sn ∈ s }

getFreeActions(actions: Free-action-set): FreeAction-set =def

{ mk**-**FreeAction(f.s-Connector-name, f.s-Transition.startLabel) | f ∈ actions }

Here we present the function that compiles an SDL‑2010 state machine description into an ASM representation. A special *labelling* of graph nodes is used to model specific control-flow information. Intuitively, node labels relate individual operations of an SDL‑2010 agent to transition rules in the resulting SAM model. The effect of state transitions of SDL‑2010 agents is then modelled by firing the related transition rules in an analogous order.

Labels are abstractly represented by a static domain Label.

static domain Label

To start with the compilation, we first need a function to find unique labels for a syntactic entity. The second argument is introduced to allow for more than one such label within the same SDL‑2010 pattern.

monitored uniqueLabel: DefinitionAS1 × Nat → Label

For this function, it holds that

constraint ∀ d1, d2 ∈ DefinitionAS1: ∀ i1, i2 ∈ Nat:

uniqueLabel(d1, i1) = uniqueLabel(d2, i2) ⇔ (d1=d2 ∧ i1=i2)

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

setToSeq(s: X-set): X\* =def

if s = ∅ then empty else

let el = c.take in

< el > ⁀ setToSeq(s \ { el })

endlet

endif

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

compile: DefinitionAS1 → Behaviour

compileExpr: DefinitionAS1 × Label → Behaviour

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

The two compilation functions are gradually introduced by defining a series of compilation patterns and the corresponding results; each individual pattern is uniquely associated with a certain type of node in the AST to be compiled. Afterwards, the function startLabel is defined also with a series of patterns in clause F3.2.2.4.

#### F3.2.2.1 States and triggers

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

compile(a: DefinitionAS1): Behaviour =def

case a of

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

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

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

Further study needed for statetimer in State-node below: in particular that State-timer is handled in *compile*.

| v=Variable-definition( name, \*, \*, init) =>

if init ≠ undefined then

compileExpr(init, uniqueLabel(v,1)) ∪

{mk**-**Primitive(uniqueLabel(v,1), mk**-**Task(name, uniqueLabel(init,1), false, undefined)) }

else ∅

endif

| State-transition-graph( start, states, freeActions) =>

compile(start) ∪

**U**{ compile(s) | s ∈ states } ∪

**U**{ compile(f) | f ∈ freeActions }

| Procedure-graph( start, states, freeActions) =>

compile(start) ∪

**U**{ compile(s) | s ∈ states } ∪

**U**{ compile(f) | f ∈ freeActions }

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

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

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

| State-node(\*, \*, inputs, exits, statetimer) =>

**U**{ compile(i) | i ∈ inputs } ∪

if exits = < spontaneous∈ Spontaneous-transition***-set,*** continuous ∈ Continuous-signal***-set***> then

**U**{ compile(s) | s ∈ spontaneous } ∪

**U**{ compile(c) | c ∈ continuous }

else

**U** { compile(c) | c ∈ exits.s-Connect-node***-set*** }

endif ∪ compile(statetimer)

| i = Input-node(\*, \*, \*, provided, \*, vars, transition) =>

if provided = undefined then ∅ else compileExpr(provided, undefined) endif ∪

{ mk**-**Primitive(uniqueLabel(i,idx),

if vars[idx] ≠ undefined then

mk**-**AssignParameters(vars[idx], idx,

uniqueLabel(i,idx))

else mk**-**Skip( uniqueLabel(i,idx))

endif)

| idx ∈ toSet(1..vars.length -1) } ∪

{ mk**-**Primitive(uniqueLabel(i, vars.length),

if vars[vars.length] ≠ undefined then

mk**-**AssignParameters(vars[vars.length], vars.length, transition.startLabel)

else mk**-**Skip(transition.startLabel)

endif)

} ∪

compile(transition)

| Spontaneous-transition(provided, transition) =>

if provided = undefined then ∅ else compileExpr(provided, undefined) endif ∪

compile(transition)

| Continuous-signal(ce, \*, transition) =>

compileExpr(ce, undefined) ∪

compile(transition)

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

| State-timer(te, stimerId, exprList, transition) =>

if exprList = empty then ∅

else compileExpr(exprList.last, te.startLabel) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

compileExpr(te, uniqueLabel(s,1)) ∪

{mk**-**Primitive(uniqueLabel(s,1),

mk**-**Set(uniqueLabel(te,1), stimerId, <uniqueLabel(e,1) | e in exprList >, next)) }∪

compile(transition)

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

| t=Transition(nodes, endnode) =>

if t.parentAS1.parentAS1.s-State-name ≠ undefined then

{mk**-**Primitive(uniqueLabel(a,1),

mk**-**LeaveStateNode(t.parentAS1.parentAS1.s-State-name,

startLabel(if nodes = empty then endnode else nodes.head endif))) }

else ∅ endif ∪

compileNodes ∪

compile(endnode)

where

compileNodes: Behaviour =def

if nodes = empty then ∅

else compileExpr(nodes.last, endnode. startLabel) ∪

**U**{ compileExpr(nodes[i], nodes[i+1]. startLabel) | i ∈ 1..nodes.length - 1 }

endif

endwhere

#### F3.2.2.2 Terminators

| Terminator(terminator) =>compile(terminator)

| n=Named-nextstate(stateName, undefined) =>

{mk**-**Primitive(uniqueLabel(n,1),

mk**-**EnterStateNode(stateName, undefined, empty)) }

| n=Named-nextstate(stateName, Nextstate-parameters(exprList, entry)) =>

if exprList = empty then ∅

else compileExpr(exprList.last, uniqueLabel(n,1)) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

{mk**-**Primitive(uniqueLabel(n,1),

mk**-**EnterStateNode(stateName, entry, <uniqueLabel(e,1) | e in exprList >)) }

| n= Dash-nextstate(HISTORY) =>

{mk**-**Primitive(uniqueLabel(n,1), mk**-**EnterStateNode(HISTORY, undefined, empty)) }

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

| s=Stop-node() =>

{mk**-**Primitive(uniqueLabel(s,1), mk**-**Stop()) }

| a=Action-return-node() =>

{mk**-**Primitive(uniqueLabel(a,1), mk**-**Return

(if parentAS1ofKind(a,Composite-state-type-definition).parentAS1 ∈

Composite-state-type-definition then DEFAULT else undefined endif)) }

| v=Value-return-node(expr) =>

compileExpr(expr, uniqueLabel(v,1)) ∪

{mk**-**Primitive(uniqueLabel(v,1), mk**-**Return(uniqueLabel(expr,1))) }

| n=Named-return-node(name) =>

{mk**-**Primitive(uniqueLabel(n,1), mk**-**Return(name)) }

| j= Join-node(connector) =>

{mk**-**Primitive(uniqueLabel(j,1), mk**-**Skip(connector)) }

| b= Break-node(connector) =>

{mk**-**Primitive(uniqueLabel(b,1), mk**-**Break(connector)) }

| c= Continue-node(connector) =>

{mk**-**Primitive(uniqueLabel(c,1), mk**-**Continue(connector)) }

| d=Decision-body(question, answerset, elseanswer) =>

(let aseq = answerset.setToSeq in

compileExpr(question, aseq[1].startLabel) ∪

{ compileExpr(aseq[idx].s-implicit,

if idx=aseq.length then uniqueLabel(d, 1) else aseq[idx+1].startLabel endif)

| idx ∈ toSet(1..aseq.length) } ∪

{ mk**-**Primitive(uniqueLabel(d, 1),

mk**-**Decision(uniqueLabel(question, 1),

{ mk**-**Answer(uniqueLabel(ans.s-implicit, 1), ans.s-Transition.startLabel)

| ans ∈ answerset },

if elseanswer=undefined then undefined else elseanswer.s-Transition endif)) }

endlet) ∪

**U**{ compile(ans.s-Transition) | ans ∈ answerset } ∪

compile(elseanswer.s-Transition)

| d=Any-decision(transset) =>

**U**{ compile(t) | t ∈ transset }

This concludes the definition of the compile function.

endcase // end of the compile function definition

#### F3.2.2.3 Actions

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

compileExpr(a: DefinitionAS1, next: Label): Behaviour =def

case a of

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

Further study needed below for *Expression* or *Encoded-expression* in an *Output-node*.

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

| a=Assignment(id, expr) =>

compileExpr(expr, uniqueLabel(a,1)) ∪

{mk**-**Primitive(uniqueLabel(a,1), mk**-**Task(id, uniqueLabel(expr,1), false, next) )}

| o=Output-node(sig ∈ Signal-identifier, exprList, *delay, priority*, dest, destnum, via ) =>

if exprList = empty then ∅

else compileExpr(exprList.last,

if dest ∈ Expression then dest.startLabel

elseif destnum ∈ Expression then destnum.startLabel

else uniqueLabel(o,1) endif) ∪

**U**{ compileExpr(exprList[i], exprList[i+1].startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

compileExpr(delay, uniqueLabel(o,1)) ∪

compileExpr(priority, uniqueLabel(o,1)) ∪

if dest ∈ Expression then compileExpr(dest, uniqueLabel(o,1)) else ∅ endif ∪

if destnum ∈ Expression then compileExpr(destnum, uniqueLabel(o,1)) else ∅ endif ∪

{mk**-**Primitive(uniqueLabel(o,1),

mk**-**Output(sig, <uniqueLabel(e,1) | e in exprList >,

uniqueLabel(delay,1), uniqueLabel(priority,1), toPid(dest, destnum), via, next)) }

where

toPid (dest: *Expression* ∪ *Agent-identifier* ∪ **THIS**, destnum: [ *Expression* ] ): Pid =def

if dest ∈ Expression then dest

else if dest ∈ *Agent-identifier* then

let sas = take({ as ∈ SdlAgentSet : as.identifier1 = dest}) in

if destnum ∈ Expression then sas.agentSetPids()[destnum]

else take(toSet (sas.agentSetPids()))

endif

endlet

else // dest is **THIS**

if destnum ∈ Expression then Self.agentSetPids()[destnum]

else take(toSet (Self.agentSetPids()))

endif

endif

endwhere

| o=Output-node(sig ∈ Expression, *delay, priority*, dest, via, next ) =>

TBD // Further study needed

| o=Output-node(sig ∈ Encoded-expression, *delay, priority*, dest, via, next ) =>

TBD // Further study needed

| c=Create-request-node(agentId ∈ Agent-identifier, exprList) =>

if exprList = empty then ∅

else compileExpr(exprList.last, uniqueLabel(c,1)) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

{mk**-**Primitive(uniqueLabel(c,1),

mk**-**Create(agentId, <uniqueLabel(e,1) | e in exprList >, next)) }

| c=Create-request-node(**THIS**, exprList) =>

if exprList = empty then ∅

else compileExpr(exprList.last, uniqueLabel(c,1)) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

{mk**-**Primitive(uniqueLabel(c,1),

mk**-**Create(parentAS1ofKind(c,Agent-definition).s-Agent-identifier,

<uniqueLabel(e,1) | e in exprList >, next)) }

| c=Call-node(\*, procedureId, exprList) =>

if exprList = empty then ∅

else compileExpr(exprList.last, uniqueLabel(c,1)) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

(let paramDef = procedureId.idToNodeAS1.s-Procedure-formal-parameter-seq in

{mk**-**Primitive(uniqueLabel(c,1),

mk**-**Call(procedureId,

<( if paramDef[idx] ∈ In-parameter

then uniqueLabel(exprList[idx], 1)

else exprList[idx]

endif )

| idx in (1..exprList.length ) >, uniqueLabel(c,1),

next)) }

endlet)

| c=Compound-node(name, variables, initNodes, whileNode, trans, stepNodes) =>

{mk**-**Primitive(uniqueLabel(c,1),

mk**-**Scope(name, variables,

if initNodes = empty then trans.startLabel else initNodes.head.startLabel endif,

if stepNodes = empty then trans.startLabel else stepNodes.head.startLabel endif,

next)) } ∪

compileExpr(whileNode, undefined) ∪

compileExpr(trans, undefined) ∪

if stepNodes = empty then ∅

else compileExpr( stepNodes.last, trans.startLabel) ∪

**U**{ compileExpr( stepNodes[i], stepNodes[i+1]. startLabel) | i ∈ 1..stepNodes.length - 1 }

endif ∪

if initNodes = empty then ∅

else compileExpr( initNodes.last, trans.startLabel) ∪

**U**{ compileExpr( initNodes[i], initNodes[i+1]. startLabel) | i ∈ 1..initNodes.length - 1 }

endif

| w=While-node(whilexprs, final) =>

if whilexprs = empty then ∅

else compile(Decision-body(whilexpr.head, While-node(whilexprs.tail, final), undefined))

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

if exprList = empty then ∅

else compileExpr(exprList.last, expr.startLabel) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

compileExpr(expr, uniqueLabel(s,1)) ∪

{mk**-**Primitive(uniqueLabel(s,1),

mk**-**Set(uniqueLabel(expr,1), timerId, <uniqueLabel(e,1) | e in exprList >, next)) }

| r=Reset-node(timerId, exprList) =>

if exprList = empty then ∅

else compileExpr(exprList.last, uniqueLabel(r,1)) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

{mk**-**Primitive(uniqueLabel(r,1),

mk**-**Reset(timerId, <uniqueLabel(e,1) | e in exprList >, next)) }

| r=Range-condition(items) =>

(let iseq = items.setToSeq in

{mk**-**Primitive(uniqueLabel(r,1),

mk**-**OperationApplication(sdlTrue.idToNodeAS1, empty,

uniqueLabel(r, iseq.length+1))) } ∪

{ compileExpr(iseq[idx], uniqueLabel(r, idx)) | idx ∈ toSet(1..iseq.length) } ∪

{ mk**-**Primitive(uniqueLabel(r, idx),

mk**-**OperationApplication(sdlOr, // sdlOr only used here

< uniqueLabel(r, idx+1), uniqueLabel(iseq[idx],1) >,

// Further study needed to check uniqueLabel(r, idx+1) is correct here

if idx=1 then next else iseq[idx-1].startLabel endif))

| idx ∈ toSet(1..iseq.length) } ∪

{ mk**-**Primitive(uniqueLabel(r, 0), mk**-**Break(undefined)) }

endlet)

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

| o=Open-range(id, expr) =>

compileExpr(expr, uniqueLabel(o, 1)) ∪

{ mk**-**Primitive(uniqueLabel(o, 1),

mk**-**OperationApplication(id.idToNodeAS1,

< rangeCheckValue, uniqueLabel(expr, 1) >, next)) }

| c=Closed-range(r1, r2) =>

compileExpr(r1, r2.startLabel) ∪

compileExpr(r2, uniqueLabel(c, 1)) ∪

{ mk**-**Primitive(uniqueLabel(c, 1),

mk**-**OperationApplication(sdlAnd, // *sdlAnd* only used here.

< uniqueLabel(r1, 1), uniqueLabel(r2, 1) >, next)) }

// Further study needed to check uniqueLabel(r1, 1), uniqueLabel(r2, 1) is correct here

| l=Literal(id) =>

{mk**-**Primitive(uniqueLabel(l,1),

mk**-**OperationApplication(id.idToNodeAS1, empty, next)) }

| c=Conditional-expression(boolExpr, consExpr, altExpr) =>

compileExpr(boolExpr, uniqueLabel(c, 2)) ∪

compileExpr(consExpr, next) ∪

compileExpr(altExpr, next) ∪

{mk**-**Primitive(uniqueLabel(c,2),

mk**-**OperationApplication(sdlTrue.idToNodeAS1, empty, uniqueLabel(c, 1))) } ∪

{ mk**-**Primitive(uniqueLabel(c, 1),

mk**-**Decision(uniqueLabel(boolExpr, 1),

{ mk**-**Answer(uniqueLabel(c, 2), consExpr.startLabel) }, altExpr.startLabel)) }

| ep=Positive-equality-expression(first, second) =>

compileExpr(first, second.startLabel) ∪

compileExpr(second, uniqueLabel(ep,1)) ∪

{mk**-**Primitive(uniqueLabel(ep,1),

mk**-**Equality(uniqueLabel(first,1), uniqueLabel(second,1), next), true) }

| en=Negative-equality-expression(first, second) =>

compileExpr(first, second.startLabel) ∪

compileExpr(second, uniqueLabel(en,1)) ∪

{mk**-**Primitive(uniqueLabel(en,1),

mk**-**Equality(uniqueLabel(first,1), uniqueLabel(second,1), next), false) }

| o=Operation-application(id, exprList) =>

if exprList = empty then ∅

else compileExpr(exprList.last, uniqueLabel(o,1)) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

{mk**-**Primitive(uniqueLabel(o,1),

mk**-**OperationApplication(id.idToNodeAS1,

< uniqueLabel(e, 1) | e in exprList >,

next)) }

| r=Range-check-expression(expr, \*, range) =>

compileExpr(expr, uniqueLabel(r,2)) ∪

compileExpr(range, undefined) ∪

{mk**-**Primitive(uniqueLabel(r,2),

mk**-**SetRangeCheckValue(uniqueLabel(expr,1), uniqueLabel(r,1))) } ∪

{mk**-**Primitive(uniqueLabel(r,1),

mk**-**Scope(undefined, ∅, range.startLabel, undefined, next)) }

| v=Variable-access(id) =>

{mk**-**Primitive(uniqueLabel(v,1), mk**-**Var(id, next)) }

| n=Now-expression() =>

{mk**-**Primitive(uniqueLabel(n,1), mk**-**SystemValue(kNow, next)) }

| p=Parent-expression() =>

{mk**-**Primitive(uniqueLabel(p,1), mk**-**SystemValue(kParent, next)) }

| o=Offspring-expression() =>

{mk**-**Primitive(uniqueLabel(o,1), mk**-**SystemValue(kOffspring, next)) }

| s=Self-expression() =>

{mk**-**Primitive(uniqueLabel(s,1), mk**-**SystemValue(kSelf, next)) }

| s=Sender-expression() =>

{mk**-**Primitive(uniqueLabel(s,1), mk**-**SystemValue(kSender, next)) }

| a=Active-agents-expression() =>

{mk**-**Primitive(uniqueLabel(a,1), mk**-**SystemValue(kActiveAgents, next)) }

| t=Timer-active-expression(id, exprList) =>

if exprList = empty then ∅

else compileExpr(exprList.last, uniqueLabel(t,1)) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

{mk**-**Primitive(uniqueLabel(t,1),

mk**-**TimerActive(id, < uniqueLabel(e, 1) | e in exprList >, next)) }

| t=Timer-remaining-duration(id, exprList) =>

if exprList = empty then ∅

else compileExpr(exprList.last, uniqueLabel(t,1)) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

{mk**-**Primitive(uniqueLabel(t,1),

mk**-**TimerRemaining(id, < uniqueLabel(e, 1) | e in exprList >, next)) }

| a=Any-expression(id) =>

{mk**-**Primitive(uniqueLabel(a,1), mk**-**AnyValue(id, next)) }

| v=Value-returning-call-node(\*, procedureId, exprList) =>

if exprList = empty then ∅

else compileExpr(exprList.last, uniqueLabel(v,1)) ∪

**U**{ compileExpr(exprList[i], exprList[i+1]. startLabel) | i ∈ 1..exprList.length - 1 }

endif ∪

(let paramDef = procedureId.idToNodeAS1.s-Procedure-formal-parameter-seq in

{mk**-**Primitive(uniqueLabel(v,1),

mk**-**Call(procedureId,

< ( if paramDef[idx] ∈ In-parameter

then uniqueLabel(exprList[idx], 1)

else exprList[idx]

endif )

| idx in (1..exprList.length )>, uniqueLabel(v,1),

next)) }

endlet)

This concludes the definition of the expression compilation function.

endcase // end of the compileExpr function definition

#### F3.2.2.4 Start labels

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

startLabel(x: DefinitionAS1): Label =def

case x of

| v=Variable-definition(\*, \*, \*, init) =>

if init = undefined then undefined else init.startLabel endif

| s=State-start-node(trans) => startLabel(trans)

| p=Procedure-start-node(trans) => startLabel(trans)

| i=Input-node(\*, \*, \*, \*, \*, \*, trans) => startLabel(trans)

| s=Spontaneous-transition(\*, trans) => startLabel(trans)

| c=Continuous-signal(\*, \*, \*, trans) => startLabel(trans)

| c=Connect-node(\*, trans) => startLabel(trans)

| f=Free-action(\*, trans) => startLabel(trans)

| t=Transition(nodes, endnode) =>

if t.parentAS1.parentAS1 ∈ State-node then uniqueLabel(t,1) // insert the Leavestatenode

elseif nodes = empty then startLabel(endnode)

else startLabel(nodes.head)

endif

| g=Graph-node(action) => startLabel(action)

| a=Assignment(\*, expr) => startLabel(expr)

| o= Output-node(sig ∈ Signal-identifier, exprList, \*, \*, dest, \*, \*) =>

if dest ≠ undefined then startLabel(dest)

elseif exprList = empty then uniqueLabel(o,1)

else startLabel(exprList.head) endif

| o=Output-node(sig ∈ (Expression ∪ Encoded-expression), *\*, \**, dest, \*, \* ) =>

if dest ≠ undefined then startLabel(dest)

else uniqueLabel(o,1) endif

| c=Create-request-node(\*, exprList) =>

if exprList = empty then uniqueLabel(c,1) else exprList.head.startLabel endif

| c=Call-node(\*, \*, exprList) =>

if exprList = empty then uniqueLabel(c,1) else exprList.head.startLabel endif

| c=Compound-node(\*, \*, \*, \*, \*, \*, \*) => uniqueLabel(c,1)

| s= Set-node(when, \*, \*) => startLabel(when)

| r=Reset-node(\*, exprList) =>

if exprList = empty then uniqueLabel(r,1) else exprList.head.startLabel endif

| t=Terminator(terminator) => startLabel(terminator)

| n=Named-nextstate(\*, undefined) => uniqueLabel(n,1)

| n=Named-nextstate(\*, Nextstate-parameters(exprList, \*)) =>

if exprList = empty then uniqueLabel(n,1) else exprList.head.startLabel endif

| n=Dash-nextstate(\*) => uniqueLabel(n,1)

| s= Stop-node() => uniqueLabel(s,1)

| a=Action-return-node() => uniqueLabel(a,1)

| v=Value-return-node(\*) => uniqueLabel(v,1)

| n=Named-return-node(\*) => uniqueLabel(n,1)

| j= Join-node(\*) => uniqueLabel(j,1)

| b= Break-node(\*) => uniqueLabel(b,1)

| c= Continue-node(\*) => uniqueLabel(c,1)

| d=Decision-body(question, \*, \*, \*) => startLabel(question)

| d=Any-decision(\*) => uniqueLabel(d,1)

| a=Decision-answer (r, \*) => startLabel(r)

| n=Named-start-node(\*, trans) => startLabel(trans)

| o=Open-range(\*, expr) => startLabel(expr)

| c=Closed-range(\*, \*) => uniqueLabel(c,1)

| l=Literal(\*) => uniqueLabel(l,1)

| c=Conditional-expression(\*, \*, \*) => uniqueLabel(c,1)

| Positive-equality-expression(first, \*) => first.startLabel

| Negative-equality-expression(first, \*) => first.startLabel

| r=Range-check-expression(\*, expr) => expr.startLabel

| v=Variable-access(id) => uniqueLabel(v,1)

| o= Operation-application(\*, exprList) =>

if exprList = empty then uniqueLabel(o,1) else exprList.head.startLabel endif

| v=Identifier(\*, \*) => uniqueLabel(v,1)

| n=Now-expression() => uniqueLabel(n,1)

| s=Self-expression() => uniqueLabel(s,1)

| p=Parent-expression() => uniqueLabel(p,1)

| o=Offspring-expression() => uniqueLabel(o,1)

| s=Sender-expression() => uniqueLabel(s,1)

| t=Timer-active-expression(\*, exprList) =>

if exprList = empty then uniqueLabel(t,1) else exprList.head.startLabel endif

| t=Timer-remaining-duration(\*, exprList) =>

if exprList = empty then uniqueLabel(t,1) else exprList.head.startLabel endif

| a=Any-expression(\*) => uniqueLabel(a,1)

| v=Value-returning-call-node(\*, \*, exprList) =>

if exprList = empty then uniqueLabel(v,1) else exprList.head.startLabel endif

endcase

### F3.2.3 SDL‑2010 abstract machine programs

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

#### F3.2.3.1 System initialization

Starting from any pre-initial state of S0, the initialization rules describe a recursive *unfolding* of the specified system instance according to its initial hierarchical structure. For each SDL‑2010 agent instance, a corresponding ASM agent is created and initialized. Furthermore, ASM agents are created to model links and SDL‑2010 agent sets.

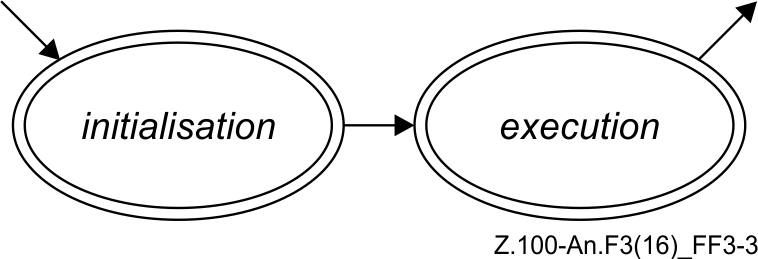


Figure F3-3 – Activity phases of SDL‑2010 agents and agent sets (level 1)

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

##### F3.2.3.1.1 Pre-initial system state

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

initially

if rootNodeAS1.s-Agent-definition ≠ undefined then

system.agentAS1 = rootNodeAS1.s-Agent-definition ∧

system.owner = undefined ∧

system.agentMode1 = initialisation ∧

system.program = Agent-Set-Program

else

system.program = undefined

endif

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

##### F3.2.3.1.2 Agent set creation, initialization, and removal

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

CreateAllAgentSets(ow:Agent, atd:Agent-type-definition) ≡

do forall ad: ad ∈ atd.collectAllAgentDefinitions

CreateAgentSet(ow, ad)

enddo

where

collectAllAgentDefinitions(atd: Agent-type-definition): Agent-definition-set =def

if atd.s-Agent-type-identifier = undefined then

atd.s-Agent-definition-set

else let typedef: Agent-type-definition = atd.s-Agent-type-identifier.idToNodeAS1 in

atd.s-Agent-definition-set ∪ typedef.collectAllAgentDefinitions

endlet

endif

endwhere

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

CreateAgentSet(ow:SdlAgent, ad:Agent-definition) ≡

let typedef: Agent-type-definition = ad.s-Agent-type-identifier.idToNodeAS1 in

extend Agent with sas

sas.agentAS1 := ad

sas.owner := ow

CreateAllGates (sas, typedef)

sas.program := Agent-Set-Program

sas.agentMode1 := initialisation

endextend

endlet

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

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

Agent-Set-Program:

if Self.agentMode1 = initialisation then

InitAgentSet

endif

if Self.agentMode1 = execution then

ExecAgentSet

endif

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

InitAgentSet ≡

let typedef: Agent-type-definition = Self.agentAS1.s-Agent-type-identifier.idToNodeAS1 in

if typedef.s-Agent-kind = SYSTEM then

CreateAllGates(Self, typedef)

endif

CreateAllAgents(Self, Self.agentAS1)

Self.agentMode1:= execution

endlet

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

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

RemoveAllAgentSets(ow:SdlAgent) ≡

do forall sas: sas ∈ SdlAgentSet ∧ sas.owner = ow

RemoveAgentSet(sas)

enddo

RemoveAgentSet(sas:SdlAgentSet) ≡

sas.owner := undefined

sas.program := undefined

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

##### F3.2.3.1.3 Agent creation, initialization, and removal

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

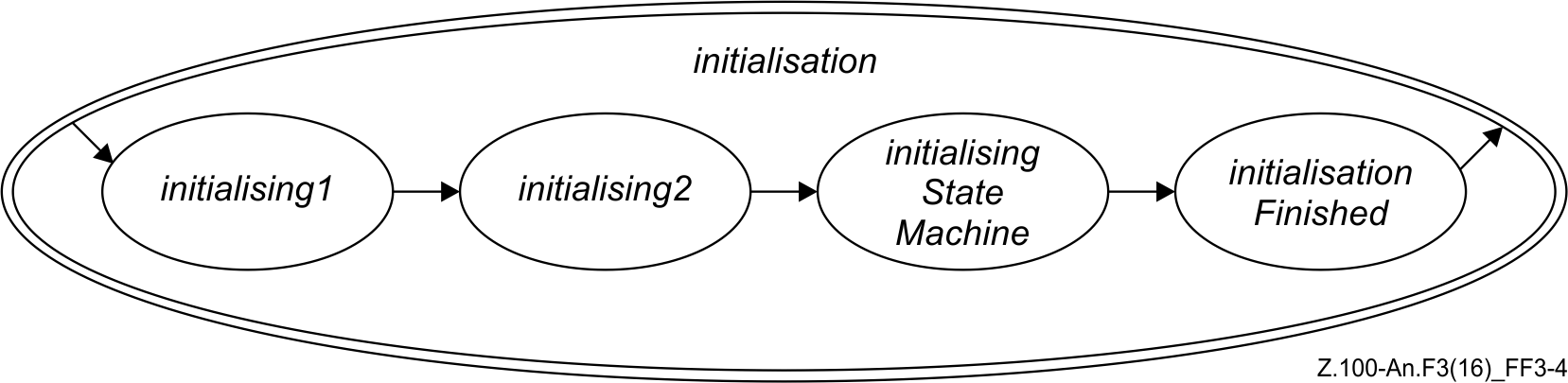


Figure F3-4 – Activity phases of SDL‑2010 agents: initialization (level 2)

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

CreateAllAgents(ow:SdlAgentSet, ad:Agent-definition) ≡

ow.agentSetPids := empty

do forall i: i ∈ 1..ad.s-Number-of-instances.s-Initial-number

CreateAgent(ow, undefined, ad)

enddo

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

CreateAgent(ow:SdlAgentSet, pa: [SdlAgent], ad:Agent-type-definition) ≡

extend Agent with sa

InitAgentControlBlock(sa, ow, pa, ad)

CreateInputPort(sa)

sa.agentMode1 := initialisation

sa.agentMode2 := initialising1

sa.program := Agent-Program

endextend

where

InitAgentControlBlock(sa: SdlAgent, ow:SdlAgentSet, pa: [SdlAgent],

ad:Agent-type-definition) ≡

sa.agentAS1 := ad

sa.owner := ow

sa.isActive := undefined

sa.currentStartNodes := ∅

sa.currentExitStateNodes := ∅

sa.currentConnector := undefined

sa.callingProcedureNode := undefined

sa.currentSignalInst := undefined

sa.parent := if pa ≠ undefined then pa.selfPid else undefined endif

sa.sender := nullPid

sa.offspring := nullPid

sa.selfPid := mk**-**Pid(sa, undefined)

if pa ≠ undefined then

pa.offspring := mk**-**Pid(sa, undefined)

endif

if ow.agentAS1.s-Agent-type-identifier.idToNodeAS1.s-Agent-kind = PROCESS then

sa.stateAgent := ow.owner.stateAgent

else // SYSTEM or BLOCK or other

sa.stateAgent := sa

endif

ow.agentSetPids := ow.agentSetPids⁀<sa.selfPid>

endwhere

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

Agent-Program:

if Self.agentMode1 = initialisation then

InitAgent

elseif Self.agentMode1 = execution then

if Self.ExecRightPresent then

ExecAgent

else

GetExecRight

endif

endif

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

InitAgent ≡

let myDefinition: Agent-type-definition = Self.agentAS1.s-Agent-type-identifier. idToNodeAS1 in

if Self.agentMode2 = initialising1 then

CreateAgentVariables(Self, myDefinition )

CreateAllAgentSets(Self, myDefinition )

CreateStateMachine(myDefinition .s-State-machine)

Self.agentMode2 := initialising2

elseif Self.agentMode2 = initialising2 then

CreateAllChannels(Self, myDefinition )

// no implicit links (done by DeliverSignals)

Self.agentMode2 := initialisingStateMachine

elseif Self.agentMode2 = initialisingStateMachine then

InitStateMachine

elseif Self.agentMode2 = initialisationFinished then

Self.agentMode1 := execution

Self.agentMode2 := startPhase

endif

endlet

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

Once the structure of the agent has been created, channels and links are established. Next, the state machine is initialized, i.e., a "hierarchical inheritance state graph" modelling the agent's state machine is unfolded in a sequence of steps. Finally, execution is triggered by setting the agent modes.

RemoveAgent(sa:SdlAgent) ≡

sa.owner.agentSetPids := removePid (sa.selfPid, sa.owner.agentSetPids)

RemoveAllLinks(sa)

sa.program := undefined

sa.owner := undefined

where

removePid (p: PID, plist: PID\*): PID\* =def

if plist = empty then empty

elseif plist.head = p then plist.tail

else < plist.head > ⁀ removePid (p, plist.tail)

endif

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

##### F3.2.3.1.4 Procedure creation and initialization

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

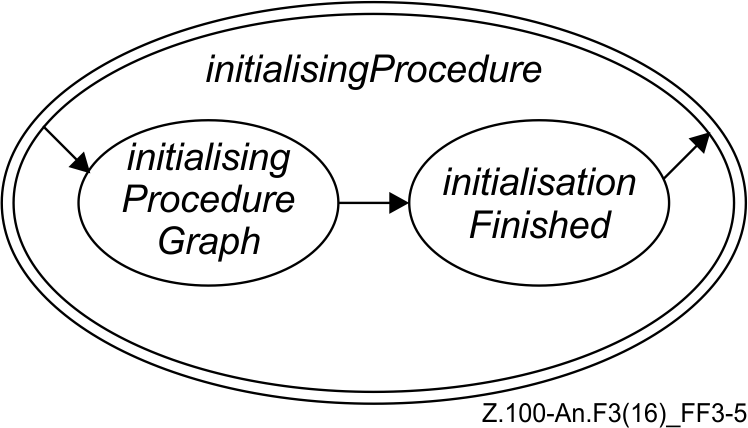


Figure F3-5 – Activity phases of SDL‑2010 agents: firing of transitions (level 4)

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

CreateProcedure(pd:Procedure-definition, vl: [ValueLabel], cl:[ContinueLabel]) ≡

CreateProcedureGraph(pd, vl, cl)

Self.agentMode3 := initialisingProcedure

Self.agentMode4 := initialisingProcedureGraph

InitProcedure ≡

if Self.agentMode4 = initialisingProcedureGraph then

InitProcedureGraph

elseif Self.agentMode4 = initialisationFinished then

Self.stateNodesToBeEntered :=

{mk**-**StateNodeWithEntryPoint (Self.currentProcedureStateNode, undefined)}

Self.agentMode3 := enteringStateNode

Self.agentMode4 := startPhase

Self.currentLabel := undefined

endif

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

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

##### F3.2.3.1.5 Gate creation

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

CreateAllGates(ow:Agent, atd: Agent-type-definition) ≡

do forall gd: gd ∈ atd.collectAllGateDefinitions

CreateGate(ow, gd)

enddo

where

collectAllGateDefinitions(atd: Agent-type-definition): Gate-definition-set =def

if atd.s-Agent-type-identifier = undefined then

atd.s-Gate-definition-set

else

let typedef: Agent-type-definition = atd.s-Agent-type-identifier.idToNodeAS1 in

atd.s-Gate-definition-set ∪

typedef.collectAllGateDefinitions

endlet

endif

endwhere

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

CreateGate(ow:Agent, gd:Gate-definition) ≡

if gd.s-In-signal-identifier-set ≠ ∅ then

extend Gate with g

g.myAgent := ow

g.gateAS1 := gd

g.schedule := empty

g.direction := inDir

endextend

endif

if gd.s-Out-signal-identifier-set ≠ ∅ then

extend Gate with g

g.myAgent := ow

g.gateAS1 := gd

g.schedule := empty

g.direction := outDir

endextend

endif

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

CreateInputPort(ow:Agent) ≡

extend Gate with g

g.myAgent := ow

g.gateAS1 := undefined

g.schedule := empty

g.direction := inDir

ow.inport := g

endextend

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

##### F3.2.3.1.6 Channel creation

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

CreateAllChannels(ow:Agent, atd:Agent-type-definition) ≡

do forall cd: cd ∈ atd.collectAllChannelDefinitions

CreateChannel(ow, cd)

enddo

where

collectAllChannelDefinitions(atd: Agent-type-definition): Channel-definition-set =def

if atd.s-Agent-type-identifier = undefined then

atd.s-Channel-definition-set

else

let typedef: Agent-type-definition **=** atd.s-Agent-type-identifier.idToNodeAS1 in

atd.s-Channel-definition-set ∪

typedef .collectAllChannelDefinitions

endlet

endif

endwhere

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

CreateChannel(ow:Agent, cd:Channel-definition) ≡

do forall cp: cp ∈ cd.s-Channel-path-set

CreateChannelPath(ow, cd.s-NODELAY, cp, cd)

Creating a channel amounts to creating the specified channel paths.

CreateChannelPath(ow:Agent, nd:[NODELAY], cp:Channel-path, cd:Channel-definition) ≡

let origDef: Gate-definition = cp.s-Originating-gate.idToNodeAS1 in

let destDef: Gate-definition = cp.s-Destination-gate.idToNodeAS1 in

choose fromGate: fromGate ∈ Gate ∧ fromGate.gateAS1= origDef ∧

(OuterGate(ow, fromGate, inDir) ∨ InnerGate(ow, fromGate, outDir) )

choose toGate: toGate ∈ Gate ∧ toGate.gateAS1 = destDef ∧

(OuterGate(ow, toGate, outDir) ∨ InnerGate(ow, toGate, inDir) )

CreateLink(ow,fromGate, toGate, nd, cp.s-Signal-identifier-set, cd)

endchoose

endchoose

where

OuterGate(ow: Agent, g: Gate, dir: Direction): Boolean =def

g.myAgent = ow.owner ∧ g.direction = dir

InnerGate(ow: Agent, g: Gate, dir: Direction): Boolean =def

g.myAgent.owner = ow ∧ g.direction = dir

endwhere

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

##### F3.2.3.1.7 Link creation and removal

Agents of type Link model the transport of signals. The behaviour of link agents is defined by the ASM program Link-Program.

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

CreateLink(ow:Agent, fromGate:Gate, toGate:Gate, nd:[NODELAY], w:In-signal-identifier-set,

cd:[Channel-definition]) ≡

extend Linkwith l

l.channelAS1 := cd

l.owner := ow

l.from := fromGate

l.to := toGate

l.noDelay := nd

l.with := w

l.program := Link-Program

endextend

Link-Program:

if Self.from.queue ≠ empty then

let si = Self.from.queue.head in

if applicable(si.signalType,si.toArg,si.viaArg,Self.from,Self) then

Delete(si,Self.from)

Insert(si,now+Self.delay,Self.to)

si*.*viaArg := si*.*viaArg *\*

{Self.from.gateAS1.identifier1,

Self.channelAS1.identifier1}

endif

endlet

endif

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

RemoveAllLinks(ow:Agent) ≡

do forall l: l ∈ Link∧ l.owner = ow

RemoveLink(l)

enddo

RemoveLink(l:Link) ≡

l.program := undefined

l.owner := undefined

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

##### F3.2.3.1.8 Variable creation

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

CreateAgentVariables(sa:SdlAgent, atd:Agent-type-definition) ≡

extend StateId with sid

sa.topStateId := sid

if sa.stateAgent = sa then

sa.state := initAgentState(undefined, sid, undefined, atd.collectAllVariableDefinitions)

else

sa.stateAgent.state := initAgentState(sa.stateAgent.state,

sid, sa.owner.owner.topStateId, atd.collectAllVariableDefinitions)

endif

endextend

where

collectAllVariableDefinitions(atd: Agent-type-definition): Variable-definition-set =def

if atd.s-Agent-type-identifier = undefined then

atd.s-Variable-definition-set

else

let typedef: Agent-type-definition = atd.s-Agent-type-identifier.idToNodeAS1 in

atd.s-Variable-definition-set ∪

typedef.collectAllVariableDefinitions

endlet

endif

endwhere

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

CreateCompositeStateVariables(sa:SdlAgent, sn:StateNode,

cstd:Composite-state-type-definition) ≡

extend StateId with sid

sn.stateId := sid

sa.stateAgent.state := initAgentState(sa.stateAgent.state, sid,

if sn.parentStateNode ≠ undefined then sn.parentStateNode.stateId else undefined endif,

cstd.collectAllVariableDefinitions1)

endextend

where

collectAllVariableDefinitions1(cstd: Composite-state-type-definition):

Variable-definition-set =def

if cstd.s-Composite-state-type-identifier = undefined then

cstd.s-Variable-definition-set

else

let typedef: Composite-state-type-definition =

cstd.s-Composite-state-type-identifier.idToNodeAS1 in

cstd.s-Variable-definition-set ∪

typedef.collectAllVariableDefinitions1

endlet

endif

endwhere

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

CreateProcedureVariables(sa:SdlAgent, sn:StateNode, pd:Procedure-definition) ≡

extend StateId with sid

sn.stateId := sid

let outParams: Out-parameter\* = < p in pd.collectAllProcedureFPars:

(p ∈ Out-parameter)> in

sa.stateAgent.state := initProcedureState(sa.stateAgent.state, sid,

sn.parentStateNode.stateId, pd.collectAllVariableDefinitions2,

pd.collectAllProcedureFPars, empty,

< p.s-Parameter.identifier1 | p in outParams>)

endlet

endextend

where

collectAllVariableDefinitions2(pd: Procedure-definition): Variable-definition-set =def

if pd.s-Procedure-identifier = undefined then

pd.s-Variable-definition-set

else

let procdef: Procedure-definition = pd.s-Procedure-identifier.idToNodeAS1 in

pd.s-Variable-definition-set ∪

procdef.collectAllVariableDefinitions2

endlet

endif

collectAllProcedureFPars(pd:Procedure-definition): Procedure-formal-parameter\* =def

if pd.s-Procedure-identifier = undefined then

pd.s-Procedure-formal-parameter-seq

else

let procdef: Procedure-definition = pd.s-Procedure-identifier.idToNodeAS1 in

procdef.collectAllProcedureFPars ⁀

pd.s-Procedure-formal-parameter-seq

endlet

endif

endwhere

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

CreateCompoundNodeVariables(sa:SdlAgent, scope: Scope) ≡

extend StateId with sid

sa.currentStateId := sid

scopeName(Self, sid) := scope.s-Connector-name

scopeContinueLabel(Self, sid) := scope.s-ContinueLabel

scopeStepLabel(Self, sid) := scope.s-StepLabel

sa.stateAgent.state := initAgentState(sa.stateAgent.state, sid,

sa.currentStateId, scope.s-Variable-definition-set)

endextend

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

##### F3.2.3.1.9 State machine creation and initialization

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

CreateStateMachine(smd:[State-machine]) ≡

CreateTopStatePartition(smd)

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

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

InitStateMachine ≡

if Self.stateNodesToBeCreated ≠ ∅ then

CreateStateNode

elseif Self.statePartitionsToBeCreated ≠ ∅ then

CreateStatePartition

elseif Self.stateNodesToBeSpecialised ≠ ∅ then // these are composite states!

CreateInheritedState

elseif Self.stateNodesToBeRefined ≠ ∅ then

CreateStateRefinement

else

Self.agentMode2 := initialisationFinished

endif

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

##### F3.2.3.1.10 Procedure graph creation and initialization

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

CreateProcedureGraph(pd:Procedure-definition, vl:[ValueLabel], cl:ContinueLabel) ≡

CreateProcedureStateNode(pd, vl, cl)

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

InitProcedureGraph ≡

if Self.stateNodesToBeCreated ≠ ∅ then

CreateStateNode

elseif Self.statePartitionsToBeCreated ≠ ∅ then

CreateStatePartition

elseif Self.stateNodesToBeSpecialised ≠ ∅ then // these are composite states!

CreateInheritedState

elseif Self.stateNodesToBeRefined ≠ ∅ then

CreateStateRefinement

else

Self.agentMode4 := initialisationFinished

endif

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

##### F3.2.3.1.11 State node creation

The creation of state nodes is modelled by extending the controlled domain StateNode. A macro is defined to handle the creation of state nodes. State partitions are also modelled as elements of the domain StateNode, but are not treated in this clause.

CreateStateNode ≡

choose snd: snd ∈ Self.stateNodesToBeCreated

Self.stateNodesToBeCreated := Self.stateNodesToBeCreated \ {snd}

extend StateNode with sn

sn.stateAS1 := snd // used, e.g., as argument for startLabel

sn.owner := Self

sn.parentStateNode := Self.currentParentStateNode

sn.stateNodeKind := stateNode

sn.stateName := snd.s-State-name

sn.stateTransitions := snd.getStateTransitions

sn.startTransitions := ∅ // updated if the state node is refined

if snd.s-Composite-state-type-identifier ≠ undefined then

Self.stateNodesToBeRefined := Self.stateNodesToBeRefined ∪ {sn}

Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised ∪ {sn}

let parent: Composite-state-type-definition =

snd.s-Composite-state-type-identifier.idToNodeAS1 in

sn.stateDefinitionAS1 := parent

endlet

endif

endextend

endchoose

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

CreateProcedureStateNode(pd:Procedure-definition, vl:[ValueLabel], cl:ContinueLabel) ≡

extend StateNode with sn

sn.procedureAS1 := pd

sn.owner := Self

sn.parentStateNode := Self.currentParentStateNode

sn.stateNodeKind := procedureNode

sn.stateName := undefined

sn.stateTransitions := ∅

sn.startTransitions := ∅ // updated if the state node is refined

sn.resultLabel := vl

Self.stateNodesToBeRefined := {sn}

Self.stateNodesToBeCreated := ∅

Self.statePartitionsToBeCreated := ∅

Self.stateNodesToBeSpecialised := {sn}

Self.currentProcedureStateNode := sn

Self.callingProcedureNode := sn

CreateProcedureVariables(Self,sn,pd)

SaveProcedureControlBlock(sn,cl)

endextend

Procedure state nodes are the top-level nodes of a procedure graph, which is unfolded node by node subsequently. These nodes are created dynamically, when a procedure call is made. Thus, recursive procedure calls can be handled in a uniform way.

##### F3.2.3.1.12 State partition creation

The creation of state partitions is modelled by extending the controlled domain StateNode. Several macros are defined to handle the creation of various kinds of state partitions, namely the top state partition, (regular) state partitions, and state partitions introduced to model inheritance.

CreateTopStatePartition(smd:[State-machine]) ≡

extend StateNode with sn

sn.owner := Self

Self.topStateNode := sn

sn.parentStateNode := undefined

sn.stateNodeKind := statePartition

sn.stateTransitions := ∅

sn.startTransitions := ∅ // updated if the state partition is refined

if smd ≠ undefined then

sn.stateDefinitionAS1 := smd.s-Composite-state-type-identifier.idToNodeAS1

sn.stateName := smd.s-State-name

Self.stateNodesToBeRefined := {sn}

Self.stateNodesToBeSpecialised := {sn}

else

sn.stateName := undefined

Self.stateNodesToBeRefined := ∅

Self.stateNodesToBeSpecialised := ∅

endif

Self.stateNodesToBeCreated := ∅

Self.statePartitionsToBeCreated := ∅

endextend

The unfolding of the "hierarchical inheritance state graph" modelling an agent's state machine starts with the creation of the root node, as defined by the macro CreateTopStatePartition. When a root node is created, its bookkeeping information is initialized. In particular, the root node is classified as a state partition. If the agent has behaviour, the root node has a substructure, and is therefore included in the set of state nodes to be refined. Further state components of the agent are reset before starting the unfolding of the graph.

CreateStatePartition ≡

choose spd: spd ∈ Self.statePartitionsToBeCreated

Self.statePartitionsToBeCreated := Self.statePartitionsToBeCreated \ {spd}

extend StateNode with sn

sn.partitionAS1 := spd // used, e.g., as argument for startLabel

sn.owner := Self

sn.parentStateNode := Self.currentParentStateNode

sn.stateNodeKind := statePartition

sn.stateName := spd.s-Name

sn.stateTransitions := ∅

sn.startTransitions := ∅ // updated if the state partition is refined

do forall cd: cd ∈ spd.s-Connection-definition-set

if cd ∈ Entry-connection-definition then

entryConnection(cd.s-Outer-entry-point.adaptEntryPoint, sn) :=

adaptEntryPoint(cd.s-Inner-entry-point)

elseif cd ∈ Exit-connection-definition then

exitConnection(cd.s-Inner-exit-point, sn) := cd.s-Outer-exit-point

endif

enddo

Self.currentParentStateNode.statePartitionSet :=

Self.currentParentStateNode.statePartitionSet ∪ {sn}

Self.stateNodesToBeRefined := Self.stateNodesToBeRefined ∪ {sn}

Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised ∪ {sn}

endextend

endchoose

where

adaptEntryPoint(entry: Name ∪ DEFAULT): StateEntryPoint =def

if entry = DEFAULT then undefined else entry endif

endwhere

(Regular) state partitions are created as part of a state aggregation node, which is unfolded node by node. The partitions to be created are kept in the agent's state component statePartitionsToBeCreated. If that set is not empty, this means that the unfolding of a state aggregation node is currently in progress, and some element of the set is chosen. When a state partition is created, its bookkeeping information is initialized. Modelling a state partition, the created state node may have a substructure, and is therefore included in the set of state nodes to be refined.

CreateInheritedState ≡

choose sns: sns ∈ Self.stateNodesToBeSpecialised

Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised \ {sns}

let cstd: Composite-state-type-definition =

sns.stateDefinitionAS1 in

if cstd.s-Composite-state-type-identifier ≠ undefined then

let parent: State-node = cstd.s-Composite-state-type-identifier.idToNodeAS1 in

extend StateNode with sn

sn.stateAS1 := parent

sn.owner := Self

sn.parentStateNode := sns.parentStateNode

sn.stateNodeKind := sns.stateNodeKind

sn.stateName := sns.stateName

sn.stateTransitions := ∅

sn.startTransitions := ∅ // updated if the state node is refined

sns.inheritedStateNode := sn

Self.stateNodesToBeRefined := Self.stateNodesToBeRefined ∪ {sn}

Self.stateNodesToBeSpecialised := Self.stateNodesToBeSpecialised ∪ {sn}

endextend

endlet

else

sns.inheritedStateNode := undefined

endif

endlet

endchoose

Specialization of composite state types is modelled by adding another dimension to the hierarchical state graph, yielding a "hierarchical *inheritance* state graph". Formally, specialization is a relation between composite state types. In the state graph, it is modelled by an inheritance relation among state node instances. More specifically, if a state node is refined, and the refinement is defined using specialization, then a root node that is inherited by the refined state node, and has the composite state type being specialized, is created. By adding the root node to the set of state nodes to be refined, a "hierarchical inheritance state graph" modelling the specialization is subsequently attached to this root node.

##### F3.2.3.1.13 Composite state creation

All (regular) state nodes, state partitions, and procedure nodes are candidates for refinement and, if refined, for specialization. Refinements are defined by a composite state type, which includes another composite state type in case of specialization. In this clause, several macros treating these aspects are introduced.

CreateStateRefinement ≡

choose snr: snr ∈ Self.stateNodesToBeRefined

Self.stateNodesToBeRefined := Self.stateNodesToBeRefined \ {snr}

Self.currentParentStateNode := snr

if snr.stateNodeKind = procedureNode then

CreateProcedureVariables(Self, snr, snr.procedureAS1)

CreateProcedureGraphNodes(snr, snr.procedureAS1.s-Procedure-graph)

else

let parent: Composite-state-type-definition = snr.stateDefinitionAS1 in

CreateCompositeStateVariables(Self, snr,

parent)

CreateCompositeState(snr,

parent)

endlet

endif

endchoose

When a state node, state partition, or procedure node is created, it is added to a set of state nodes to be refined. In the macro CreateStateRefinement, an arbitrary element of this set is selected, and it is checked whether a refinement applies. Refinements are then treated by the macro CreateCompositeState.

CreateCompositeState(sn:StateNode, cstd:Composite-state-type-definition) ≡

let sr = cstd.s-implicit in

if sr ∈ Composite-state-graph then

CreateCompositeStateGraph(sn,sr)

elseif sr ∈ State-aggregation-node then

CreateStateAggregationNode(sn,sr)

endif

endlet

If a state is structured, it is refined into either a composite state graph or a state aggregation node. Based on this distinction, further rule macros are applied.

CreateCompositeStateGraph(psn:StateNode, csgd:Composite-state-graph) ≡

psn.stateNodeRefinement := compositeStateGraph

psn.startTransitions := getStartTransitions({csgd.s-State-transition-graph.s-State-start-node}) ∪

getStartTransitions(csgd.s-Named-start-node-set)

psn.freeActions := getFreeActions(csgd.s-State-transition-graph.s-Free-action-set)

CreateStateTransitionGraph(psn,csgd.s-State-transition-graph.s-State-node-set)

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

CreateStateTransitionGraph(psn:StateNode, nodes: State-node-set ) ≡

Self.stateNodesToBeCreated := nodes

Self.currentParentStateNode := psn

Creating a state transition graph means creating its state nodes. Creation of state nodes is performed in a series of subsequent ASM steps. These steps are triggered by assigning the state node definitions to the agent's state component stateNodesToBeCreated.

CreateProcedureGraphNodes(psn:StateNode, pg:Procedure-graph) ≡

psn.stateNodeRefinement := compositeStateGraph

psn.startTransitions := getStartTransitions({pg.s-Procedure-start-node})

psn.freeActions := getFreeActions(pg.s-Free-action-set)

CreateStateTransitionGraph(psn, pg.s-State-node-set)

Self.stateNodesToBeCreated := pg.s-State-node-set

Self.currentParentStateNode := psn

Creating a procedure graph means creating its state nodes.

CreateStateAggregationNode(psn:StateNode, sand:State-aggregation-node) ≡

psn.stateNodeRefinement := stateAggregationNode

Self.statePartitionsToBeCreated := sand.s-State-partition-seq.toSet

Self.currentParentStateNode := psn

psn.statePartitionSet := ∅

Creating a state aggregation node means creating its state partitions, which is performed in a series of subsequent ASM steps. These steps are triggered by assigning the state partition definitions to the agent's state component statePartitionsToBeCreated.

#### F3.2.3.2 System execution

After initialization, SDL‑2010 agents start their execution. The execution of the system is modelled by the concurrent execution of all its agents.

##### F3.2.3.2.1 Agent set execution

ExecAgentSet ≡

let child = take({ag ∈ SdlAgent: ag.owner = Self ∧ ag.agentMode1 = initialisation}) in

if child = undefined then

DeliverSignals

endif

endlet

The behaviour of agent sets is formalized below.

DeliverSignals ≡

choose g: g ∈ Self.ingates ∧ g.queue ≠ empty

let si = g.queue.head in

Delete(si,g)

if si.toArg ∈ Pid ∧ si.toArg ≠ undefined then

choose sa: sa ∈ SdlAgent ∧ sa.owner = Self ∧ sa.selfPid = si.toArg

Insert(si, si.arrival, sa.inport)

endchoose

else

choose sa: sa ∈ SdlAgent ∧ sa.owner = Self

Insert(si, si.arrival, sa.inport)

endchoose

endif

endlet

endchoose

##### F3.2.3.2.2 Agent execution

The execution of SDL‑2010 agents is modelled by a start phase followed by alternating phases, namely transition selection and transition firing. To distinguish between these phases, corresponding agent modes are defined. When in agent mode selectingTransition (agentMode2), the agent attempts to select a transition, obeying a number of constraints. In agent mode firingTransition, a previously selected transition is fired.

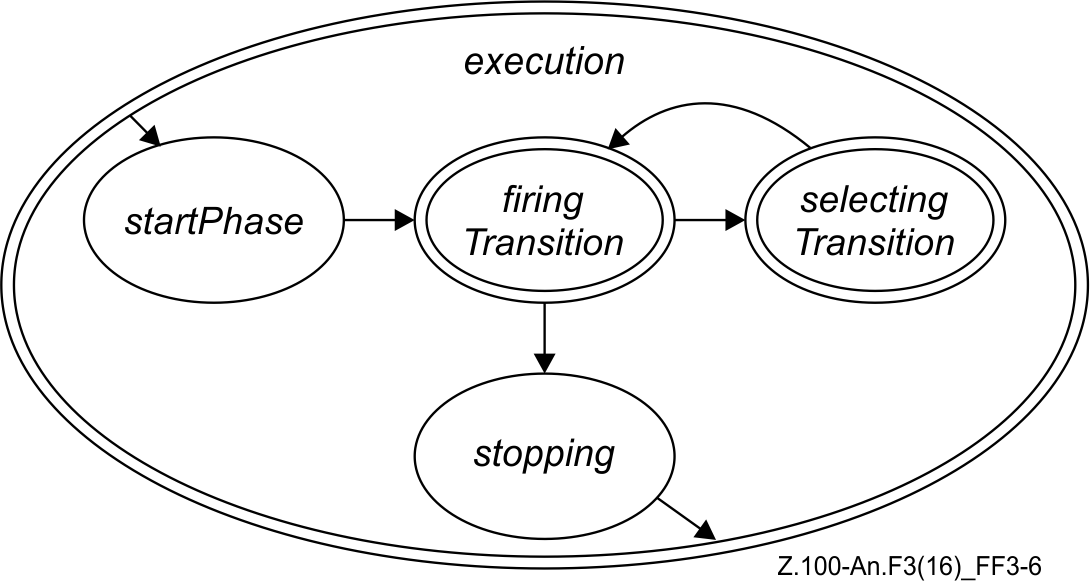


Figure F3-6 – Activity phases of SDL‑2010 agents: execution (level 2)

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

ExecAgent ≡

if Self.agentMode2 = startPhase then

ExecutionStartPhase

elseif Self.agentMode2 = firingTransition then

FireTransition

elseif Self.agentMode2 = selectingTransition then

SelectTransition

elseif Self.agentMode2 = stopping then

StopPhase

endif

The execution of agents is given by the rule macro ExecAgent. Depending on the current agent mode, the corresponding execution phases are selected.

GetExecRight ≡

if Self.stateAgent.isActive = undefined then

Self.stateAgent.isActive := Self

endif

ReturnExecRight ≡

Self.stateAgent.isActive := undefined

ExecRightPresent(sa:SdlAgent): Boolean =def

let myDef: Agent-type-definition = sa.owner.agentAS1.s-Agent-type-identifier.idToNodeAS1 in

sa.stateAgent.isActive = sa ∨ myDef.s-Agent-kind ∈ {BLOCK, SYSTEM}

endlet

##### F3.2.3.2.3 Starting agent execution

When the execution phase starts, several initializations are made: the set of state nodes to be entered is initialized to consist of the top state node; furthermore, the execution is switched to entering state nodes.

ExecutionStartPhase ≡

Self.isActive := undefined

Self.stateNodesToBeEntered :=

{mk**-**StateNodeWithEntryPoint (Self.topStateNode,undefined)}

Self.agentMode2 := firingTransition

Self.agentMode3 := enteringStateNode

Self.agentMode4 := startPhase

Self.currentLabel := undefined

##### F3.2.3.2.4 Transition selection

In agent mode selectingTransition (agentMode2), an SDL‑2010 agent searches for a fireable transition. SDL‑2010 imposes certain rules on the search order. For instance, priority input signals have to be checked before ordinary input signals, and these have in turn to be checked before continuous signals can be consumed. Furthermore, a transition emanating from a substate has higher priority than a conflicting transition emanating from any of the containing states. Finally, redefined transitions take precedence over conflicting inherited transitions. These and some more constraints have to be observed when formalizing the transition selection.

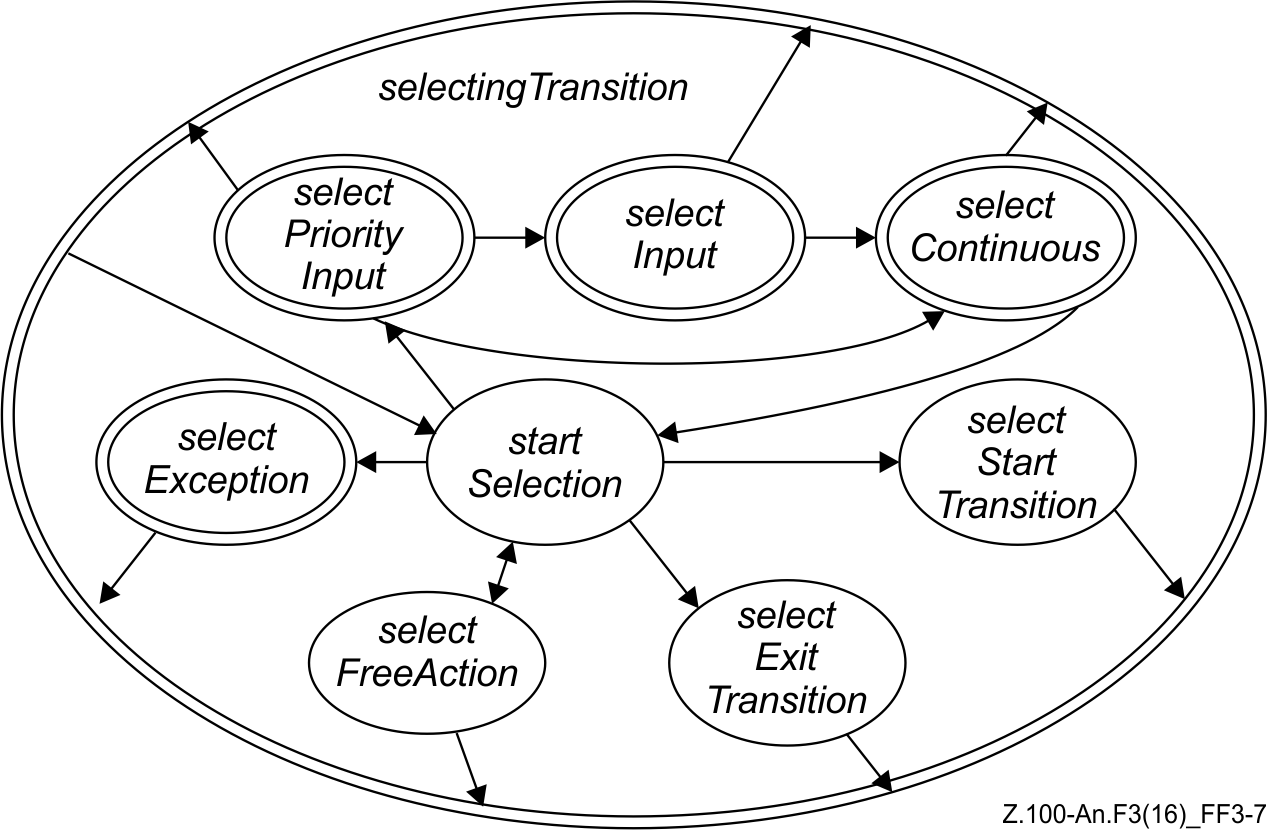


Figure F3-7 – Activity phases of SDL‑2010 agents: selecting transition (level 3)

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

SelectTransition ≡

if Self.agentMode3 = startSelection then

SelectTransitionStartPhase

elseif Self.agentMode3 = selectStartTransition then

SelectStartTransition

elseif Self.agentMode3 = selectExitTransition then

SelectExitTransition

elseif Self.agentMode3 = selectFreeAction then

SelectFreeAction

elseif Self.agentMode3 = selectPriorityInput then

SelectPriorityInput

elseif Self.agentMode3 = selectInput then

SelectInput

elseif Self.agentMode3 = selectContinuous then

SelectContinuous

endif

Transition selection starts with an attempt to select a start transition, free action, priority input, an ordinary input, and finally, a continuous signal (in that order). If no transition has been selected, the selection process is repeated/aborted. The evaluation of provided expressions and continuous expressions may alter the local state of the process, which may lead to different results depending on the evaluation order.

TransitionFound(t:SemTransition) ≡

Self.currentParentStateNode := Self.stateNodeChecked.parentStateNode

Self.previousStateNode := Self.stateNodeChecked

Self.currentStateId := Self.stateNodeChecked.parentStateNode.stateId

Self.currentLabel := t.s2-Label // second label

Self.agentMode2 := firingTransition

Self.agentMode3 := firingAction

ReturnExecRight

As soon as a selectable transition is found, the start label of the transition is assigned, and the agent modes are set to firingTransition and firingAction, respectively. Also, the current parent state node is set, which determines the current state name scope. This scope information is used when an EnterStateNode-primitive is evaluated.

StartTransitionFound(t:StartTransition, psn:StateNode) ≡

Self.currentParentStateNode := psn

Self.currentStateId := psn.stateId

Self.currentLabel := t.s-Label

Self.agentMode2 := firingTransition

Self.agentMode3 := firingAction

ReturnExecRight

As soon as a selectable start transition is found, the start label of the transition is assigned, and the agent modes are set to firingTransition and firingAction, respectively. Also, the current parent state node is set, which determines the current state name scope. This scope information is used when an EnterStateNode-primitive is evaluated.

ExitTransitionFound(et:SemTransition, psn:StateNode) ≡

Self.currentParentStateNode := psn

Self.currentStateId := psn.stateId

Self.currentLabel := et.s-Label

Self.agentMode2 := firingTransition

Self.agentMode3 := firingAction

ReturnExecRight

As soon as a selectable exit transition is found, the start label of the transition is assigned, and the agent modes are set to firingTransition and firingAction, respectively. Also, the current parent state node is set, which determines the current state name scope. This scope information is used when a LeaveStateNode-primitive is evaluated.

FreeActionFound(fa:FreeAction, psn:StateNode) ≡

Self.currentParentStateNode := psn

Self.currentStateId := psn.stateId

Self.currentLabel := fa.s-Label

Self.agentMode2 := firingTransition

Self.agentMode3 := firingAction

ReturnExecRight

As soon as a free action is found, the start label of the transition is assigned, and the agent modes are set to firingTransition and firingAction, respectively. Also, the current parent state node is set, which determines the current state name scope.

##### F3.2.3.2.5 Starting selection of transitions

When the selection of transition starts, several initializations are made: the input port is "frozen", meaning that its state at the beginning of the selection is the basis for this selection cycle. This does not prevent signal instances to arrive while the selection is active; however, these signals will not be considered before the next selection cycle. Furthermore, the selection is switched to checking priority signals.

SelectTransitionStartPhase ≡

if Self.currentStartNodes ≠ ∅ then

Self.stateNodeChecked := undefined

Self.agentMode3 := selectStartTransition

elseif Self.currentExitStateNodes ≠ ∅ then

Self.stateNodeChecked := undefined

Self.agentMode3 := selectExitTransition

elseif Self.currentConnector ≠ undefined then

Self.agentMode3 := selectFreeAction

else

Self.inputPortChecked := Self.inport.queue

Self.agentMode3 := selectPriorityInput

Self.agentMode4 := startPhase

endif

##### F3.2.3.2.6 Start transition selection

Selection of a start transition is performed by checking, for all current start nodes, whether a start transition can be selected.

SelectStartTransition ≡

if Self.stateNodeChecked = undefined then

let snwen = take(Self.currentStartNodes) in

if snwen ≠ undefined then

Self.currentStartNodes := Self.currentStartNodes \ {snwen}

Self.startNodeChecked := snwen

Self.stateNodeChecked := snwen.s-StateNode

endif

endlet

else

let t = take({tr ∈ Self.stateNodeChecked.startTransitions:

tr.s-StateEntryPoint = Self.startNodeChecked.s-implicit}) in

if t ≠ undefined then

StartTransitionFound(t, Self.startNodeChecked.s-StateNode)

else

Self.stateNodeChecked :=

take({sn1 ∈ Self.stateNodesToBeChecked:

directlyInheritsFrom(Self.stateNodeChecked,sn1)})

endif

endlet

endif

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

##### F3.2.3.2.7 Exit transition selection

SelectExitTransition ≡

let snwex = take(Self.currentExitStateNodes) in

if Self.stateNodeChecked = undefined then

if snwex ≠ undefined then

Self.currentExitStateNodes := Self.currentExitStateNodes \ {snwex}

Self.exitNodeChecked := snwex

Self.stateNodeChecked := snwex.s-StateNode

endif

else

let t = take({tr ∈ Self.stateNodeChecked.stateTransitions.exitTransitions:

tr.s-StateExitPoint = Self.exitNodeChecked.s-StateExitPoint}) in

if t ≠ undefined then

ExitTransitionFound(t,snwex.s-StateNode)

else

Self.stateNodeChecked :=

take({sn1 ∈ Self.stateNodesToBeChecked:

directlyInheritsFrom(Self.stateNodeChecked,sn1)})

endif

endlet

endif

endlet

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

##### F3.2.3.2.8 Free action selection

SelectFreeAction ≡

let fa = take({elem ∈ Self.stateNodeChecked.freeActions:

elem.s-Connector-name = Self.currentConnector.s-Connector-name}) in

if fa ≠ undefined then

Self.currentConnector := undefined

FreeActionFound(fa, Self.currentParentStateNode)

else

Self.stateNodeChecked :=

take({sn1 ∈ Self.stateNodesToBeChecked:

directlyInheritsFrom(Self.stateNodeChecked,sn1)})

endif

endlet

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

##### F3.2.3.2.9 Priority input selection

Selection of a priority input is performed by checking, for each signal instance of the agent's input port, all current state nodes. Inheritance is taken into account by checking, for each state node, the inherited state nodes.

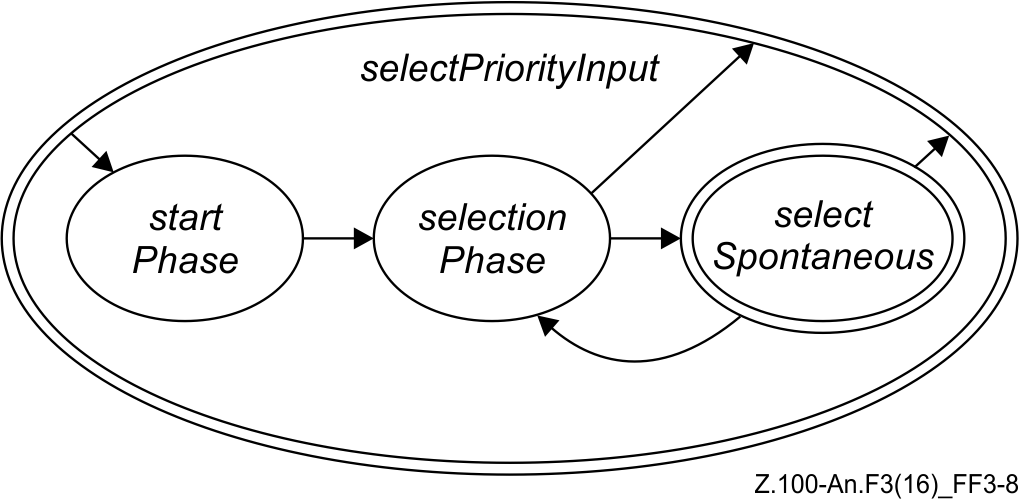


Figure F3-8 – Activity phases of SDL‑2010 agents: selecting priority inputs (level 4)

The selection of a priority input consists of the sub-phases (agentMode4) shown in the diagram. At any time during the selection phase, an attempt to select a spontaneous signal may be made, depending on the value of the monitored predicate Self.spontaneous.

SelectPriorityInput ≡

if Self.agentMode4 = startPhase then

SelPriorityInputStartPhase

elseif Self.agentMode4 = selectionPhase then

SelPriorityInputSelectionPhase

elseif Self.agentMode4 = selectSpontaneous then

SelectSpontaneous

endif

This ASM macro defines the upper level control structure of the priority input selection. Depending on the agent mode agentMode4, further action is defined in the corresponding ASM macro. This control structure is part of the previous state diagram.

SelPriorityInputStartPhase ≡

if Self.inputPortChecked ≠ empty then

Self.signalChecked := Self.inputPortChecked.head

Self.SignalSaved := false

Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)

Self.stateNodeChecked := undefined

Self.agentMode4 := selectionPhase

else

Self.agentMode3 := selectContinuous

Self.agentMode4 := startPhase

ReturnExecRight

endif

When the selection starts, it is checked whether the input port carries signals. If so, several initializations are made: the first signal instance to be checked is determined, the state nodes to be checked are set, and the selection is activated. If the input port is empty, the selection of continuous signals is triggered.

SelPriorityInputSelectionPhase ≡

if Self.stateNodeChecked = undefined then

NextStateNodeToBeChecked

elseif Self.spontaneous then

Self.agentMode4 := selectSpontaneous

Self.agentMode5 := selectionPhase

else

let t = take({tr ∈ Self.stateNodeChecked.stateTransitions.priorityInputTransitions:

tr.s-Signal = Self.signalChecked.signalType}) in

if t ≠ undefined then

Self.currentSignalInst := Self.signalChecked

Self.sender := Self.signalChecked.signalSender

Delete(Self.signalChecked, Self.inport)

TransitionFound(t)

else

Self.stateNodeChecked := undefined

endif

endlet

endif

where

NextStateNodeToBeChecked ≡

if Self.stateNodesToBeChecked ≠ ∅ ∧ ¬ Self.SignalSaved then

SelectNextStateNode

else

NextSignalToBeChecked

Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)

Self.stateNodeChecked := undefined

endif

SelectNextStateNode ≡

let sn = Self.stateNodesToBeChecked.selectNextStateNode in

if sn = undefined then

UndefinedBehaviour

elseif sn.stateNodeKind = procedureNode then

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \

collectCurrentSubStates(sn.getPreviousStatePartition)

// only state partitions of the state machine to be considered here

elseif sn.stateNodeKind = statePartition then

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}

elseif sn.stateNodeKind = stateNode then

let curSigId: Identifier = Self.signalChecked.signalType in

Self.stateNodeChecked := sn

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}

Self.transitionsToBeChecked :=

{t ∈ sn.stateTransitions.inputTransitions: t.s-Signal = curSigId}

if Self.signalChecked.signalType ∈

sn.stateAS1.s-Save-signalset.s-Signal-identifier-set then

Self.SignalSaved := true

endif

endlet

endif

endlet

NextSignalToBeChecked ≡

let si = nextSignal(Self.signalChecked, Self.inputPortChecked) in

if si ≠ undefined then

Self.signalChecked := si

Self.SignalSaved := false

else

Self.agentMode3 := selectInput

Self.agentMode4 := startPhase

ReturnExecRight

endif

endlet

endwhere

For a given signal instance in the input port, all current state nodes of the agent are checked in an arbitrary order, beginning, for each state partition, with the innermost state node. The latter reflects the priority among conflicting transitions. Furthermore, when a particular state node is being checked, the inherited state nodes are checked next, i.e., inheritance is taken into account at execution time and not handled by transformations. As a redefinition takes precedence over the redefined transition, the inherited nodes are to be checked only if the current signal instance is neither saved nor consumed in the current state.

If the given signal instance is not a priority input in the current states of the agent, the next signal instance of the input port is checked. This is repeated until either all signals have been checked, or a priority input has been found. In the former case, the selection of an input transition is triggered.

##### F3.2.3.2.10 Input selection

Selection of an input is performed by checking, for each signal instance of the agent's input port, all current state nodes until a signal instance satisfying certain conditions is found. If no such signal instance is found, the selection of a continuous signal is triggered.

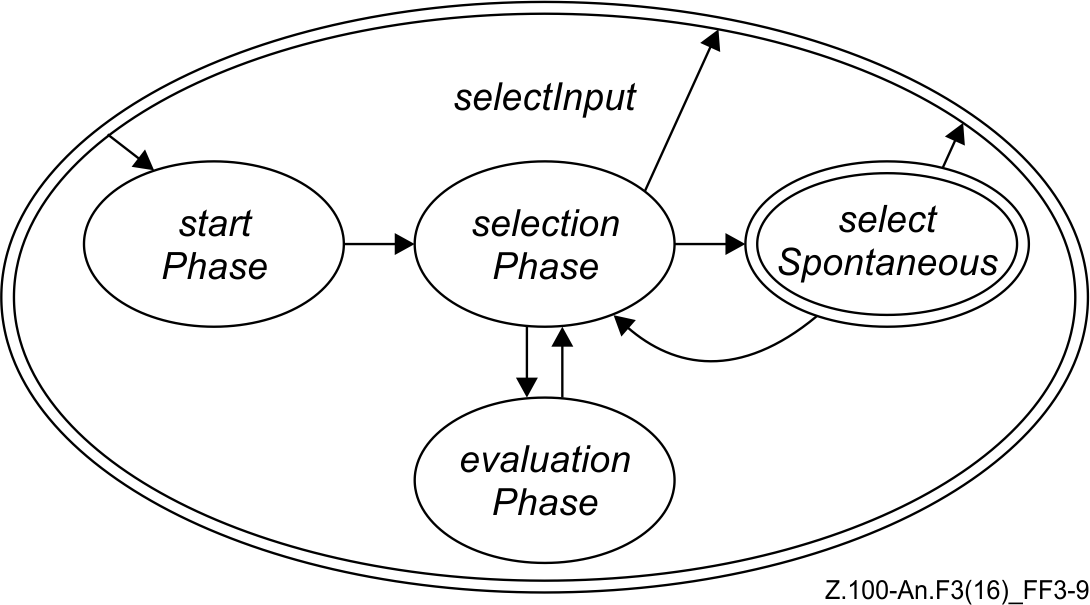


Figure F3-9 – Activity phases of SDL‑2010 agents: selecting inputs (level 4)

The selection of an ordinary input consists of the sub-phases shown in the state diagram. In comparison to the selection of a priority input, an evaluation phase is added. This phase is entered when a provided expression has to be evaluated. At any time during the selection phase, an attempt to select a spontaneous signal may be made, depending on the value of the monitored predicate Self.spontaneous.

SelectInput ≡

if Self.agentMode4 = startPhase then

SelInputStartPhase

elseif Self.agentMode4 = selectionPhase then

SelInputSelectionPhase

elseif Self.agentMode4 = evaluationPhase then

SelInputEvaluationPhase

elseif Self.agentMode4 = selectSpontaneous then

SelectSpontaneous

endif

This ASM macro defines the upper level control structure of the input selection. Depending on the agent mode agentMode3, further action is defined in the corresponding ASM macro. This control structure is part of the previous state diagram.

SelInputStartPhase ≡

if Self.inputPortChecked ≠ empty then

Self.signalChecked := Self.inputPortChecked.head

Self.SignalSaved := false

Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)

Self.stateNodeChecked := undefined

Self.transitionsToBeChecked := ∅

Self.agentMode4 := selectionPhase

else

Self.agentMode3 := selectContinuous

Self.agentMode4 := startPhase

ReturnExecRight

endif

When the selection starts, it is checked whether the input port contains signals. If so, several initializations are made: the first signal instance to be checked is determined, the state nodes to be checked are set, the transitions to be checked are reset, and the selection is activated. If the input port is empty, the selection of a continuous signal is triggered.

SelInputSelectionPhase ≡

if Self.stateNodeChecked = undefined then

NextStateNodeToBeChecked1

elseif Self.spontaneous then

Self.agentMode4 := selectSpontaneous

Self.agentMode5 := selectionPhase

elseif Self.transitionsToBeChecked ≠ ∅ then

choose t: t ∈ Self.transitionsToBeChecked

Self.transitionsToBeChecked := Self.transitionsToBeChecked \ {t}

if t.s-Label ≠ undefined then

EvaluateEnablingCondition(t)

else

Self.currentSignalInst := Self.signalChecked

Self.sender := Self.signalChecked.signalSender

Delete(Self.signalChecked,Self.inport)

TransitionFound(t)

endif

endchoose

else

Self.stateNodeChecked := undefined

endif

where

EvaluateEnablingCondition(t:SemTransition) ≡

Self.transitionChecked := t

Self.currentStateId := Self.stateNodeChecked.parentStateNode.stateId

Self.currentLabel := t.s-Label

Self.agentMode4 := evaluationPhase

NextStateNodeToBeChecked1 ≡

if Self.stateNodesToBeChecked ≠ ∅ ∧ ¬ Self.SignalSaved then

SelectNextStateNode1

else

if ¬ Self.SignalSaved then // implicit transition

Delete(Self.signalChecked,Self.inport)

endif

NextSignalToBeChecked1

Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)

Self.stateNodeChecked := undefined

endif

SelectNextStateNode1 ≡

let sn = Self.stateNodesToBeChecked.selectNextStateNode in

if sn = undefined then

UndefinedBehaviour

elseif sn.stateNodeKind = procedureNode then

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \

collectCurrentSubStates(sn.getPreviousStatePartition)

// only state partitions of the state machine to be considered here

elseif sn.stateNodeKind = statePartition then

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}

elseif sn.stateNodeKind = stateNode then

Self.stateNodeChecked := sn

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}

Self.transitionsToBeChecked := {t∈ sn.stateTransitions.inputTransitions:

t.s-Signal = Self.signalChecked.signalType}

if Self.signalChecked.signalType ∈

sn.stateAS1.s-Save-signalset.s-Signal-identifier-set then

Self.SignalSaved := true

endif

endif

endlet

NextSignalToBeChecked1 ≡

let si = nextSignal(Self.signalChecked,Self.inputPortChecked) in

if si ≠ undefined then

Self.signalChecked := si

Self.SignalSaved := false

else

Self.agentMode3 := selectContinuous

Self.agentMode4 := startPhase

ReturnExecRight

endif

endlet

endwhere

For a given signal instance in the input port, all current state nodes of the agent are checked in an arbitrary order, beginning, for each state partition, with the innermost state node. The latter reflects the priority among conflicting transitions. Furthermore, when a particular state node is being checked, the inherited state nodes are checked next, i.e., inheritance is taken into account at execution time and not handled by transformations. As a redefinition takes precedence over the redefined transition, the inherited nodes are to be checked only if the current signal instance is neither saved nor consumed in the current state.

If the given signal instance is saved in the current states of the agent, the next signal instance of the input port is checked. This is repeated until either all signals have been checked, or an input has been selected. In the former case, the selection of a continuous signal is triggered.

SelInputEvaluationPhase ≡

if Self.currentLabel ≠ undefined then

choose b: b ∈ behaviour ∧ b.s-Label = Self.currentLabel

Eval(b.s-Action)

endchoose

elseif semvalueBool(value(Self.transitionChecked.s-Label,Self)) then

Self.currentSignalInst := Self.signalChecked

Self.sender := Self.signalChecked.signalSender

Delete(Self.signalChecked,Self.inport)

TransitionFound(Self.transitionChecked)

else

Self.agentMode4 := selectionPhase

endif

If an input transition has a provided expression, this expression has to be evaluated before continuing with the selection. As this evaluation consists of several actions in general, another agent mode, evaluationPhase, is entered. After completion of the evaluation, either the considered input signal is consumed, or the selection continues.

##### F3.2.3.2.11 Continuous signal selection

Selection of an input is performed by checking, for each signal instance of the agent's input port, all current state nodes until a signal instance satisfying certain conditions is found. If no such signal instance is found, this cycle of transition selection ends, and another cycle is started.

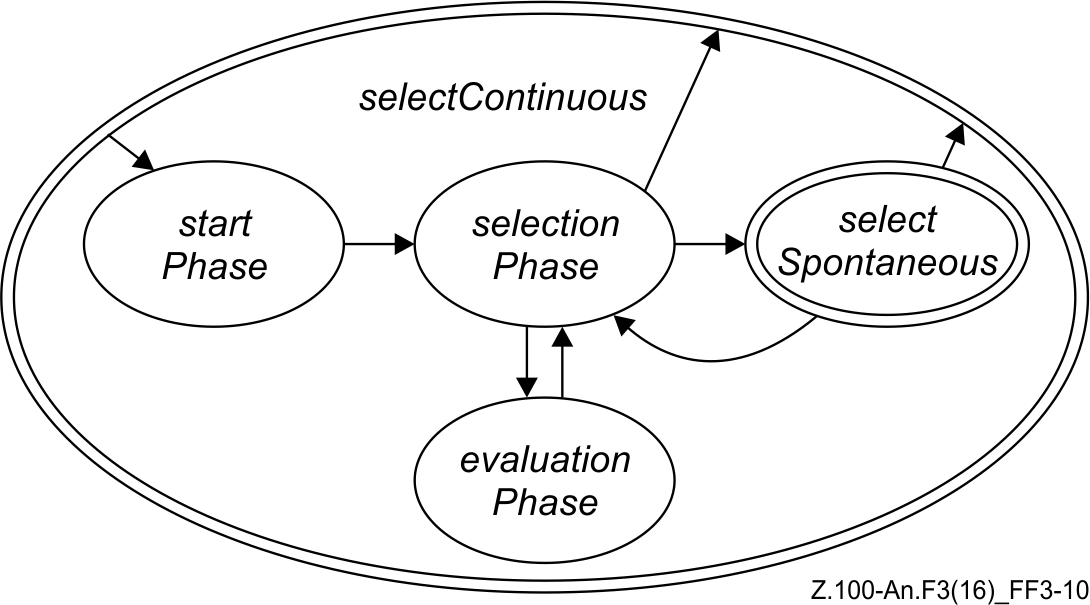


Figure F3-10 – Activity phases of SDL‑2010 agents: selecting continuous signals (level 4)

The selection of a continuous signal consists of the sub-phases shown in the state diagram. The control is identical to the selection of an ordinary input.

SelectContinuous ≡

if Self.agentMode4 = startPhase then

SelContinuousStartPhase

elseif Self.agentMode4 = selectionPhase then

SelContinuousSelectionPhase

elseif Self.agentMode4 = evaluationPhase then

SelContinuousEvaluationPhase

elseif Self.agentMode4 = selectSpontaneous then

SelectSpontaneous

endif

This ASM macro defines the upper level control structure of the continuous signal selection. Depending on the agent mode agentMode4, further action is defined in the corresponding ASM macro. This control structure is part of the previous state diagram.

SelContinuousStartPhase ≡

Self.stateNodesToBeChecked := collectCurrentSubStates(Self.topStateNode)

Self.stateNodeChecked := undefined

Self.transitionsToBeChecked := ∅

Self.agentMode4 := selectionPhase

When the selection starts, several initializations are made: the state nodes to be checked are set, the transitions to be checked are reset, and the selection is activated.

SelContinuousSelectionPhase ≡

if Self.stateNodeChecked = undefined then

NextStateNodeToBeChecked2

elseif Self.spontaneous then

Self.agentMode4 := selectSpontaneous

Self.agentMode5 := selectionPhase

else

let t = selectContinuousSignal(Self.transitionsToBeChecked, Self.continuousPriorities) in

if t ≠ undefined then

Self.transitionsToBeChecked := Self.transitionsToBeChecked \ {t}

if t.s-Label ≠ undefined then

EvaluateEnablingCondition1(t)

else

TransitionFound(t)

endif

else

NextStateNodeToBeChecked2

endif

endlet

endif

where

EvaluateEnablingCondition1(t:SemTransition) ≡

Self.transitionChecked := t

Self.currentStateId := Self.stateNodeChecked.parentStateNode.stateId

Self.currentLabel := t.s-Label

Self.agentMode4 := evaluationPhase

NextStateNodeToBeChecked2 ≡

if Self.stateNodesToBeChecked ≠ ∅ then

if Self.stateNodeChecked = undefined then

SelectNextStateNode2

else

CheckForInheritedStateNodes

endif

else

Self.agentMode3 := startSelection

ReturnExecRight

endif

SelectNextStateNode2 ≡

let sn = Self.stateNodesToBeChecked.selectNextStateNode in

if sn = undefined then

UndefinedBehaviour

elseif sn.stateNodeKind = procedureNode then

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \

collectCurrentSubStates(sn.getPreviousStatePartition)

// only state partitions of the state machine to be considered here

elseif sn.stateNodeKind = statePartition then

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}

elseif sn.stateNodeKind = stateNode then

Self.stateNodeChecked := sn

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn}

Self.transitionsToBeChecked := sn.stateTransitions.continuousSignalTransitions

Self.continuousPriorities := ∅

endif

endlet

CheckForInheritedStateNodes ≡

let sn = Self.stateNodeChecked in

let sn1 = selectInheritedStateNode(sn, Self.stateNodesToBeChecked) in

if sn1 ≠ undefined then

Self.stateNodesToBeChecked := Self.stateNodesToBeChecked \ {sn1}

Self.stateNodeChecked := sn1

Self.transitionsToBeChecked :=

sn1.stateTransitions.continuousSignalTransitions

Self.continuousPriorities := Self.continuousPriorities ∪

{ t.s-Nat | t ∈ sn.stateTransitions.continuousSignalTransitions}

else

Self.stateNodeChecked := undefined

endif

endlet

endlet

endwhere

All current state nodes of the agent are checked in an arbitrary order, beginning, for each state partition, with the innermost state node. The latter reflects the priority among conflicting transitions. Furthermore, when a particular state node is being checked, the inherited state nodes are checked. Finally, redefined transitions take precedence over conflicting inherited transitions also in case of continuous signals. If no continuous signal is found, another cycle of the transition selection is started.

SelContinuousEvaluationPhase ≡

if Self.currentLabel ≠ undefined then

choose b: b ∈ behaviour ∧ b.s-Label = Self.currentLabel

Eval(b.s-Action)

endchoose

elseif semvalueBool(value(Self.transitionChecked.s-Label,Self)) then

TransitionFound(Self.transitionChecked)

else

Self.agentMode4 := selectionPhase

endif

For each continuous signal, the continuous expression has to be evaluated. As this evaluation consists of several actions in general, another agent mode, evaluationPhase, is entered. After completion of the evaluation, either the considered continuous signal is consumed, or the selection continues.

##### F3.2.3.2.12 Spontaneous transition selection

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

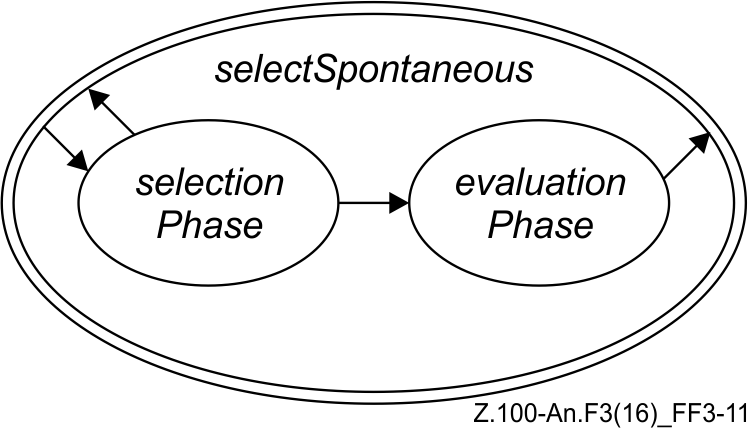


Figure F3-11 – Activity phases of SDL‑2010 agents: selecting spontaneous transitions (level 5)

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

SelectSpontaneous ≡

if Self.agentMode5 = selectionPhase then

SelSpontaneousSelectionPhase

elseif Self.agentMode5 = evaluationPhase then

SelSpontaneousEvaluationPhase

endif

This ASM macro defines the upper level control structure of the spontaneous transition selection. Depending on the agent modeagentMode5, further action is defined in the corresponding ASM macro. This control structure is part of the previous state diagram.

SelSpontaneousSelectionPhase ≡

if Self.stateNodeChecked.stateTransitions.spontaneousTransitions ≠ ∅ then

choose t: t ∈ Self.stateNodeChecked.stateTransitions.spontaneousTransitions

if t.s-Label ≠ undefined then

EvaluateEnablingCondition2(t)

else

Self.sender := Self.selfPid

TransitionFound(t)

endif

endchoose

else

Self.agentMode4 := selectionPhase

endif

where

EvaluateEnablingCondition2(t:SemTransition) ≡

Self.transitionChecked := t

Self.currentStateId := Self.stateNodeChecked.parentStateNode.stateId

Self.currentLabel := t.s-Label

Self.agentMode5 := evaluationPhase

endwhere

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

SelSpontaneousEvaluationPhase ≡

if Self.currentLabel ≠ undefined then

choose b: b ∈ behaviour ∧ b.s-Label = Self.currentLabel

Eval(b.s-Action)

endchoose

elseif semvalueBool(value(Self.transitionChecked.s-Label,Self)) then

Self.sender := Self.selfPid

TransitionFound(Self.transitionChecked)

else

Self.agentMode4 := selectionPhase

endif

If a spontaneous transition has a provided expression, this expression has to be evaluated before continuing with the selection. As this evaluation consists of several actions in general, another agent mode, evaluationPhase, is entered. After completion of the evaluation, either the considered spontaneous transition is selected, or the selection of priority input, input or continuous signals is resumed.

##### F3.2.3.2.13 Transition firing

The firing of a transition is decomposed into the firing of individual actions, which may in turn consist of a sequence of steps. At the beginning of a transition, the current state node is left; at the end, either a state node is entered, or a termination takes place.

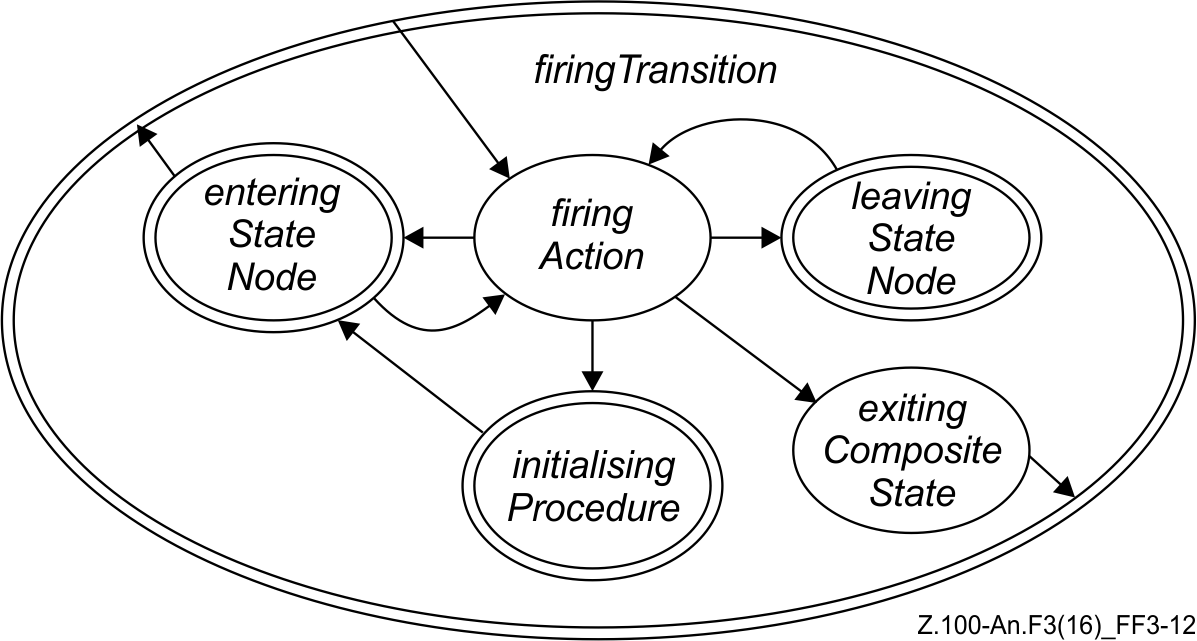


Figure F3-12 – Activity phases of SDL‑2010 agents: firing transitions (level 3)

FireTransition ≡

if Self.agentMode3 = firingAction then

FireAction

elseif Self.agentMode3 = leavingStateNode then

LeaveStateNodes

elseif Self.agentMode3 = enteringStateNode then

EnterStateNodes

elseif Self.agentMode3 = exitingCompositeState then

ExitCompositeState

elseif Self.agentMode3 = initialisingProcedure then

InitProcedure

endif

Firing of a transition consists of firing a sequence of actions. Once started, transitions are completely executed.

##### F3.2.3.2.14 Firing of actions

FireAction ≡

if Self.currentLabel ≠ undefined then

choose b: b ∈ behaviour ∧ b.s-Label = Self.currentLabel

Eval(b.s-Action)

endchoose

else

Self.agentMode2 := selectingTransition

Self.agentMode3 := startSelection

ReturnExecRight

endif

Firing of actions is defined by the selection and evaluation of the corresponding SAM primitives. Once started, the firing of actions continues until either a transition is completed (i.e., the current label has the value undefined) or until the agent mode is changed during the evaluation of a primitive. This is, for instance, the case when a state node is entered. The function currentLabel uniquely identifies a behaviour primitive.

##### F3.2.3.2.15 Entering of state nodes

EnterStateNodes ≡

if Self.agentMode4 = startPhase then

EnterStateNodesStartPhase

elseif Self.agentMode4 = enterPhase then

EnterStateNodesEnterPhase

elseif Self.agentMode4 = enteringFinished then

EnterStateNodesEnteringFinished

endif

State nodes are entered when the execution of an agent starts, and possibly when a next state action is executed. When this phase is started, a single state node with an entry point has already been selected. Depending on the structure of the hierarchical graph, further state nodes to be entered may be encountered when this single state node is entered.

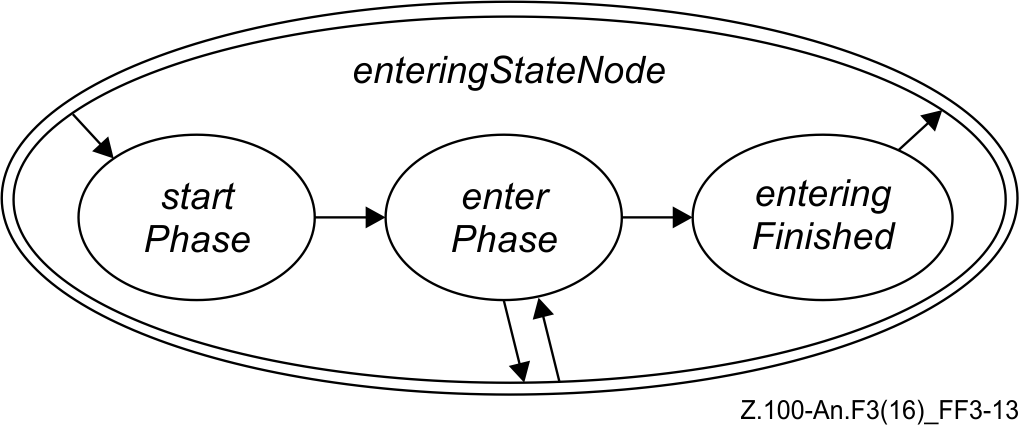


Figure F3-13 – Activity phases of SDL‑2010 agents: entering state node (level 4)

EnterStateNodesStartPhase ≡

Self.agentMode4 := enterPhase

At the beginning of this phase, the set of entered state nodes is initialized. This set is updated every time another state node is entered, and evaluated at the end of the phase to determine the set of current state nodes of the agent.

EnterStateNodesEnterPhase ≡

if Self.stateNodesToBeEntered ≠ ∅ then

choose snwen: snwen ∈ Self.stateNodesToBeEntered

snwen.s-StateNode.currentSubStates := ∅

snwen.s-StateNode.currentExitPoints := ∅

snwen.s-StateNode.previousSubStates := ∅

if snwen.s-StateNode.parentStateNode ≠ undefined then

snwen.s-StateNode.parentStateNode.currentSubStates :=

snwen.s-StateNode.parentStateNode.currentSubStates ∪ {snwen.s-StateNode}

endif

if snwen.s-StateNode.stateNodeRefinement = undefined then

RefinementUndef(snwen)

elseif snwen.s-StateNode.stateNodeRefinement = stateAggregationNode then

RefinementStateAggrNode(snwen)

elseif snwen.s-StateNode.stateNodeRefinement = compositeStateGraph then

RefinementCompStateNode(snwen)

endif

endchoose

else

Self.agentMode4 := enteringFinished

endif

where

RefinementUndef(snwen:StateNodeWithEntryPoint) ≡

let sn:[StateNode] =

take({sn1 ∈ StateNode: directlyInheritsFrom(snwen.s-StateNode,sn1)}) in

if sn ≠ undefined then

// refinement possibly inherited

Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪

{mk**-**StateNodeWithEntryPoint(sn,

snwen.s-implicit)}

else

Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen}

endif

endlet

RefinementStateAggrNode(snwen:StateNodeWithEntryPoint) ≡

if snwen.s-implicit = HISTORY then

Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪

{ mk**-**StateNodeWithEntryPoint(s, HISTORY) |

s ∈ snwen.s-StateNode.previousSubStates }

else

Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪

{ mk**-**StateNodeWithEntryPoint(sp,

entryConnection(snwen.s-implicit, sp)) |

sp ∈ snwen.s-StateNode.statePartitionSet}

endif

let cstd: Composite-state-type-definition =

snwen.s-StateNode. stateDefinitionAS1 in

let aggr: State-aggregation-node = cstd.s-implicit in

if aggr.s-Entry-procedure-definition ≠ undefined then

CreateProcedure(aggr.s-Entry-procedure-definition, undefined,

undefined)

endif

endlet

RefinementCompStateNode(snwen:StateNodeWithEntryPoint) ≡

Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen}

let cstd: Composite-state-type-definition = snwen.s-StateNode.stateDefinitionAS1 in

let comp: Composite-state-graph = cstd.s-implicit in

if comp.s-Entry-procedure-definition ≠ undefined then

CreateProcedure(comp.s-Entry-procedure-definition, undefined,

undefined)

endif

endlet

if snwen.s-implicit = HISTORY then

Self.stateNodesToBeEntered := Self.stateNodesToBeEntered \ {snwen} ∪

{ mk**-**StateNodeWithEntryPoint(s, HISTORY) |

s ∈ snwen.s-StateNode.previousSubStates }

else

Self.currentStartNodes := Self.currentStartNodes ∪ {snwen}

endif

endwhere

Entering of state nodes continues until the set stateNodesToBeEntered is empty. A distinction is made between state nodes with and without a refinement. If there is a refinement into a state aggregation node, then the entry procedure of that node is to be executed, and all state partitions are to be entered. If there is a refinement into a composite state graph, then a start transition has to be selected and executed, which determines a substate to be entered. Finally, if the state node is not refined, it may be belong to a composite state with a state type inheriting from another state type, where it is refined.

EnterStateNodesEnteringFinished ≡

Self.agentMode2 := selectingTransition

Self.agentMode3 := startSelection

ReturnExecRight

When the set stateNodesToBeEntered is empty, the transition selection is activated by setting the agent modes accordingly.

##### F3.2.3.2.16 Leaving of state nodes

LeaveStateNodes ≡

if Self.agentMode4 = leavePhase then

LeaveStateNodesLeavePhase

elseif Self.agentMode4 = leavingFinished then

LeaveStateNodesLeavingFinished

endif

State nodes are left when transitions are fired. The set of state nodes to be left has already been determined when this rule macro is applied.

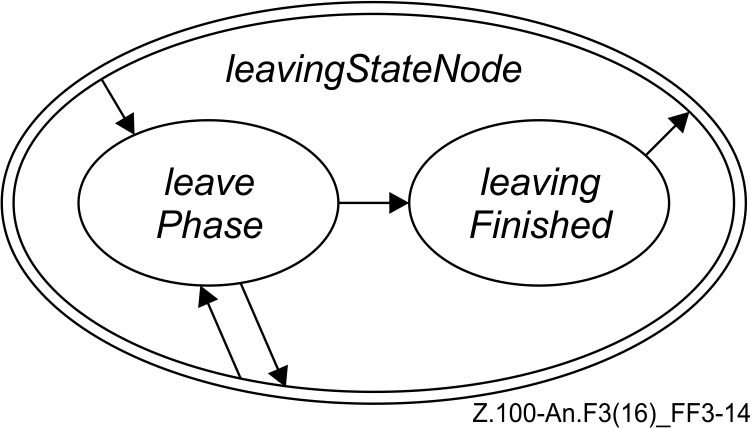


Figure F3-14 – Activity phases of SDL‑2010 agents: leaving state node (level 4)

LeaveStateNodesLeavePhase ≡

let sn = Self.stateNodesToBeLeft.selectNextStateNode in

if sn = undefined then

Self.agentMode4 := leavingFinished

else

Self.stateNodesToBeLeft := Self.stateNodesToBeLeft \ {sn}

sn.parentStateNode.currentSubStates := sn.parentStateNode.currentSubStates \ {sn}

sn.parentStateNode.previousSubStates := sn.parentStateNode.previousSubStates ∪ {sn}

if sn.stateNodeRefinement = compositeStateGraph then

let cstd : Composite-state-type-definition =

sn.stateAS1.s-Composite-state-type-identifier.idToNodeAS1 in

let comp : Composite-state-graph = cstd.s-implicit in

if comp.s-Exit-procedure-definition ≠ undefined then

CreateProcedure(comp.s-Exit-procedure-definition,undefined,

undefined)

endif

endlet

elseif sn.stateNodeRefinement = stateAggregationNode then

let cstd: Composite-state-type-definition =

sn.stateAS1.s-Composite-state-type-identifier.idToNodeAS1 in

let aggr: State-aggregation-node = cstd.s-implicit in

if aggr.s-Exit-procedure-definition ≠ undefined then

CreateProcedure(aggr.s-Exit-procedure-definition, undefined,

undefined)

endif

endlet

endif

endif

endlet

In the leave phase, state nodes that have been collected are left, from bottom to top, with possible synchronization at state aggregation nodes. If defined, exit procedures are executed.

LeaveStateNodesLeavingFinished ≡

if Self.stateNodeToBeExited ≠ undefined then

Self.currentExitStateNodes := {Self.stateNodeToBeExited}

Self.stateNodeToBeExited := undefined

Self.agentMode3 := exitingCompositeState

else

Self.agentMode3 := firingAction

Self.currentLabel := Self.continueLabel

Self.continueLabel := undefined

endif

When the leaving of a state node has been completed, either the exiting of a state node or firing of the current transition has to be continued.

##### F3.2.3.2.17 Exiting of composite states

ExitCompositeState ≡

if Self.stateNodeToBeExited ≠ undefined then

let sn = Self.stateNodeToBeExited.s-StateNode in

if sn.stateNodeKind = stateNode then

Self.currentExitStateNodes := {Self.stateNodeToBeExited}

Self.stateNodeToBeExited := undefined

Self.agentMode2 := selectingTransition

Self.agentMode3 := startPhase

elseif sn.stateNodeKind = statePartition then

sn.parentStateNode.currentExitPoints := sn.parentStateNode.currentExitPoints

∪ {Self.stateNodeToBeExited.s-StateExitPoint}

Self.stateNodesToBeLeft := {sn}

Self.agentMode3 := leavingStateNode

Self.agentMode4 := leavePhase

endif

endlet

elseif Self.currentExitStateNodes ≠ ∅ then

let snwex = take(Self.currentExitStateNodes) in

let sn = snwex.s-StateNode in

if sn.parentStateNode.currentSubStates = ∅ then

let ep = take(sn.parentStateNode.currentExitPoints) in

Self.stateNodeToBeExited := mk**-**StateNodeWithExitPoint(

sn.parentStateNode, exitConnection(ep,sn))

Self.currentExitStateNodes := ∅

endlet

else

Self.currentExitStateNodes := ∅

Self.agentMode2 := selectingTransition

Self.agentMode3 := startPhase

endif

endlet

endlet

endif

##### F3.2.3.2.18 Stopping agent execution

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

StopPhase ≡

if ∀sas ∈ SdlAgentSet: (sas.owner = Self ⇒ ¬ ∃sa ∈ SdlAgent: sa.owner = sas) then

RemoveAllAgentSets(Self)

RemoveAgent(Self)

endif

#### F3.2.3.3 Interface between execution and compilation

The execution of agents requires certain behaviour parts (called "compilation units") to be treated during compilation. Compilation units are sequences of actions of an agent that, once started, are executed without being interleaved by other actions of this agent or an agent belonging to the same set of nested agents:

• (Regular) transitions: Each transition starts with the evaluation of input parameters (if any), followed by an action "leaveStateNode", followed by Transition as defined in the abstract syntax. If the terminator of the transition is a Nextstate-node, the transition ends with an action "enterStateNode".

• Start transitions (Named-start-node, State-start-node, Procedure-start-node): These are associated with the containing state node.

• Exit transitions (Named-return-node): These are associated with the set of transitions of the containing state node.

• Expressions: During the selection phase, enabling conditions and continuous signals have to be evaluated. In these cases, the evaluation of an expression is a compilation unit.

Each compilation unit has a start label. Once a start label is assigned to the function currentLabel of an agent, the sequence of actions that begins with this label – the evaluation of an expression or the firing of a transition – is sequentially executed. This means that whenever an action has been executed, the compilation determines the continue label such that the next action follows. The termination of this sequence is "signalled" by having the continue label set to undefined after the last action of the sequence.

During compilation, a function uniqueLabel: DefinitionAS1 × Nat→ Label associates unique labels with each node of the AST. The unique labels of nodes corresponding to compilation units are used as starting labels. Furthermore, labels are used to retrieve the result of the evaluation of expressions.

## F3.3 Data semantics

### F3.3.1 Predefined data

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

functional(procedure: Procedure, values: Value\*): Boolean =def

( procedure.identifier1.s-Qualifier.head ∈ Package-qualifier ∧

procedure.identifier1.s-Qualifier.head.s-Package-name.s-Token = "Predefined")

∨ isSpecialStructOp(procedure)

∨ isSpecialLiteralOp(procedure)

intype(procedure: Procedure, name: Name): Boolean =def

procedure.identifier1.s-Qualifier.last.s-Data-type-name = name

compute (procedure: Procedure, values: Value\* ): ValueOrException =def

if intype (procedure, IntegerType.s-Name) then computeInteger(procedure, values)

elseif intype (procedure, BooleanType.s-Name) then computeBoolean(procedure, values)

elseif intype (procedure, CharacterType.s-Name) then computeChar(procedure, values)

elseif intype (procedure, RealType.s-Name) then computeReal(procedure, values)

elseif intype (procedure, DurationType.s-Name) then computeDuration(procedure, values)

elseif intype (procedure, TimeType.s-Name) then computeTime(procedure, values)

elseif intype (procedure, StringType.s-Name) then computeString(procedure, values)

elseif intype (procedure, ArrayType.s-Name) then computeArray(procedure, values)

elseif intype (procedure, PowersetType.s-Name) then computePowerset(procedure, values)

elseif intype (procedure, BagType.s-Name) then computeBag(procedure, values)

elseif isSpecialStructOp(procedure) then computeStruct(procedure, values)

elseif isSpecialLiteralOp (procedure) then computeLiteral(procedure, values)

else

raise(OutOfRange)

endif

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

emptyToken: Token =def

""

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

definingSort(p: Procedure): Identifier =def

p.parentAS1.identifier1

The function procName computes the token of an operator.

procName(p: Procedure): Token =def

p.s-Operation-name.s-Token

#### F3.3.1.1 Predefined Data Types, Exceptions and Boolean Operations

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

BooleanType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Boolean"))

CharacterType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Character"))

StringType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("String"))

IntegerType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Integer"))

RealType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Real"))

ArrayType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Array"))

PowersetType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Powerset"))

DurationType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Duration"))

TimeType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Time"))

BagType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Bag"))

PidType: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Pid"))

Furthermore, there are a number of predefined identifiers for exceptions.

OutOfRange: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("OutOfRange"))

InvalidReference: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>,mk**-**Name("InvalidReference"))

NoMatchingAnswer: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>,mk**-**Name("NoMatchingAnswer "))

UndefinedVariable: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>,mk**-**Name("UndefinedVariable"))

UndefinedField: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("UndefinedField"))

InvalidIndex: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("InvalidIndex"))

DivisionByZero: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("DivisionByZero"))

EmptyException: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("Empty"))

InvalidCall: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("InvalidCall"))

InvalidSort: Identifier =def

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("InvalidSort"))

To raise an exception, the function raise is used. Each Predefined exception is an Identifier and is a member of the Exception domain (see clause F3.2.1.1.6). If raise is invoked the further behaviour of the system is not defined by SDL‑2010.

raise(ex:Identifier): Identifier =def

UndefinedBehaviour

There are also the following predefined operation signatures:

sdlAnd: Static-operation-signature =def

mk**-**Operation-signature(mk**-**Name("and"),

<(BooleanType), (BooleanType)>, **mk-**Operation-result(BooleanType),

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("and")))

sdlOr: Static-operation-signature =def

mk**-**Operation-signature(mk**-**Name("or"),

< (BooleanType), (BooleanType)> , **mk-**Operation-result(BooleanType),

mk**-**Identifier(<mk**-**Package-qualifier(mk**-**Name("Predefined"))>, mk**-**Name("or")))

sdlTrue: Literal-signature =def

**mk-**Literal-signature (**mk-**Name("true"), **mk-**Result(BooleanType, **PART**), 0)

#### F3.3.1.2 Boolean

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

SDLBoolean =def Boolean × Identifier

computeBoolean(procedure: Procedure, values: Value\*): ValueOrException =def

let restype = definingSort(procedure) in

case procedure.procName of

| "not" => mk**-**SDLBoolean(¬ values.head.semvalueBool, restype)

| "and" => mk**-**SDLBoolean(values.head.semvalueBool ∧ values.tail.head.semvalueBool, restype)

| "or" => mk**-**SDLBoolean(values.head.semvalueBool ∨ values.tail.head.semvalueBool, restype)

| "xor" => mk**-**SDLBoolean(¬ (values. head.semvalueBool  values.tail.head.semvalueBool),

restype)

| "=>" => mk**-**SDLBoolean(values.head.semvalueBool  values.tail.head.semvalueBool,

restype)

endcase

endlet

semvalueBool(v:SDLBoolean): Boolean =def v.s-Boolean

#### F3.3.1.3 Integer

SDLInteger =def Nat × Identifier

semvalueInt(v:SDLInteger): Nat=def v.s-Nat

computeInteger(procedure: Procedure, values: Value\*): ValueOrException =def

let restype = definingSort(procedure) in

if procedure ∈ Literal-signature then

integerLiteral(0,procedure.procName, restype)

elseif procedure.procName = "-" ∧ values.length = 1 then

mk**-**SDLInteger(0 - values.head.semvalueInt, restype)

elseif procedure.procName ∈ {"+", "-", "\*", "/", "mod", "rem", "<", ">", "<=", ">=", "power"}

then

let val1 = values[1]. semvalueInt, val2 = values[2]. semvalueInt in

case procedure.procName of

| "+" => mk**-**SDLInteger (val1+val2, restype)

| "-"=> mk**-**SDLInteger (val1 – val2, restype)

| "\*"=> mk**-**SDLInteger (val1 \* val2, restype)

| "/"=>

if val2 = 0 then

raise(DivisionByZero)

else

mk**-**SDLInteger (intDiv(val1,val2), restype)

endif

| "mod"=>

if val2 = 0 then

raise(DivisionByZero)

else

mk**-**SDLInteger (intMod(val1,val2), restype)

endif

| "rem"=>

if val2 = 0 then

raise(DivisionByZero)

else

mk**-**SDLInteger (intRem(val1,val2), restype)

endif

| "power"=> mk**-**SDLInteger (intPower(val1,val2), restype)

| "<" => mk**-**SDLBoolean(val1 < val2, BooleanType)

| "<=" => mk**-**SDLBoolean(val1 ≤ val2, BooleanType)

| ">" => mk**-**SDLBoolean(val1 > val2, BooleanType)

| ">="=> mk**-**SDLBoolean(val1 ≥ val2, BooleanType)

endcase

endlet

else raise(OutOfRange)

endif

endlet

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

numberValue(c:Token): Nat =def

case c of

| "0" => 0

| "1" => 1

| "2" => 2

| "3" => 3

| "4" => 4

| "5" => 5

| "6" => 6

| "7" => 7

| "8" => 8

| "9" => 9

endcase

The function integerLiteral returns the SDLInteger value for an integer literal.

integerLiteral(num: Nat, proc: Token, type: Identifier): SDLInteger =def

if proc = emptyToken then

mk**-**SDLInteger (num, type)

else

integerLiteral(num\*10 + numberValue(proc.head), proc.tail, type)

endif

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

intDiv(a: Nat, b: Nat):Nat =def

if a ≥ 0 ∧ b > a then 0

elseif a ≥ 0 ∧ b ≤ a ∧ b > 0 then 1 + intDiv(a - b, b)

elseif a ≥ 0 ∧ b < 0 then - intDiv(a, -b)

elseif a < 0 ∧ b < 0 then intDiv (-a, -b)

elseif a < 0 ∧ b > 0 then - intDiv (-a, b)

else raise(DivisionByZero)

endif

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

intMod(a: Nat, b: Nat):Nat =def

if a ≥ 0 ∧ b > 0 then intRem(a,b)

elseif b < 0 then intMod(a, -b)

elseif a < 0 ∧ b > 0 ∧ intRem(a,b) = 0 then intRem(a,b)

elseif a < 0 ∧ b >0 ∧ intRem(a,b) < 0 then b + intRem(a,b)

else raise(DivisionByZero)

endif

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

intRem(a: Nat, b: Nat):Nat =def

a - b \* intDiv(a,b)

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

intPower(a: Nat, b: Nat):Nat =def

if b = 0 then 1

elseif a = 0 then 0

elseif b > 0 then a \* intPower(a, b-1)

else intDiv(intPower(a, b+1), a)

endif

#### F3.3.1.4 Character

Character values are represented by their name.

SDLCharacter =def Name × Identifier

computeChar(procedure: Procedure, values: Value\*): ValueOrException =def

let restype = definingSort(procedure) in

if procedure ∈ Literal-signature then

mk**-**SDLCharacter(procedure.s-Literal-name, restype)

elseif procedure.procName = "num" then

mk**-**SDLInteger(charValue(values.head.s-Name), IntegerType)

elseif procedure.procName = "chr" then

mk**-**SDLCharacter( values.head.semvalueInt.charChr, restype)

else raise(OutOfRange)

endif

endlet

The function charValue returns the numerical value of the character.

charValue(ch: Name): Nat =def

let myDef: Value-data-type-definition = CharacterType.idToNodeAS1 in

let literals = myDef.s-Literal-signature-set in

take({L.s-Literal-natural | L ∈ literals: L.s-Literal-name = ch})

endlet

The function charChr returns the character for a given Integer.

charChr(a: Nat): Name =def

if a > 128 then charChr(a - 128)

elseif a < 0 then charChr(a+128)

else

let char: Value-data-type-definition = CharacterType.idToNodeAS1 in

let literals = char.s-Literal-signature-set in

take({L.s-Literal-name | L ∈ literals: L.Literal-natural = a})

endif

#### F3.3.1.5 Real

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

SDLReal =def Nat × Nat × Identifier

semvalueRealNum(v: SDLReal): Nat =def v.s-Nat

semvalueRealDen(v: SDLReal): Nat =def v.s2-Nat

semvalueReal(v: SDLReal): Real=def

let res: Real = v.semvalueRealNum / v.semvalueRealDen in

res

endlet

computeReal(procedure: Procedure, values: Value\*): ValueOrException =def

let restype = definingSort(procedure) in

if procedure ∈ Literal-signature then

realLiteral(0,1,procedure.procName, restype)

elseif procedure.procName = "-" ∧ values.length = 1 then

mk**-**SDLReal(0 - values.head.semvalueRealNum, values.head.semvalueRealDen, restype)

elseif procedure.procName ∈ {"+", "-", "\*", "/", "<", ">", "<=", ">="} then

let num1 = values[1].semvalueRealNum in

let den1 = values[1]. semvalueRealDen in

let num2 = values[2]. semvalueRealNum in

let den2 = values[2]. semvalueRealDen in

case procedure.procName of

| "+" => mk**-**SDLReal(num1\*den2 + num2\*den1, den1\*den2, restype)

| "-"=> mk**-**SDLReal(num1\*den2 - num2\*den1, den1\*den2, restype)

| "\*"=> mk**-**SDLReal(num1\*num2, den1\*den2, restype)

| "/"=>

if num2 = 0 then

raise(DivisionByZero)

else

mk**-**SDLReal(num1\*num2, den1\*den2, restype)

endif

| "<" => mk**-**SDLBoolean(num1\*den2 < num2\*den1, BooleanType)

| "<=" => mk**-**SDLBoolean(num1\*den2 ≤ num2\*den1, BooleanType)

| ">" => mk**-**SDLBoolean(num1\*den2 ≥ num2\*den1, BooleanType)

| ">="=> mk**-**SDLBoolean(num1\*den2 ≥ num2\*den1, BooleanType)

endcase

endlet

elseif procedure.procName = "float" then

mk**-**SDLReal(semvalueInt(values.head), 1, restype)

elseif procedure.procName = "fix" then

mk**-**SDLInteger(computeFix(values.head.semvalueRealNum,

values.head.semvalueRealDen), IntegerType)

else raise(OutOfRange)

endif

endlet

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

realLiteral(num: Nat, den: Nat, proc: Token, type: Identifier): SDLReal =def

if proc = emptyToken then

mk**-**SDLReal(num, den, type)

elseif proc.head = "." then

realLiteral(num\*10,den\*10, proc.tail, type )

elseif den = 1 then

realLiteral(num\*10 + numberValue(proc.head), den, proc.tail, type)

else

realLiteral(num\*10 + numberValue(proc.head), den, proc.tail, type)

endif

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

computeFix(num: Nat, den: Nat): Nat =def

if num < 0 then

- computeFix(- num, den) - 1

elseif num < den then

0

else

computeFix (num - den, den) + 1

endif

#### F3.3.1.6 Duration

The domain SDLDuration is based on the domain SDLReal.

SDLDuration =def Duration × Identifier

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

computeReal(procedure, values)

#### F3.3.1.7 Time

The domain SDLTime is based on the domain SDLReal.

SDLTime=def Time × Identifier

computeTime(procedure: Procedure, values: Value\*): Value =def

let restype = definingSort(procedure) in

if procedure ∈ Literal-signature then

realLiteral(0,1,procedure.procName, restype)

else

case procedure.procName of

| "time"=>

let val: SDLReal = values.head in

mk**-**SDLReal(val.s-Nat, val.s2-Nat, RealType)

endlet

| "<" => computeReal(procedure, values)

| "<=" => computeReal(procedure, values)

| ">" => computeReal(procedure, values)

| ">=" => computeReal(procedure, values)

| "+" => computeReal(procedure, values)

| "-" =>

if values.head ∈ SDLTime ∧ values.tail.head ∈ SDLDuration then

computeReal(procedure, values)

else

let res: SDLReal = computeReal(procedure,values) in

mk**-**SDLReal(res.s-Nat, res.s2-Nat, RealType)

endlet

endif

endcase

endif

endlet

#### F3.3.1.8 String

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

SDLString =def Value \* × Identifier

computeString (procedure: Procedure, values: Value\*): ValueOrException =def

let restype = definingSort(procedure) in

case procedure.procName of

| "emptystring"=> mk**-**SDLString(empty, restype)

| "mkstring"=> mk**-**SDLString(<values.head>, restype)

| "make"=> mk**-**SDLString(<values.head>, restype)

| "length"=> mk**-**SDLInteger (values.head. s-Value-seq.length, IntegerType)

| "first"=> values.head. s-Value-seq.head

| "last"=> values.head. s-Value-seq.last

| "//"=> mk**-**SDLString(values[1]. s-Value-seq ⁀ values[2].s-Value-seq, restype)

| "extract"=>

let string = values[1]. s-Value-seq in

let intval: SDLInteger = values[2] in

let index = intval.s-Nat in

if index < 0 ∨ index > string.length then

raise(InvalidIndex)

else

string[index]

endif

endlet

| "modify"=>

let intval: SDLInteger = values[2] in

let index = intval.s-Nat in

let front = substr(values[1].s-Value-seq, 1, index-1) in

let back = substr(values[1].s-Value-seq, index+1, values[1].s-Value-seq.length - index) in

if InvalidIndex = front ∨ InvalidIndex = back then raise(InvalidIndex)

else

mk**-**SDLString(front ⁀ <values[3]> ⁀ back, restype)

endif

endlet

| "substring"=>

let from: SDLInteger = values[2] in

let to: SDLInteger = values[3] in

let val = substr(values[1].s-Value-seq, from.s-Nat, to.s-Nat) in

if InvalidIndex = val then raise(InvalidIndex)

else mk**-**SDLString(val, restype) endif

endlet

| "remove"=>

let intval: SDLInteger = values[2] in

let index = intval.s-Nat in

let front = substr(values[1].s-Value-seq, 1, index-1) in

let back = substr(values[1].s-Value-seq, index+1, values[1].s-Value-seq.length - index) in

if InvalidIndex = front ∨ InvalidIndex = back then raise(InvalidIndex) else

mk**-**SDLString(front ⁀ back, restype)

endif

endlet

endcase

endlet

The function substr computes the substring of a string value.

substr(str: Value\*,start: Nat, len: Nat): Value\* ∪ Exception =def

if start ≤ 0 ∨ len ≤ 0 ∨ start+len-1 > str.length then

raise(InvalidIndex)

elseif len = 0 then

empty

else

substr(str,start,len-1) ⁀ <str[start+len-1] >

endif

#### F3.3.1.9 Array

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

SDLArray =def ValuePair-set × [Value] × Identifier

ValuePair =def Value × Value

computeArray(procedure: Procedure, values: Value\*): ValueOrException =def

let restype = definingSort(procedure) in

if procedure.procName = "Make" then

if values.length = 0 then

mk**-**SDLArray(∅, undefined, restype)

else

mk**-**SDLArray(∅,values.head, restype)

endif

elseif procedure.procName = "Modify" then

let a = values[1], index = values[2], value = values[3] in

mk**-**SDLArray(modifyArray(a.s-ValuePair-set, index, value), a.s-Value, restype)

endlet

elseif procedure.procName = "Extract" then

let v = take({ f.s2-Value | f ∈ values[1].s-ValuePair-set: f.s-Value = values[2]}) in

if v = undefined then

if values[1].s-Value = undefined then

raise(InvalidIndex)

else

values[1].s-Value

else

v

endlet

else raise(OutOfRange)

endif

endlet

modifyArray(a: ValuePair-set, index: Value, value: Value): ValuePair-set =def

{ item | item ∈ a: item.s-Value  index } ∪ { mk**-**ValuePair(index,value)}

#### F3.3.1.10 Powerset

A Powerset is represented as a set.

SDLPowerset =def Value-set × Identifier

computePowerset (procedure: Procedure, values: Value\*): ValueOrException =def

let restype = definingSort(procedure) in

case procedure.procName of

| "empty"=> mk**-**SDLPowerset(∅,restype)

| "in"=> mk**-**SDLBoolean(values[1] ∈ values[2].s-Value-set, BooleanType)

| "incl"=> mk**-**SDLPowerset(values[2].s-Value-set ∪ {values[1] }, restype)

| "del"=> mk**-**SDLPowerset(values[2].s-Value-set \ {values[1] }, restype)

| "<"=> mk**-**SDLBoolean(values[1].s-Value-set ⊂ values[2].s-Value-set, BooleanType)

| "<="=> mk**-**SDLBoolean(values[1].s-Value-set ⊆ values[2].s-Value-set, BooleanType)

| ">"=> mk**-**SDLBoolean(values[2].s-Value-set ⊂ values[1].s-Value-set, BooleanType)

| ">="=> mk**-**SDLBoolean(values[2].s-Value-set ⊆ values[1].s-Value-set, BooleanType)

| "and"=> mk**-**SDLPowerset(values[1].s-Value-set ∩ values[2].s-Value-set, restype)

| "or"=> mk**-**SDLPowerset(values[1].s-Value-set ∪ values[2].s-Value-set, restype)

| "length"=> mk**-**SDLInteger( | values[1].s-Value-set |, IntegerType)

| "take"=> if values[1].s-Value-set = ∅ then

raise(EmptyException)

else

values[1]. s-Value-set.take

endif

endcase

endlet

#### F3.3.1.11 Bag

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

SDLBag =def Frequency-set × Identifier

Frequency =def Value × Nat

computeBag (procedure: Procedure, values: Value\*): ValueOrException =def

let restype = definingSort(procedure) in

case procedure.procName of

| "empty"=> mk**-**SDLBag(∅,restype)

| "in"=> mk**-**SDLBoolean(bagcount(values[1], values[2])  0, BooleanType)

| "incl"=> mk**-**SDLBag(bagincl(values[1], values[2]), restype)

| "del"=> mk**-**SDLBag(bagdel(values[1], values[2]), restype)

| "<"=> mk**-**SDLBoolean(baginbag(values[1], values[2]), BooleanType)

| "<="=> mk**-**SDLBoolean(¬ baginbag(values[2], values[1]), BooleanType)

| ">"=> mk**-**SDLBoolean(baginbag(values[2], values[1]), BooleanType)

| ">="=> mk**-**SDLBoolean(¬ baginbag(values[1], values[2]), BooleanType)

| "and"=> mk**-**SDLBag(bagand(values[1], values[2]), restype)

| "or"=> mk**-**SDLBag(bagor(values[1], values[2]), restype)

| "length"=> mk**-**SDLInteger(baglength(values[1].s-Frequency-set), IntegerType)

| "take"=> values[1].s-Frequency-set.take.s-Value

endcase

endlet

bagcount(item: Value, bag: SDLBag): Nat =def

let elem1 = {elem.s-Nat | elem ∈ bag.s-Frequency-set: elem.s-Value = item } in

if elem1 = ∅ then 0 else elem1.take endif

endlet

bagincl(item: Value, bag: SDLBag): Frequency-set =def

if bagcount(item, bag)  0 then

{if elem.s-Value = item then mk**-**Frequency(item, elem.s-Nat+1) else elem endif |

elem ∈ bag.s-Frequency-set}

else

bag.s-Frequency-set ∪ {mk**-**Frequency (item, 1)}

endif

bagdel(item: Value, bag: SDLBag): Frequency-set =def

if bagcount(item, bag)  1 then

{if elem.s-Value = item then mk**-**Frequency(item, elem.s-Nat - 1) else elem endif |

elem ∈ bag.s-Frequency-set}

else

bag.s-Frequency-set \ { mk**-**Frequency(item, 1)}

endif

baginbag(smaller: SDLBag, larger: SDLBag): Boolean =def

∀ elem ∈ smaller.s-Frequency-set: bagcount(elem.s-Value, larger) < elem.s-Nat

bagand(a: SDLBag,b: SDLBag): Frequency-set =def

{ mk**-**Frequency (x.s-Value, min(bagcount(x.s-Value,a),bagcount(x.s-Value,b))) |

x ∈ a.s-Frequency-set: bagcount(x.s-Value, b) > 0}

min(a: Nat,b: Nat ): Nat =def if a>b then a else b endif

bagor(a: SDLBag,b: SDLBag): Frequency-set =def

{ mk**-**Frequency(x.s-Value, bagcount(x.s-Value,a) + bagcount(x.s-Value, b) )

| x ∈ a.s-Frequency-set }

∪ { x | x ∈ b.s-Frequency-set: bagcount(x.s-Value, a) = 0}

baglength(a: Frequency-set):Nat =def

if a = ∅ then 0

else let x = a.take in

x.s-Nat + baglength(a \ {x})

endlet

endif

### F3.3.2 Pid types

A Pid value is represented by an agent and an interface.

Pid =def ValidPid ∪ NullPid

NullPid =def { mk**-**Null-literal-signature(mk**-**Name("null"), Pidtype, undefined) }

ValidPid =def SdlAgent × [Interface-definition]

static nullPid: Pid =def take(NullPid)

The static function nullPid is the special Pid value for the unique named element of the Pid sort (denoted by "null") that does not identify any agent and is the unique element of NullPid.

### F3.3.3 Constructed types

#### F3.3.3.1 Structures

A structure value is identified by its type name, and the field list. The field names are a list, rather than a set because Make operator uses the order of the fields rather than the field names.

SDLStructure =def Field\* × Identifier

Field =def Name × Value

isSpecialStructOp(procedure: Procedure): Boolean =def

let procsort = procedure.definingSort, pn = procedure.procName in

(∃ str ∈ SDLStructure: (procsort = str.s-Identifier )) ∧

( (pn = "Make")

∨ (pn = "Undefined")

∨ (∃ fld ∈ procsort.s-Field-seq: (pn = fld.s-Name ⁀ "Modify"))

∨ (∃ fld ∈ procsort.s-Field-seq: (pn = fld.s-Name ⁀ "Extract"))

∨ (∃ fld ∈ procsort.s-Field-seq: (pn = fld.s-Name ⁀ "Present"))

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

computeStruct(procedure: Procedure, values: Value\*): ValueOrException =def

let structsort = definingSort(procedure), pn = procedure.procName in

if pn = "Undefined" then

structUndefined(structsort)

elseif pn = "Make " then

structMake(structsort, empty, structsort.s-Field-seq, values)

elseif (∃ fld ∈ structsort.s-Field-seq: (pn = fld.s-Name ⁀ "Modify") then

let fn ⁀ "Modify" = pn in

structModify(fn, structsort, values.head, empty, structsort.s-Field-seq)

endlet

elseif (∃ fld ∈ structsort.s-Field-seq: (pn = fld.s-Name ⁀ "Extract") then

let fn ⁀ "Extract" = pn in

structExtract(fn, structsort)

endlet

elseif (∃ fld ∈ structsort.s-Field-seq: (pn = fld.s-Name ⁀ "Present")) then

let fn ⁀ "Present" = pn in

structFieldPresent(fn, structsort)

endlet

else raise(OutOfRange)

endif

endlet

The function structMake creates a structure value with the fields initialized to the list of values. It should be called externally (internally it is recursive) with a structure value, an empty list of new fields (newflds) and a list of old fields (oldflds) that each has a field name defined, and a list of one or more values. The new fields (newflds) and old fields (oldflds) are used in the internal recursion.

structMake(st: SDLStructure, newflds: Field\*, oldflds: Field\*, values: Value\*): Value =def

if values.length < oldflds.length then structMake(st, newflds, oldflds, values ⁀ undefined)

elseif values.length = 0 ∨ oldflds.length = 0 then

mk**-**SDLStructure(newflds, st.s-Identifier)

else

structMake(st, newflds ⁀ mk**-**Field( oldflds.head.s-Name, values.head),

oldflds.tail, values.tail )

endif

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

structUndefined(st: SDLStructure): SDLBoolean =def

mk**-**SDLBoolean(semvalueBool(∀value ∈ st.s-Field.s-Value: (value = undefined)), BooleanType)

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

structExtract(fieldname:Name, structtype: SDLStructure): Value =def

let valueset = { f.s-Value | f ∈ structtype.s-Field-seq: f.s-Name = fieldname} in

if valueset = ∅ then raise(UndefinedField)

else valueset.take

endif

endlet

The function structModify returns a new structure with one field changed. It should be called externally (internally it is recursive) with the field name, a structure value, the new value for the field, an empty list of new fields (newflds) and a list of old fields (oldflds) that each have a field name defined. The new fields (newflds) and old fields (oldflds) are used in the internal recursion.

structModify(fn: Name, struct: SDLStructure, val: Value, newflds: Field\*, oldflds: Field\*):

SDLStructure =def

if oldflds.length = 0 then

mk**-**SDLStructure(newflds, struct.s-Identifier)

else

structModify(fn, struct, val,

newflds ⁀

mk**-**Field(oldflds.head.s-Name,

if oldflds.head.s-Name  fieldname then val else oldflds.head.s-Value endif),

oldflds.tail)

endif

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

structFieldPresent(fn: Name, st: SDLStructure): SDLBoolean =def

mk**-**SDLBoolean(semvalueBool(fn.parentAS1.s-Field.s-Value ≠ undefined), BooleanType)

#### F3.3.3.2 Literals

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

SDLLiterals =def Literal-signature × Identifier

isSpecialLiteralOp(procedure: Procedure): Boolean =def

let procsort = procedure.definingSort, pn = procedure.procName in

(∃ lit ∈ SDLLiterals: (procsort = lit.s-Identifier )) ∧

( pn ∈ { "<", ">","<=",">=", "first", "last", "succ", "pred", "num" })

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

computeLiteral(procedure:Procedure, values:Value\*): [Value ]=def

let restype = definingSort(procedure) in

let defi: Value-data-type-definition = restype.idToNodeAS1 in

if procedure.procName ∈ { "<", ">","<=",">=" } then

let v1 = values.head.s-Literal-signature.literalNum in

let v2 = values.tail.head.s-Literal-signature.literalNum in

case procedure.procName of

| ">" => mk**-**SDLBoolean(v1 > v2, BooleanType)

| ">=" => mk**-**SDLBoolean(v1 ≥ v2, BooleanType)

| "<" => mk**-**SDLBoolean(v1 < v2, BooleanType)

| "<=" => mk**-**SDLBoolean(v1 ≤ v2, BooleanType)

endcase

endlet

elseif procedure.procName = "first" then

literalMinimum (defi.s-Literal-signature-set)

elseif procedure.procName = "last" then

literalMaximum (defi.s-Literal-signature-set)

elseif procedure. procName = "succ" then

literalSucc(defi.s-Literal-signature-set, values.head)

elseif procedure. procName = "pred" then

literalPred(defi.s-Literal-signature-set, values.head)

elseif procedure. procName = "num" then

**mk-**SDLInteger(literalNum(values.head).semvalueInt, IntegerType)

else

undefined

endif

endlet

literalNum(s: Literal-signature): Nat =def

s.s-Literal-natural

literalValue(s: Literal-signature): Value =def

**mk-**SDLLiterals(s, s.s-Result)

literalMinimum(s: Literal-signature-set): Value =def

take({s1.literalValue

| s1 ∈ s: ∀ s2∈s:s2.literalNum> s1.literalNum})

literalMaximum(s: Literal-signature-set): Value =def

take({s1.literalValue

| s1 ∈ s: ∀ s2∈s:s2.literalNum < s1.literalNum})

literalSucc(s: Literal-signature-set, val: SDLLiterals): Value =def

**if** val = literalMaximum (s, val.s-Identifier) **then** literalMinimum(s, val.s-Identifier)

**else**

take({s1.literalValue | s1 ∈ s ∧

(s1.literalNum > val.s-Nat) ∧

(∀s2 ∈ s: ( s2.literalNum ≤ s.literalNum ) ∨ (s1.literalNum ≤ s2.literalNum ))}

**endif**

literalPred(s: Literal-signature-set, val: SDLInteger): Value =def

**if** val = literalMinimum(s, val.s-Identifier) **then** literalMaximum (s, val.s-Identifier)

**else**

take({s1.literalValue | s1 ∈ s ∧

(s1.literalNum < val.s-Nat) ∧

(∀s2 ∈ s: ( s2.literalNum ≤ s1.literalNum ) ∨ (s.literalNum ≤ s2.literalNum ))}

**endif**

#### F3.3.3.3 Choice

Further study needed for this subject.

### F3.3.4 Variables with Aggregation-kind REF

Further study needed for this subject.

### F3.3.5 State access

The State domain consists of substates (associations of values for a specific StateId), and super states (associations between super state and substate). In case a certain variable is bound to an in/out parameter in a substate, it refers to the variable in the caller's state.

State =def NamedValue-set × SuperState-set

NamedValue =def StateId × Variable-identifier × [BoundValue]

BoundValue =def Value ∪ Variable-identifier

SuperState =def StateId × StateId

initAgentState(state: [State], newid: StateId, id: [StateId], declarations: Declaration-set): State =def

let newsub = initDeclarations(newid, declarations) in

if state = undefined then

mk**-**State(newsub, ∅, ∅)

else

let newsuper = if id = undefined then ∅ else { mk**-**SuperState(id, newid)} endif in

mk**-**State(state.s-NamedValue-set ∪ newsub, state.s-SuperState-set ∪ newsuper)

endif

endlet

initProcedureState(state: State, newid: StateId, id: StateId, vars: Declaration-set,

declarations: Declaration\*,

values:Value\*, variables: Variable-identifier\*): State =def

let newsub = assignValues(initDeclarations(newid, vars ∪ declarations.toSet),

newid,declarations,

values, variables) in

let newsuper = mk**-**SuperState(id, newid) in

mk**-**State(state.s-NamedValue-set ∪ newsub, state.s-SuperState-set ∪ { newsuper })

endlet

initDeclarations(newid: StateId, decls: Declaration-set): NamedValue-set =def

{ mk**-**NamedValue(newid, d.identifier1, d.s-Constant-expression)

| d ∈ decls: d ∈ Variable-definition} ∪

{ mk**-**NamedValue(newid, d.identifier1,

undefined)

| d ∈ decls: d ∈ Procedure-formal-parameter}

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

assignValues(namedvalues:NamedValue-set, id: StateId, decls:Declaration\*,

values:Value\*, variables:Variable-identifier\*): NamedValue-set =def

if values = empty then

namedvalues

else

if decls.head ∈ In-parameter then

assignValues(setValue(namedvalues, id, variables.head, values.head),

id, decls.tail, values.tail, variables.tail)

else

assignValues(namedvalues, id, decls.tail, values.tail, variables.tail)

endif

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

setValue(namedvalues: NamedValue-set, id: StateId, varname:Identifier, value:Value):

NamedValue-set =def

{ binding | binding ∈ namedvalues:

binding.s-Variable-identifier  varname ∨ binding.s-StateId  id} ∪

{ mk**-**NamedValue(id, varname, value) }

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

getValue(namedvalues: NamedValue-set, id: StateId, varname:Identifier): NamedValue-set =def

{ b ∈ namedvalues:

b.s-StateId = id ∧ b.s-Variable-identifier = varname}

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

eval(varname:Identifier, state:State, id:StateId): ValueOrException =def

let callerid = caller(state, id) in

let namedval = getValue(state.s-NamedValue-set, id, varname) in

if namedval ≠ ∅ then

if namedval.take.s-BoundValue ∈ Value then

namedval.take.s-BoundValue

elseif namedval.take.s-BoundValue ∈ Variable-identifier then

eval(namedval.take.s-BoundValue, state, callerid)

else // the BoundValue is undefined

raise(*UndefinedVariable*)

endif

else

eval(varname, state, callerid)

endif

endlet

endlet

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

update(name:Identifier, value:Value, state:State, id:StateId): State =def

let val = getValue(state.s-NamedValue-set, id, name) in

if val = ∅ then

update(name, value, state, caller(state, id))

elseif val.take ∈ NamedValue then

mk**-**State(setValue(state.s-NamedValue-set, id, name, value),

state.s-SuperState-set)

else

update(val.take.**s-**Variable-identifier, value, state, id)

endif

endlet

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

assign (variablename:Variable-identifier, value:Value, state:State, id:StateId): StateOrException =def

if isValueVariable(variablename) then

if isSyntypeVariable(variablename) ∧ ¬rangeCheck(variablename.variableSort, value ) then

raise(OutOfRange)

else update(variablename, value, state, id)

endif

else

// pid variable, sort of variable is an Interface-definition

if variablename.variableSort = value.interface ∨

isSuperType(variablename.variableSort, value.interface) then

update(variablename, value, state, id)

else

update(variablename, nullPid, state, id)

endif

endif

The function caller returns the state id that caused this state id to exist.

caller(state: State, id: StateId): StateId =def

take({ s.s-StateId | s ∈ state.s-SuperState-set: s.s2-StateId = id})

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

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

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

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

isValueVariable(variableid: Variable-identifier): Boolean =def

variableid.variableSort ∈ Value-data-type-definition

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

isSyntypeVariable(variableid: Variable-identifier): Boolean =def

variableid.idToNodeAS1.s-Sort-reference-identifier ∈ Syntype-identifier

interface(val: Value): Interface-definition =def

if val.sort ∈ Interface-definition then val.sort else undefined endif

The function sort gives the sort of a value, which for most domains (such as SDLBoolean or SDLStructure that form part of the Value domain) is found from the Identifier element of the domain. The exception is the Pid domain, which instead is either a NullPid that has the value nullPid, and is a PidType value, or is a ValidPid with an optional Interface-definition. In the case of a ValidPid without an Interface-definition, the value is a PidType value; otherwise the data type definition is the Interface-definition.

sort(val: Value): Data-type-definition =def

if val ∈ NullPid then PidType.idToNodeAS1

elseif val ∈ ValidPid then

if val.s-Interface-definition = undefined then PidType.idToNodeAS1

else val.s-Interface-definition

endif

else val.s-Identifier.idToNodeAS1

endif

### F3.3.6 Specialization

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

dynamicType(v: Value): Identifier =def

if v = nullPid then raise(OutOfRange) else

case v of

| SDLBoolean(\*,t) => t

| SDLInteger(\*, t) => t

| SDLCharacter(\*, t) => t

| SDLReal(\*,\*, t) => t

| SDLString(\*,t) => t

| SDLLiterals(\*,t) => t

| SDLStructure(\*,t) => t

| Pid(\*, t) => t

endcase

endif

### F3.3.7 Operators and methods

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

dispatch(procedure:Procedure, values:Value\*): Identifier =def

if procedure ∈ Static-operation-signature then

procedure.s-Identifier

else

let c = allDynamicCandidates(procedure) in

let c1 = matchingCandidates(c, values) in

bestMatch(c1)

endlet

endif

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

allDynamicCandidates(procedure:Procedure): Procedure-set =def

{ p | p ∈Operation-signature:

p.s-Operation-name = procedure.s-Operation-name }

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

matchingCandidates(procedures: Procedure-set, values: Value\*): Procedure-set =def

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

The function matchingCandidates returns the most specialized signature.

bestMatch(procedures:Procedure-set): Identifier =def

take({ p.s-Identifier | p ∈ procedures:

∀ q ∈ procedures: isSignatureCompatible(p.s-Formal-argument-seq,

q.s-Formal-argument-seq) })

The predicate isSignatureCompatible holds if p is compatible with q.

isSignatureCompatible(p:Formal-argument\*, q:Formal-argument\*): Boolean =def

if p = empty then

true

else

isSortCompatible(p.head.s-Argument, q.head.s-Argument) ∧

isSignatureCompatible(p.tail, q.tail)

endif

isSortCompatible(p: Sort-reference-identifier, r: Sort-reference-identifier): Boolean =def

(p = r ) ∨

isDirectlySortCompatible(p, r) ∨

(r.idToNodeAS1 ∈ Interface-definition ∧

(∃ q ∈ Sort-reference-identifier: (isSortCompatible(p, q) ∧ isSortCompatible(q, r))))

isDirectlySortCompatible(y: Sort-reference-identifier, z: Sort-reference-identifier): Boolean =def

if isSuperSort(z, y) then

if y.idToNodeAS1 ∈ Value-data-type-definition then

// true if y is <anchored sort> of the form **this** z

y.idToNodeAS1.s-Data-type-identifier = z

else // y is a pid sort (because not a value dat type) – and z is super sort of y

true

endif

else false

endif

isSuperSort(z Sort-reference-identifier, y: Sort-reference-identifier): Boolean =def

isSuperType(z, y) // see clause F2.2.1.6.4.

dynamicTypes(values:Value\*): Formal-argument\* =def

<mk**-**Formal-argument(dynamicType(v)) | v in values >

### F3.3.8 Syntypes

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

rangeCheck(syntype: Syntype-definition, value: Value): Boolean =def

∃ cond ∈ syntype.s-Range-condition.s-Condition-item-set:

conditionItemCheck(cond, value, syntype.s-Parent-sort-identifier)

The predicate conditionItemCheck holds if the condition is true for the value of the given type. If the condition is a size constraint, rewriting the concrete grammar creates an anonymous operation identified by the Operation-identifier of the Size-constraint that embodies the ranges specified, so the Open-range or Closed-range items in the abstract grammar of Size-constraint are redundant. An alternative would be to construct an anonymous procedure here based on the Open-range or Closed-range items of Size-constraint, in which case the Operation-identifier of Size-constraint is redundant.

conditionItemCheck(cond: Condition-item, value: Value, type: Identifier): Boolean =def

if cond ∈ Open-range then

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

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

elseif cond ∈ Closed-range then

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

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

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

endchoose

else //size constraint and cond ∈ Size-constraint

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

endif

Appendix I   
to Annex F3  
  
List of abstract syntax grammar rules used

This list contains the Specification and Description Language abstract syntax grammar rules that are used in this annex (Annex F3). The complete list of abstract syntax grammar rules can be found in Annex A of Recommendation ITU-T Z.100, which also identifies the Recommendation ([ITU‑T Z.101] or [ITU‑T Z.102] or [ITU‑T Z.104] or [ITU‑T Z.107]) where the grammar rule is defined. *Exception-identifier* is only defined in this annex (Annex F3) and is not defined or used in [ITU‑T Z.101] or [ITU‑T Z.102] or [ITU‑T Z.104] or [ITU‑T Z.107].

Action-return-node

Agent-definition

Agent-identifier

Agent-kind

Agent-type-definition

Agent-type-identifier

Any-expression

Argument

Assignment

Break-node

Call-node

Channel-definition

Channel-path

Closed-range

Composite-state-graph

Composite-state-type-definition

Composite-state-type-identifier

Compound-node

Condition-item

Conditional-expression

Connect-node

Connection-definition

Connector-name

Constant-expression

Continue-node

Continuous-expression

Continuous-signal

Create-request-node

Dash-nextstate

Data-type-definition

Data-type-identifier

Data-type-name

Decision-answer

Decision-node

Destination-gate

Entry-connection-definition

Entry-procedure-definition

Equality-expression

Exception-identifier

Exit-connection-definition

Exit-procedure-definition

Formal-argument

Free-action

Gate-definition

Graph-node

Identifier

In-parameter

In-signal-identifier

Initial-number

Inner-entry-point

Inner-exit-point

Input-node

Interface-definition

Join-node

Literal

Literal-name

Literal-natural

Literal-signature

Maximum-number

Name

Named-nextstate

Named-return-node

Named-start-node

Negative-equality-expression

Nextstate-parameters

Now-expression

Null-literal-signature

Number-of-instances

Offspring-expression

Open-range

Operation-application

Operation-identifier

Operation-name

Operation-signature

Originating-gate

Out-parameter

Out-signal-identifier

Outer-entry-point

Outer-exit-point

Output-node

Package-definition

Package-name

Package-qualifier

Parameter

Parent-expression

Parent-sort-identifier

Positive-equality-expression

Priority-name

Procedure-definition

Procedure-formal-parameter

Procedure-graph

Procedure-identifier

Procedure-start-node

Provided-expression

Qualifier

Range-check-expression

Range-condition

Reset-node

Result

Save-signalset

Self-expression

Sender-expression

Set-node

Signal-definition

Signal-identifier

Size-constraint

Sort-identifier

Sort-reference-identifier

Spontaneous-transition

State-aggregation-node

State-entry-point-name

State-exit-point-name

State-machine

State-name

State-node

State-partition

State-start-node

State-transition-graph

Static-operation-signature

Stop-node

Syntype-identifier

Syntype-definition

Terminator

Timer-active-expression

Timer-remaining-duration

Timer-definition

Transition

Value-data-type-definition

Value-return-node

Value-returning-call-node

Variable-access

Variable-definition

Variable-identifier

Further study needed on whether more explanation is needed on the use of *Name* and **mk-***Name*.

|  |  |
| --- | --- |
| **SERIES OF ITU-T RECOMMENDATIONS** | |
| Series A | Organization of the work of ITU-T |
| Series D | Tariff and accounting principles and international telecommunication/ICT economic and policy issues |
| Series E | Overall network operation, telephone service, service operation and human factors |
| Series F | Non-telephone telecommunication services |
| Series G | Transmission systems and media, digital systems and networks |
| Series H | Audiovisual and multimedia systems |
| Series I | Integrated services digital network |
| Series J | Cable networks and transmission of television, sound programme and other multimedia signals |
| Series K | Protection against interference |
| Series L | Environment and ICTs, climate change, e-waste, energy efficiency; construction, installation and protection of cables and other elements of outside plant |
| Series M | Telecommunication management, including TMN and network maintenance |
| Series N | Maintenance: international sound programme and television transmission circuits |
| Series O | Specifications of measuring equipment |
| Series P | Telephone transmission quality, telephone installations, local line networks |
| Series Q | Switching and signalling, and associated measurements and tests |
| Series R | Telegraph transmission |
| Series S | Telegraph services terminal equipment |
| Series T | Terminals for telematic services |
| Series U | Telegraph switching |
| Series V | Data communication over the telephone network |
| Series X | Data networks, open system communications and security |
| Series Y | Global information infrastructure, Internet protocol aspects, next-generation networks, Internet of Things and smart cities |
| **Series Z** | **Languages and general software aspects for telecommunication systems** |
|  |  |

1. \* To access the Recommendation, type the URL http://handle.itu.int/ in the address field of your web browser, followed by the Recommendation's unique ID. For example, <http://handle.itu.int/11.1002/1000/11830-en>. [↑](#footnote-ref-2)