INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100
Annex F.3
(03/93)

SERIES Z: PROGRAMMING LANGUAGES

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

Specification and Description Language (SDL) – SDL formal definition: Dynamic semantics

ITU-T Recommendation Z.100 – Annex F.3

(Previously CCITT Recommendation)

ITU-T Z-SERIES RECOMMENDATIONS

PROGRAMMING LANGUAGES

FORMAL DESCRIPTION TECHNIQUES (FDT)	
Specification and Description Language (SDL)	**Z.100–Z.109**
Application of Formal Description Techniques	Z.110–Z.119
Message Sequence Chart	Z.120–Z.129
PROGRAMMING LANGUAGES	
CHILL: The ITU-T 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.399
QUALITY OF TELECOMMUNICATION SOFTWARE	Z.400–Z.499
METHODS FOR VALIDATION AND TESTING	Z.500–Z.599

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

ITU-T RECOMMENDATIONS SERIES

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

4375

Printed in Switzerland
Geneva, 1998

FOREWORD

The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of the International Telecommunication Union. The 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 Conference (WTSC), which meets every four years, established the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

ITU-T Recommendation Z.100 – Annex F.3 was revised by the ITU-T Study Group X (1988-1993) and was approved by the WTSC (Helsinki, March 1-12, 1993).

____

NOTES

1 As a consequence of a reform process within the International Telecommunication Union (ITU), the CCITT ceased to exist as of 28 February 1993. In its place, the ITU Telecommunication Standardization Sector (ITU-T) was created as of 1 March 1993. Similarly, in this reform process, the CCIR and the IFRB have been replaced by the Radiocommunication Sector.

In order not to delay publication of this Recommendation, no change has been made in the text to references containing the acronyms "CCITT, CCIR or IFRB" or their associated entities such as Plenary Assembly, Secretariat, etc. Future editions of this Recommendation will contain the proper terminology related to the new ITU structure.

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

CONTENTS

Page

SPECIFICATION AND DESCRIPTION LANGUAGE (SDL) – SDL FORMAL DEFINITION: DYNAMIC SEMANTICS

(Melbourne, 1988; revised at Helsinki, 1993)

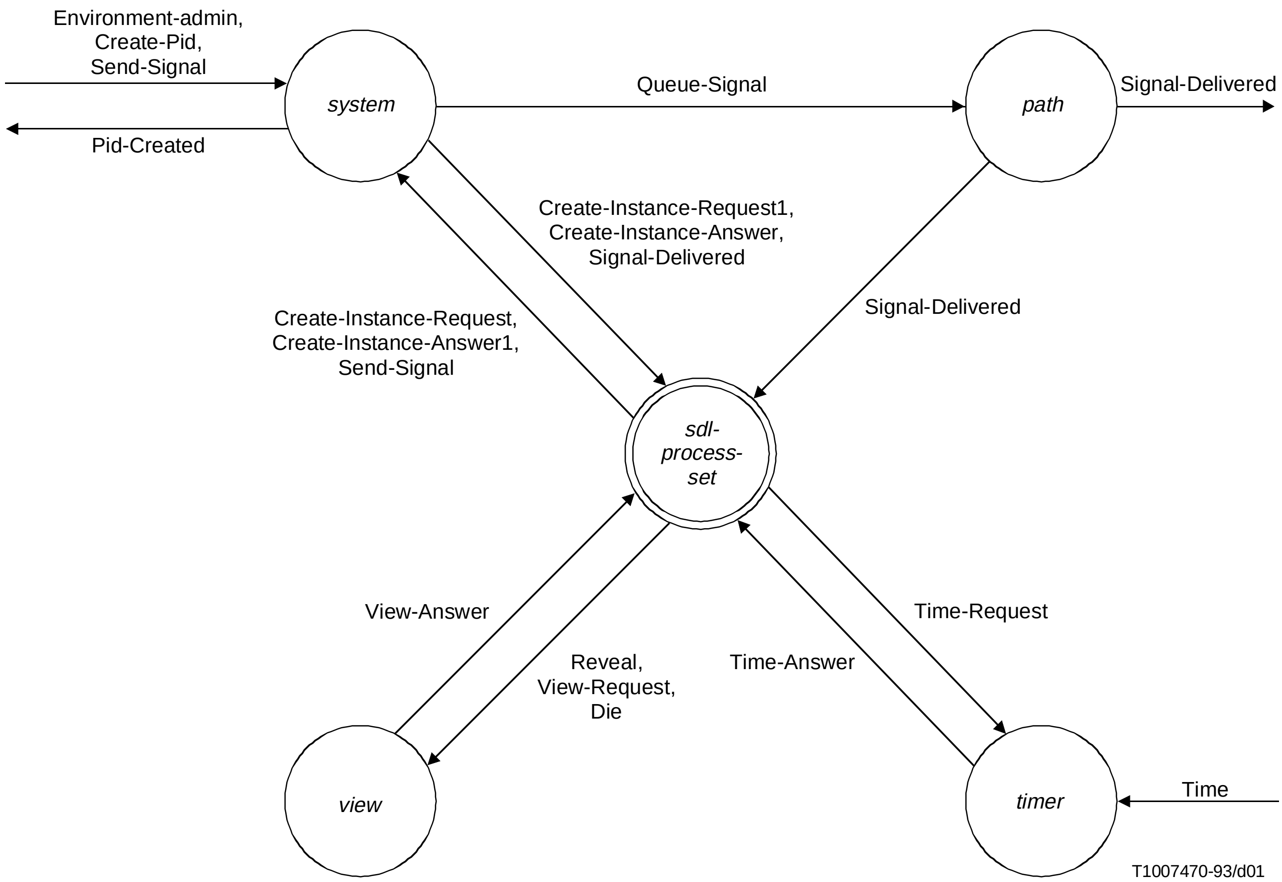

FIGURE 1/Z.100

Overall Structure of Interpretation Model

Introduction

This part of the Formal Definition defines the dynamic properties of SDL. For a description of the overall structure of the Formal Definition and for an explanation of the notation used, refer to Annex F.1: Introduction to the Formal Definition.

An SDL system is interpreted by a number of concurrent meta-processes. The communication between these is synchronous, CSP-like communication. The lines in figures 1 and 2 indicate communication by means of CSP-output.

Overall Interpretation Model

Figure 1 shows the overall structure of the interpretation model. The *system*-process is the “entry point” for interpretation of an SDL system and takes care of creating instances of the other processes: one instance of the *view*- and *timer*-process, one instance of the *path*-process for each distinct delaying path by which an SDL signal may be transported, and one instance of the *process-set-admin*-process (shown in the next figure) for each process instance set in the SDL system. The *process-set-admin*-process manages a couple of (meta-)processes which is shown as *sdlprocessset* in figure 1 and detailed in figure 2.

The processes are:

system Which handles the signal routing between SDL process instance sets and the generation of unique Pid values.

There is one living instance of *system* during the whole life time of the SDL system.

path Which handles the nondeterministic delay of channels. Note that all potential delays from the channels traversed by one signal instance have been added into one delay in an instance of *path*.

There is one living instance of *path* for each (non-empty) sequence of delaying channel paths which connects two leaf blocks (in the selected consistent subset) or one leaf block and the system environment. The meta-process instances are living during the whole life time of the SDL system.

view Which keeps track of all revealed variables. Each time an SDL process updates a revealed variable, it sends the new value to *view*. When a process is using the **view** expression, it will request the current value from *view*.

There is one living instance of *view* during the whole life time of the SDL system.

timer Which keeps track of the current time. When an SDL process is using the **now** expression it will request *timer* for the time value.

It is assumed that the environment in regular intervals sends a clock signal to the *timer*. This mechanism is sketched as the tick-process. It must be noted that the informal model of the tick-process does not form part of the dynamic semantics, it is only included for explanatory reasons.

There is one living instance of *timer* during the whole life time of the SDL system.

Interpretation Model for SDL Process Instance Set

Figure 2 shows the interpretation model for an SDL process instance set. The meta-process *process-set-admin* is the “entry point” for interpretation of an SDL process instance set and takes of creating one instance of the *input-port*- and *sdl-process*-processes whenever a new SDL process instance is to be created. If the SDL process is decomposed into services, the *sdl-process*-process creates one *sdl-service*-process for each service.

The processes are:

process-set-admin Which handles all ingoing SDL signals and create requests and manages the other meta-processes needed to interpret an SDL process instance set. A create request results in one instance of *input-port* and one instance of *sdl-process* unless this would lead to violation of the maximum number of SDL process instances. An ingoing signal is either directed to some *input-port* instance or discarded, depending on the receiver information conveyed with the signal and the current set of living SDL process instances.

There is one living instance of *process-set-admin* for each SDL process instance set. These meta-process instances are living during the whole life time of the SDL system.

input-port Which handles the queueing of signals in an SDL-process. Signals are always received by an *sdl-process* in its *input-port*. The *input-port* also takes care of timer handling.

At any point of time there is one living instance of *input-port* for each living SDL process instance.

sdl-process Which interprets the behaviour of an SDL process.

If the SDL process is *not* decomposed into services this implies interpretation of its process graph, and in this case *sdl-service* and its associated arrows in the figure do not apply.

If the SDL process *is* decomposed into services, *sdl-process* creates one instance of *sdl-service* for each SDL service. The *sdl-process* then coordinates the execution of the services such that all service start transitions are executed before any input and spontaneous transitions of the services, and such

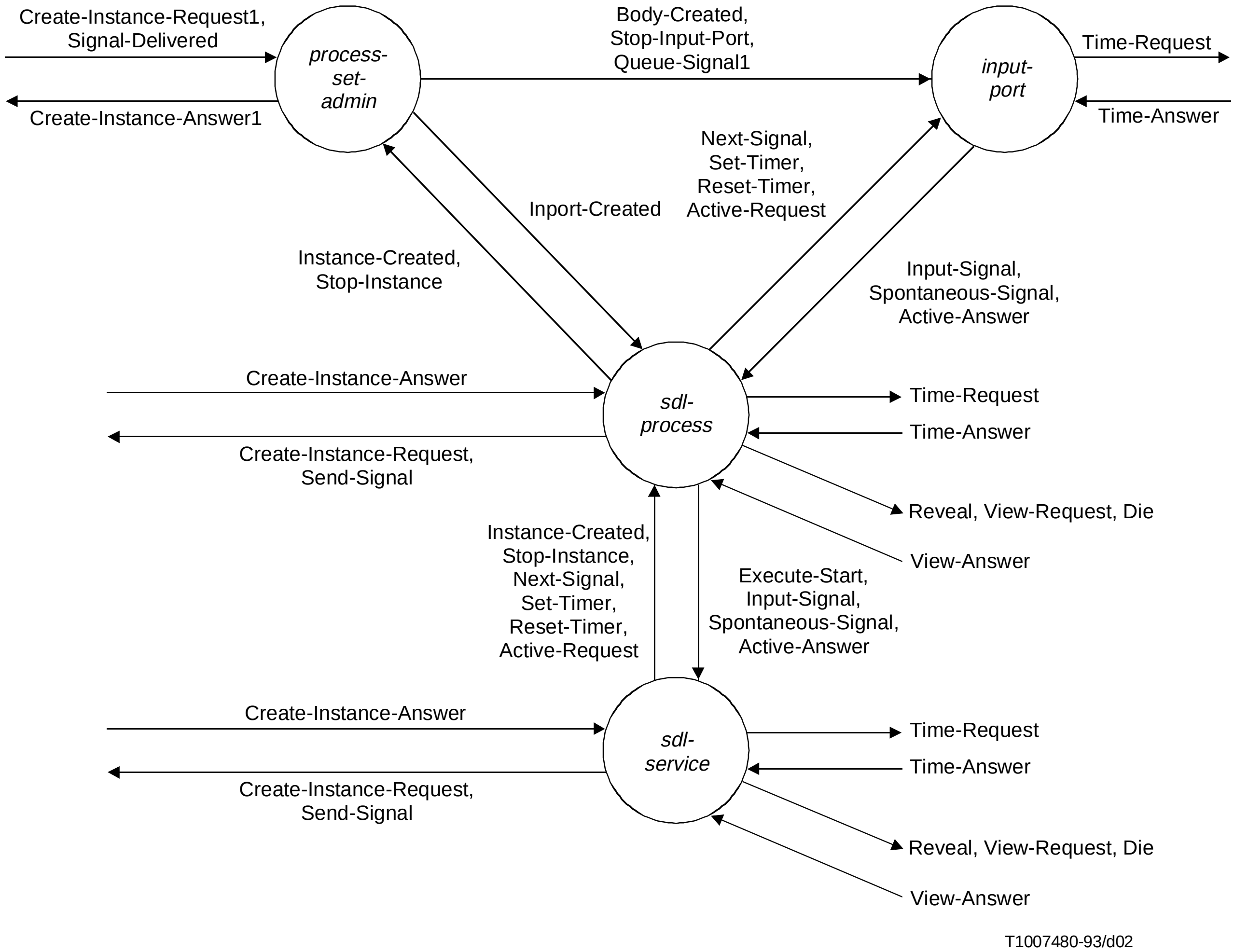

FIGURE 2/Z.100

Structure of Interpretation Model for SDL Process Instance Set

that no two service transitions are executed at the same time. All communication between *sdl-service* on one side and *process-set-admin* and *input-port* on the other goes through *sdl-process* which in several cases simply acts as a relay for this communication. This scheme has been chosen in order to make the interpretation functions for behaviour graph nodes as independent as possible of whether they occur in a process or service graph.

At any point of time there is one living instance of *sdl-process* for each living SDL process instance.

sdl-service Which interprets the behaviour of an SDL service.

At any point of time there is one living instance of *sdl-service* for each living SDL service instance.

1 SDL Abstract Syntax Summary

This section contains a summary of the abstract syntax (AS_1) domains for SDL as defined in Z.100. No further comments are attached to these domain definitions here.

1.1 Basic SDL

Visibility rules, names and identifiers

1	*Identifier*$_1$	::	*Qualifier*$_1$ *Name*$_1$
2	*Qualifier*$_1$	=	*Path-item*$_1$+
3	*Path-item*$_1$	=	*System-qualifier*$_1$ | *Block-qualifier*$_1$ | *Block-substructure-qualifier*$_1$ | *Process-qualifier*$_1$ | *Service-qualifier*$_1$ | *Procedure-qualifier*$_1$ | *Signal-qualifier*$_1$ | *Sort-qualifier*$_1$
4	*System-qualifier*$_1$	::	*System-name*$_1$
5	*Block-qualifier*$_1$	::	*Block-name*$_1$
6	*Block-substructure-qualifier*$_1$	::	*Block-substructure-name*$_1$
7	*Process-qualifier*$_1$	::	*Process-name*$_1$
8	*Service-qualifier*$_1$	::	*Service-name*$_1$
9	*Procedure-qualifier*$_1$	::	*Procedure-name*$_1$
10	*Signal-qualifier*$_1$	::	*Signal-name*$_1$
11	*Sort-qualifier*$_1$	::	*Sort-name*$_1$
12	*Name*$_1$	::	*Token*

Informal text

13	*Informal-text*$_1$	::	...

System

14	*System-definition*$_1$	::	*System-name*$_1$ *Block-definition*$_1$-**set** *Channel-definition*$_1$-**set** *Signal-definition*$_1$-**set** *Data-type-definition*$_1$ *Syn-type-definition*$_1$-**set**
15	*System-name*$_1$	=	*Name*$_1$

Block

16	*Block-definition*$_1$	::	*Block-name*$_1$ *Process-definition*$_1$-**set** *Signal-definition*$_1$-**set** *Channel-to-route-connection*$_1$-**set** *Signal-route-definition*$_1$-**set** *Data-type-definition*$_1$ *Syn-type-definition*$_1$-**set** [*Block-substructure-definition*$_1$]
17	*Block-name*$_1$	=	*Name*$_1$

Process

18	*Process-definition*$_1$	::	*Process-name*$_1$ *Number-of-instances*$_1$ *Process-formal-parameter*$_1$* *Procedure-definition*$_1$-**set** *Signal-definition*$_1$-**set** *Data-type-definition*$_1$ *Syn-type-definition*$_1$-**set** *Variable-definition*$_1$-**set** *View-definition*$_1$-**set** *Timer-definition*$_1$-**set** (*Process-graph*$_1$ | *Service-decomposition*$_1$)
19	*Number-of-instances*$_1$	::	*Intg* [*Intg*]
20	*Process-name*$_1$	=	*Name*$_1$
21	*Process-graph*$_1$	::	*Process-start-node*$_1$ *State-node*$_1$-**set**
22	*Process-formal-parameter*$_1$	::	*Variable-name*$_1$ *Sort-reference-identifier*$_1$
23	*Service-decomposition*$_1$	::	*Service-definition*$_1$-**set** *Signal-route-definition*$_1$-**set** *Signal-route-to-route-connection*$_1$-**set**

Service

24	*Service-definition*$_1$	::	*Service-name*$_1$ *Procedure-definition*$_1$-**set** *Data-type-definition*$_1$ *Syn-type-definition*$_1$-**set** *Variable-definition*$_1$-**set** *View-definition*$_1$-**set** *Timer-definition*$_1$-**set** *Service-graph*$_1$
25	*Service-name*$_1$	=	*Name*$_1$
26	*Service-graph*$_1$	::	*Service-start-node*$_1$ *State-node*$_1$-**set**
27	*Service-start-node*$_1$	::	*Transition*$_1$

Procedure

28	*Procedure-definition*$_1$	::	*Procedure-name*$_1$ *Procedure-formal-parameter*$_1$* *Procedure-definition*$_1$-**set** *Data-type-definition*$_1$ *Syn-type-definition*$_1$-**set** *Variable-definition*$_1$-**set** *Procedure-graph*$_1$
29	*Procedure-name*$_1$	=	*Name*$_1$
30	*Procedure-formal-parameter*$_1$	=	*In-parameter*$_1$ | *Inout-parameter*$_1$
31	*In-parameter*$_1$	::	*Variable-name*$_1$ *Sort-reference-identifier*$_1$
32	*Inout-parameter*$_1$	::	*Variable-name*$_1$ *Sort-reference-identifier*$_1$
33	*Procedure-graph*$_1$	::	*Procedure-start-node*$_1$ *State-node*$_1$-**set**

34	*Procedure-start-node*$_1$	::	*Transition*$_1$

Channel

35	*Channel-definition*$_1$	::	*Channel-name*$_1$ [NODELAY] *Channel-path*$_1$ [*Channel-path*$_1$]
36	*Channel-path*$_1$	::	*Originating-block*$_1$ *Destination-block*$_1$ *Signal-identifier*$_1$-**set**
37	*Originating-block*$_1$	=	*Block-identifier*$_1$ | ENVIRONMENT
38	*Destination-block*$_1$	=	*Block-identifier*$_1$ | ENVIRONMENT
39	*Block-identifier*$_1$	=	*Identifier*$_1$
40	*Signal-identifier*$_1$	=	*Identifier*$_1$
41	*Channel-name*$_1$	=	*Name*$_1$

Signal route

42	*Signal-route-definition*$_1$	::	*Signal-route-name*$_1$ *Signal-route-path*$_1$ [*Signal-route-path*$_1$]
43	*Signal-route-path*$_1$	::	*Origin*$_1$ *Destination*$_1$ *Signal-identifier*$_1$-**set**
44	*Origin*$_1$	=	*Process-identifier*$_1$ | *Service-identifier*$_1$ | ENVIRONMENT
45	*Destination*$_1$	=	*Process-identifier*$_1$ | *Service-identifier*$_1$ | ENVIRONMENT
46	*Signal-route-name*$_1$	=	*Name*$_1$
47	*Process-identifier*$_1$	=	*Identifier*$_1$
48	*Service-identifier*$_1$	=	*Identifier*$_1$

Connection

49	*Channel-to-route-connection*$_1$	::	*Channel-identifier*$_1$-**set** *Signal-route-identifier*$_1$-**set**
50	*Signal-route-identifier*$_1$	=	*Identifier*$_1$
51	*Signal-route-to-route-connection*$_1$	::	*External-signal-route-identifier*$_1$-**set** *Signal-route-identifier*$_1$-**set**
52	*External-signal-route-identifier*$_1$	=	*Identifier*$_1$

Signal

53	*Signal-definition*$_1$	::	*Signal-name*$_1$ *Sort-reference-identifier*$_1$* [*Signal-refinement*$_1$]
54	*Signal-name*$_1$	=	*Name*$_1$

Variable definition

55	*Variable-definition*$_1$	::	*Variable-name*$_1$ *Sort-reference-identifier*$_1$ [*Ground-expression*$_1$] [REVEALED]
56	*Variable-name*$_1$	=	*Name*$_1$

View definition

57	*View-definition*$_1$	::	*View-name*$_1$ *Sort-reference-identifier*$_1$
58	*View-name*$_1$	=	*Name*$_1$

Start

59	*Process-start-node*$_1$	::	*Transition*$_1$

State

60	*State-node*$_1$	::	*State-name*$_1$ *Save-signalset*$_1$ *Input-node*$_1$-**set** *Spontaneous-transition*$_1$-**set**
61	*State-name*$_1$	=	*Name*$_1$

Input

62	*Input-node*$_1$	::	*Signal-identifier*$_1$ [*Variable-identifier*$_1$]* *Transition*$_1$
63	*Variable-identifier*$_1$	=	*Identifier*$_1$

Save

64	*Save-signalset*$_1$	::	*Signal-identifier*$_1$-**set**

Spontaneous transition

65	*Spontaneous-transition*$_1$	::	*Transition*$_1$

Transition

66	*Transition*$_1$	::	*Graph-node*$_1$* (*Terminator*$_1$ | *Decision-node*$_1$)
67	*Graph-node*$_1$	=	*Task-node*$_1$ | *Output-node*$_1$ | *Create-request-node*$_1$ | *Call-node*$_1$ | *Set-node*$_1$ |

			Reset-node$_1$
68	*Terminator*$_1$	=	*Nextstate-node*$_1$ |
			Stop-node$_1$ |
			Return-node$_1$
69	*Nextstate-node*$_1$	::	*State-name*$_1$
70	*Stop-node*$_1$	::	( )
71	*Return-node*$_1$	::	( )

Task

72	*Task-node*$_1$	::	*Assignment-statement*$_1$ |
			Informal-text$_1$

Create

73	*Create-request-node*$_1$	::	*Process-identifier*$_1$
			[*Expression*$_1$]*

Procedure call

74	*Call-node*$_1$	::	*Procedure-identifier*$_1$
			[*Expression*$_1$]*
75	*Procedure-identifier*$_1$	=	*Identifier*$_1$

Output

76	*Output-node*$_1$	::	*Signal-identifier*$_1$
			[*Expression*$_1$]*
			[*Signal-destination*$_1$]
			Direct-via$_1$
77	*Signal-destination*$_1$	=	*Expression*$_1$ | *Process-identifier*$_1$
78	*Direct-via*$_1$	=	(*Signal-route-identifier*$_1$ | *Channel-identifier*$_1$)-**set**

Decision

79	*Decision-node*$_1$	::	*Decision-question*$_1$
			Decision-answer$_1$-**set**
			[*Else-answer*$_1$]
80	*Decision-question*$_1$	=	*Expression*$_1$ |
			Informal-text$_1$
81	*Decision-answer*$_1$	::	(*Range-condition*$_1$ | *Informal-text*$_1$)
			Transition$_1$
82	*Else-answer*$_1$	::	*Transition*$_1$

Timer

83	*Timer-definition*$_1$	::	*Timer-name*$_1$
			Sort-reference-identifier$_1$*
84	*Timer-name*$_1$	=	*Name*$_1$
85	*Set-node*$_1$	::	*Time-expression*$_1$
			Timer-identifier$_1$
			Expression$_1$*

86	*Reset-node*$_1$	::	*Timer-identifier*$_1$
			Expression$_1$*
87	*Timer-identifier*$_1$	=	*Identifier*$_1$
88	*Time-expression*$_1$	=	*Expression*$_1$

1.2 Structural Decomposition Concepts in SDL

Block partitioning

1	*Block-substructure-definition*$_1$	::	*Block-substructure-name*$_1$
			Sub-block-definition$_1$-**set**
			Channel-connection$_1$-**set**
			Channel-definition$_1$-**set**
			Signal-definition$_1$-**set**
			Data-type-definition$_1$
			Syn-type-definition$_1$-**set**
2	*Block-substructure-name*$_1$	=	*Name*$_1$
3	*Sub-block-definition*$_1$	=	*Block-definition*$_1$
4	*Channel-connection*$_1$	::	*Channel-identifier*$_1$-**set**
			Sub-channel-identifier$_1$-**set**
5	*Sub-channel-identifier*$_1$	=	*Channel-identifier*$_1$
6	*Channel-identifier*$_1$	=	*Identifier*$_1$

Refinement

7	*Signal-refinement*$_1$	::	*Subsignal-definition*$_1$-**set**
8	*Subsignal-definition*$_1$	::	[REVERSE] *Signal-definition*$_1$

1.3 Data in SDL

Data type definitions

1	*Data-type-definition*$_1$	::	*Sorts*$_1$ *Signature*$_1$-**set** *Equations*$_1$
2	*Sorts*$_1$	=	*Sort-name*$_1$-**set**
3	*Sort-name*$_1$	=	*Name*$_1$
4	*Equations*$_1$	=	*Equation*$_1$-**set**

Literals and parameterised operators

5	*Signature*$_1$	=	*Literal-signature*$_1$ | *Operator-signature*$_1$
6	*Literal-signature*$_1$	::	*Literal-operator-name*$_1$ *Result*$_1$
7	*Operator-signature*$_1$	::	*Operator-name*$_1$ *Argument-list*$_1$ *Result*$_1$
8	*Argument-list*$_1$	=	*Sort-reference-identifier*$_1$+
9	*Result*$_1$	=	*Sort-reference-identifier*$_1$
10	*Sort-reference-identifier*$_1$	=	*Sort-identifier*$_1$ | *Syntype-identifier*$_1$
11	*Literal-operator-name*$_1$	=	*Name*$_1$
12	*Operator-name*$_1$	=	*Name*$_1$
13	*Sort-identifier*$_1$	=	*Identifier*$_1$

Axioms

14	*Equation*$_1$	=	*Unquantified-equation*$_1$ | *Quantified-equations*$_1$ | *Conditional-equation*$_1$ | *Informal-text*$_1$
15	*Unquantified-equation*$_1$	::	*Term*$_1$ *Term*$_1$
16	*Quantified-equations*$_1$	::	*Value-name*$_1$-**set** *Sort-identifier*$_1$ *Equations*$_1$
17	*Value-name*$_1$	=	*Name*$_1$
18	*Term*$_1$	=	*Ground-term*$_1$ | *Composite-term*$_1$ | *Error-term*$_1$
19	*Composite-term*$_1$	::	*Value-identifier*$_1$ | *Operator-identifier*$_1$ *Term*$_1$+ | *Conditional-composite-term*$_1$
20	*Value-identifier*$_1$	=	*Identifier*$_1$
21	*Operator-identifier*$_1$	=	*Identifier*$_1$
22	*Ground-term*$_1$	::	*Literal-operator-identifier*$_1$ | *Operator-identifier*$_1$ *Ground-term*$_1$+ | *Conditional-ground-term*$_1$
23	*Literal-operator-identifier*$_1$	=	*Identifier*$_1$

Conditional equations

24	*Conditional-equation*$_1$	::	*Restriction*$_1$-**set** *Restricted-equation*$_1$
25	*Restriction*$_1$	=	*Unquantified-equation*$_1$
26	*Restricted-equation*$_1$	=	*Unquantified-equation*$_1$

Conditional terms

27	*Conditional-composite-term*$_1$	=	*Conditional-term*$_1$
28	*Conditional-ground-term*$_1$	=	*Conditional-term*$_1$
29	*Conditional- term*$_1$	::	*Condition*$_1$ *Consequence*$_1$ *Alternative*$_1$
30	*Condition*$_1$	=	*Term*$_1$
31	*Consequence*$_1$	=	*Term*$_1$
32	*Alternative*$_1$	=	*Term*$_1$

Errors

33	*Error-term*$_1$	::	( )

Syntypes

34	*Syntype-identifier*$_1$	=	*Identifier*$_1$
35	*Syn-type-definition*$_1$	::	*Syntype-name*$_1$ *Parent-sort-identifier*$_1$ *Range-condition*$_1$
36	*Syntype-name*$_1$	=	*Name*$_1$
37	*Parent-sort-identifier*$_1$	=	*Sort-identifier*$_1$
38	*Range-condition*$_1$	::	*Or-operator-identifier*$_1$ *Condition-item*$_1$-**set**
39	*Condition-item*$_1$	=	*Open-range*$_1$ | *Closed-range*$_1$
40	*Open-range*$_1$	::	*Operator-identifier*$_1$ *Ground-expression*$_1$
41	*Closed-range*$_1$	::	*And-operator-identifier*$_1$ *Open-range*$_1$ *Open-range*$_1$
42	*Or-operator-identifier*$_1$	=	*Identifier*$_1$
43	*And-operator-identifier*$_1$	=	*Identifier*$_1$

Expressions

44	*Expression*$_1$	=	*Ground-expression*$_1$ | *Active-expression*$_1$

Ground expressions

45	*Ground-expression*$_1$	::	*Ground-term*$_1$

Active expressions

46	*Active-expression*$_1$	=	*Variable-access*$_1$ | *Conditional-expression*$_1$ | *Operator-application*$_1$ | *Imperative-operator*$_1$ | *Error-term*$_1$

Variable access

47	*Variable-access*$_1$	=	*Variable-identifier*$_1$

Conditional expression

48	*Conditional-expression*$_1$	::	*Boolean-expression*$_1$ *Consequence-expression*$_1$ *Alternative-expression*$_1$
49	*Boolean-expression*$_1$	=	*Expression*$_1$
50	*Consequence-expression*$_1$	=	*Expression*$_1$
51	*Alternative-expression*$_1$	=	*Expression*$_1$

Operator application

52	*Operator-application*$_1$	::	*Operator-identifier*$_1$ *Expression*$_1$+

Assignment statement

53	*Assignment-statement*$_1$	::	*Variable-identifier*$_1$ *Expression*$_1$

Imperative operators

54	*Imperative-operator*$_1$	=	*Now-expression*$_1$ | *Pid-expression*$_1$ | *View-expression*$_1$ | *Timer-active-expression*$_1$ | *Anyvalue-expression*$_1$

Now expression

55	*Now-expression*$_1$	::	( )

PId expression

56	*Pid-expression*$_1$	=	*Self-expression*$_1$ | *Parent-expression*$_1$ | *Offspring-expression*$_1$ | *Sender-expression*$_1$
57	*Self-expression*$_1$	::	( )

58	*Parent-expression*$_1$	::	( )
59	*Offspring-expression*$_1$	::	( )
60	*Sender-expression*$_1$	::	( )

View expression

61	*View-expression*$_1$	::	*View-identifier*$_1$ [*Expression*$_1$]
62	*View-identifier*$_1$	=	*Identifier*$_1$

Timer active expression

63	*Timer-active-expression*$_1$	::	*Timer-identifier*$_1$ *Expression*$_1$*

Anyvalue expression

64	*Anyvalue-expression*$_1$	::	*Sort-reference-identifier*$_1$

2 Domains for the Meta-Process Communication

2.1 SDL Process Creation and Stopping

This section defines the communication domains used when creating and stopping SDL process instances. This includes the creation and stopping of instances in the environment of the SDL system.

1	*Create-Instance-Request*	::	*Process-identifier*$_1$ *Value-List Parent-Value*
2	*Parent-Value*	=	*Pid-Value*
3	*Create-Instance-Request*1	::	*Value-List Parent-Value Offspring-Value*
4	*Offspring-Value*	=	*Pid-Value*
5	*Body-Created*	::	*II*(*sdl-process*)
6	*Inport-Created*	::	*II*(*input-port*)
7	*Instance-Created*	::	( )
8	*Create-Instance-Answer*1	::	*Exceed*
9	*Exceed*	=	*Bool*
10	*Create-Instance-Answer*	::	*Offspring-Value*

The domains above are used when an SDL process or service instance executes a create request node. The interpreting *sdl-process* or *sdl-service* outputs *Create-Instance-Request* to *system* which, when having performed the necessary communication with other meta-processes, responds by outputting *Create-Instance-Answer* to the *sdl-process/sdl-service.* The data carried by *Create-Instance-Request* are the identifier of the SDL process of which an instance is to be started, the list of actual parameters, and the Pid value of the SDL process instance performing the create request. The data carried by *Create-Instance-Answer* is the Pid value of the created SDL process instance (which is Null if a new instance could not be created due to maximum number of instances).

When *system* receives a *Create-Instance-Request*, it outputs *Create-Instance-Request*$_1$ to the *process-set-admin* corresponding to the *Process-identifier*$_1$. When having performed the necessary actions, the *process-set-admin* respond by outputting *Create-Instance-Answer*$_1$ to *system.* The data carried by *Create-Instance-Request*$_1$ is the list of actual parameters, the Pid value of the creating SDL process, and the Pid value of the new SDL process. The data carried by *Create-Instance-Answer*$_1$ is a Boolean value indicating whether or not a new SDL process could be created without violating the maximum number of instances of the corresponding SDL process set.

When a *process-set-admin* receives a *Create-Instance-Request*$_1$, it creates an *input-port* and an *sdl-process* (unless this would lead to violation of the maximum number of instances). Immediately after creation of these two meta-processes, the *process-set-admin* outputs *Body-Created* to the *input-port*, and *Inport-Created* to the *sdl-process.* The data carried by *Body-Created* and *Inport-Created* are the meta-pid (II) values of the s*dl-process* instance resp. the *input-port* instance such that these two meta-process instances can address communication to each other.

When the *sdl-process* has performed its necessary setup, it outputs *Instance-Created* to the *process-set-admin.*

If the created SDL process is decomposed into services, the interpreting *sdl-process* creates one *sdl-service* instance for each SDL service. Each individual *sdl-service* outputs *Instance-Created* to the *sdl-process* when having performed the necessary setup.

11	*Stop-Instance*	::	( )
12	*Stop-Input-Port*	::	( )

The domains above are used when an SDL process or service instance executes a **stop** node. If the SDL process is not decomposed into services, the interpreting *sdl-process* outputs *Stop-Instance* to its managing *process-set-admin* when interpreting a **stop** node. When having input *Stop-Instance* the *process-set-admin* outputs *Stop-Input-Port* to the corresponding *input-port*.

If the SDL process is decomposed into services, then when an *sdl-service* interprets a **stop** node, it outputs *Stop-Instance* to the managing *sdl-process.* When the last service instance has stopped, the *sdl-process* outputs *Stop-Instance* to its *process-set-admin.*

13	*Environment-admin*	::	*II*(*process-set-admin*)
14	*Create-Pid*	::	( )
15	*Pid-Created*	::	*Pid-Value*

Since as few assumptions as possible should be made about the environment, a special scheme for creation of instances in the environment has been defined. It is considerably simpler than the scheme for creation of processes within the system. It is assumed that all SDL process instances in the environment are managed by the same *process-set-admin* instance in the environment, and that the meta-pid (II) value of this is communicated to *system* carried by *Environment-admin* during system start up.

When an SDL process instance is to be created in the environment, the environment outputs *Create-Pid* to *system.* The *system* responds by outputting a new, unique SDL Pid value to the environment carried by *Pid-Created.*

The main purpose of this scheme is to justify the administration within the system of Pid values in the environment.

2.2 SDL Signal Communication

This section defines the communication domains used for handling of SDL signal communication.

1	*Send-Signal*	::	*Signal-identifier*$_1$ *Value-List Sender-Id* *Sender-Value* [*Receiver*] *Direct-via*$_1$
2	*Sender-Id*	=	ENVIRONMENT | *Process-identifier*$_1$ | *Service-identifier*$_1$
3	*Sender-Value*	=	*Pid-Value*
4	*Receiver*	=	*Receiver-Value* | *Process-identifier*$_1$
5	*Receiver-Value*	=	*Pid-Value*
6	*Queue-Signal*	::	*Signal-identifier*$_1$ *Value-List* *Sender-Value II* (*process-set-admin*) [*Receiver-Value*]
7	*Signal-Delivered*	::	*Signal-identifier*$_1$ *Value-List* *Sender-Value* [*Receiver-Value*]
8	*Queue-Signal*1	::	*Signal-identifier*$_1$ *Value-List Sender-Value*

The domains above are used when communicating signals between SDL process instances. When an SDL process or service interprets an **output** node, the interpreting *sdl-process* or *sdl-service* outputs *Send-Signal* to *system.* The data carried are the identifier of the SDL signal being sent, the list of optional values carried by the signal, the SDL process or service identifier of the sender (or ENVIRONMENT if it is the environment of the system which sends the signal), the SDL Pid value of the sender, the optional SDL Pid value/process identifier of the receiver, and the optional **via** set of channel/signal route identifiers.

When *system* receives a *Send-Signal* it chooses a communication path taking into consideration the destination and routing information contained in *Send-Signal.* If the chosen path does not contain any delaying channels, *system* outputs *Signal-Delivered* to the *process-set-admin* instance corresponding to the destination endpoint of the communication path. If the chosen path contains delaying channels, *system* outputs *Queue-Signal* to the *path* instance corresponding to (the delaying part) of the path. The data carried by both *Signal-Delivered* and *Queue-Signal* are the SDL signal identifier, the list of optional values carried by the signal, the SDL sender Pid value, and the optional receiver Pid value. In addition, *Queue-Signal* carries the meta-pid value of the *process-set-admin* instance at the destination endpoint of the chosen communication path such that the *path* instance can deliver the signal to the correct *process-set-admin* instance.

In case a signal was sent via a delaying path, the corresponding *path* instance delivers after some delay the signal by outputting *Signal-Delivered* to the receiving *process-set-admin.*

When a *process-set-admin* receives a *Signal-Delivered,* it will either deliver the signal to an *input-port* or discard it, taking into consideration the destination information contained in *Signal-Delivered* and the current set of SDL process instances alive. If the signal is equipped with an explicit destination Pid value which denotes a living instance in the SDL process instance set, the signal is delivered to the *input-port* of this instance; if the signal is *not* equipped with an explicit destination Pid value, and there is at least one living instance in the SDL process instance set, an *input-port* belonging to one of the SDL process instances is chosen nondeterministically; in all other cases no *input-port* is chosen, i.e. the signal is discarded. In case a possible receiver is found, the *process-set-admin* outputs *Queue-Signal*$_1$ to its *input-port.* The data values carried are the signal identifier, the value list and the sender.

9	*Next-Signal*	::	*Signal-identifier*$_1$-**set** *Spontaneous-Present*
10	*Spontaneous-Present*	=	*Bool*
11	*Input-Signal*	::	*Signal-identifier*$_1$ *Value-List Sender-Value*
12	*Spontaneous-Signal*	::	( )

These domains are used for signal communication between the input port and body of an SDL process instance. When the SDL process instance enters a state, the interpreting *sdl-process* outputs *Next-Signal* to its *input-port*. The data values carried are the save signal set of the state, and a boolean value indicating whether or not the state contains spontaneous transitions. The *input-port* responds by outputting *Input-Signal* or *Spontaneous-Signal* to the *sdl-process.*

If the SDL process is decomposed into services, the interpreting *sdl-service* instances communicate these domains with the *input-port* via their managing *sdl-process.* When an SDL service instance enters a state, the interpreting *sdl-service* outputs *Next-Signal* to its *sdl-process* which then passes on this output to *input-port.* When the *input-port* has responded with *Input-Signal* or *Spontaneous-Signal* to the *sdl-process,* the *sdl-process* passes on this output to an *sdl-service* which needs not be the one which most recently output *Next-Signal*. The *sdl-service* instance is chosen by having the *sdl-process* maintain a table with information about which SDL services have which signals in their valid input signal set.

2.3 SDL Service Handling

This section defines the communication domains used for SDL service handling.

1	*Execute-Start*	::	( )

This domain is used by *sdl-process* for coordinating the execution of service start transitions when the interpreted SDL process is decomposed into services. When the *sdl-process* has started all *sdl-service* instances, it outputs *Execute-Start* to each *sdl-service* instance one by one and waits for each *sdl-service* to complete its start transition before outputting *Execute-Start* to the next *sdl-service*.

No other special domains for service execution coordination are necessary as some of the other domains already defined can easily be used for this purpose.

2.4 SDL Timer Handling

This section defines the communication domains used for SDL timer handling.

1	*Set-Timer*	::	*Timer-identifier*$_1$ *Arglist Timeout-Value*
2	*Timeout-Value*	=	*Value*
3	*Reset-Timer*	::	*Timer-identifier*$_1$ *Arglist*
4	*Active-Request*	::	*Timer-identifier*$_1$ *Arglist*
5	*Active-Answer*	::	*Bool*

When an SDL process instance executes a **set** node, the interpreting *sdl-process* outputs *Set-Timer* to its *input-port* which then starts a timer instance. The data carried are the SDL timer identifier, a list of timer argument values, and the expiration time for this timer instance setting.

When an SDL process instance executes a **reset** node the interpreting *sdl-process* outputs *Reset-Timer* to its *input-port* which then stops the timer instance. The data carried are the SDL timer identifier and a list of timer argument values.

When an SDL process evaluates a timer **active** expression, the interpreting *sdl-process* outputs *Active-Request* to its *input-port* which then responds by outputting *Active-Answer* to the *sdl-process.* The boolean data value carried indicates whether or not the timer instance is active.

If the SDL process is decomposed into services, the interpreting *sdl-services* communicate these domains with the *input-port* via their managing *sdl-process* which in this case simply acts as a relay.

2.5 Time Handling

This section defines the communication domains used for time handling.

1	*Time-Request*	::	( )
2	*Time-Answer*	::	*Value*
3	*Time*	::	( )

When an SDL process or service instance evaluates a **now** expression, the interpreting *sdl-process* or *sdl-service* outputs *Time-Request* to *timer*. The *timer* responds by outputting *Time-Answer* which carries the value of the current time.

Each *input-port* instance continuously tests on the expiration time of its timer instances. For that purpose it needs the current time from the *timer*. This communication is the same as between *sdl-process/sdl-service* and *timer*.

2.6 Revealed Variable Handling

This section defines the communication domains used for revealed variable handling.

1	*Reveal*	::	*Variable-identifier*$_1$ *Sort-reference-identifier*$_1$ *Pid-Value* (*Value* | UNDEFINED)
2	*View-Request*	::	*View-identifier*$_1$ *Sort-reference-identifier*$_1$ [*Pid-Value*]
3	*View-Answer*	::	(*Value* | UNDEFINED)
4	*Die*	::	*Pid-Value* (*Process-identifier*$_1$ | *Service-identifier*$_1$)

When an SDL process or service instance updates a revealed variable, the interpreting *sdl-process* or *sdl-service* outputs *Reveal* to *view*. The data carried are the identifier and sort/syntype of the revealed variable, the Pid value of the SDL process instance directly or indirectly (i.e. from a service) revealing the variable, and the new value of the variable.

When an SDL process or service evaluates a **view** expression, the interpreting *sdl-process* or *sdl-service* outputs *View-Request* to *view* which then responds with *View-Answer*. The data carried by *View-Request* is the identifier and sort/syntype of the viewed variable, and the optional SDL Pid value of the intended revealer. The data carried by *View-Answer is* the value of the viewed variable.

When an SDL process or service instance stops, the interpreting *sdl-process* or *sdl-service* outputs *Die* to *view* which then removes from its internal map of revealed variables all revealed variables of the owning process or service instance. The data carried are the SDL Pid value of the stopping process instance or the process instance owning the stopping service, and the SDL identifier of the stopping process or service instance.

2.7 Common Domains

This section defines some common domains which are either used in the communication domains above or to address the communication between meta-processes.

1	*Value-List*	=	(*Value* | UNDEFINED)*
2	*Arglist*	=	*Value**
3	*Pid-Value*	=	*Value*
4	*Value*	=	*Ground-term*$_1$

A *Value-List* is the result of evaluating a list of actual parameters to a **create** or **output** node. If a given actual parameter is absent, the corresponding “value” is UNDEFINED.

An *Arglist* is the result of evaluating an argument list in a **set** node, **reset** node or **active** expression.

A Value is an SDL ground term. For each equivalence class of the data sorts in the SDL system, the same ground term will always represent this equivalence class during interpretation of the SDL system. A *Pid-Value* is a *Value.*

5	*Admin-processor*	= *II* (*process-set-admin*) | *II* (*sdl-process*)
6	*Input-processor*	= *II* (*input-port*) | *II* (*sdl-process*)
7	*Body-processor*	= *II* (*sdl-process*) | *II* (*sdl-service*)

The domains *Admin-processor* and *Input-processor* are used when interpreting the nodes of a behaviour graph. If the graph is interpreted by an *sdl-process*, the administrating processor is a *process-set-admin*, and the SDL signal input is obtained from an *input-port.* If the graph is interpreted by an *sdl-service,* the administrating processor is an *sdl-process*, and the SDL signal input is also obtained from *sdl-process.*

A behaviour graph is interpreted by a *Body-processor* which is either an *sdl-process* or *sdl-service* instance.

3 Domains for the Entity Information

Entity-dict contains information of all SDL identifiers referred to in the SDL processes and services, i.e. whenever a process or service needs information of an identifier *Entity-dict* is used. Initially, it is deduced from AS_1. Each SDL process and service has its own instance of *Entity-dict.*

1

$$
\begin{aligned}
\textit{Entity-dict} \quad = {} & (\textit{Qualifier}_1\ \mathsf{TYPE}) \xrightarrow[m]{} \textit{TypeDD} \cup \\
& (\textit{Identifier}_1\ \mathsf{SORT}) \xrightarrow[m]{} (\textit{SortDD} \mid \textit{SyntypeDD}) \cup \\
& (\textit{Identifier}_1\ \mathsf{VALUE}) \xrightarrow[m]{} (\textit{OperatorDD} \mid \textit{VarDD} \mid \textit{ViewDD}) \cup \\
& (\textit{Identifier}_1\ \mathsf{SIGNAL}) \xrightarrow[m]{} \textit{SignalDD} \cup \\
& (\textit{Identifier}_1\ \mathsf{PROCESS}) \xrightarrow[m]{} \textit{ProcessDD} \cup \\
& (\textit{Identifier}_1\ \mathsf{SERVICE}) \xrightarrow[m]{} \textit{ServiceDD} \cup \\
& (\textit{Identifier}_1\ \mathsf{PROCEDURE}) \xrightarrow[m]{} \textit{ProcedureDD} \cup \\
& \mathsf{ENVIRONMENT} \xrightarrow[m]{} \textit{Reachabilities} \cup \\
& \mathsf{EXPIREDF} \xrightarrow[m]{} \textit{Is-expiredF} \cup \\
& \mathsf{SYSTEMLEVEL} \xrightarrow[m]{} \textit{Qualifier}_1 \cup \\
& \mathsf{PIDSORT} \xrightarrow[m]{} \textit{Sort-identifier}_1 \cup \\
& \mathsf{NULLVALUE} \xrightarrow[m]{} \textit{Value} \cup \\
& \mathsf{TRUEVALUE} \xrightarrow[m]{} \textit{Value} \cup \\
& \mathsf{FALSEVALUE} \xrightarrow[m]{} \textit{Value} \cup \\
& \mathsf{SCOPEUNIT} \xrightarrow[m]{} \textit{Qualifier}_1 \cup \\
& \mathsf{SELF} \xrightarrow[m]{} \textit{Pid-Value} \cup \\
& \mathsf{PARENT} \xrightarrow[m]{} \textit{Pid-Value} \cup \\
& \mathsf{OFFSPRING} \xrightarrow[m]{} \mathbf{ref}\ \textit{Pid-Value} \cup \\
& \mathsf{SENDER} \xrightarrow[m]{} \mathbf{ref}\ \textit{Pid-Value} \cup \\
& \mathsf{ADMIN} \xrightarrow[m]{} \textit{Admin-processor} \cup \\
& \mathsf{PORT} \xrightarrow[m]{} \textit{Input-processor}
\end{aligned}
$$

Entity-dict is a map from pairs of identifiers (*Identifier*$_1$s) or qualifiers (*Qualifier*$_1$s) and their associated entity kind into descriptors. An entity kind is either TYPE, SORT, VALUE, SIGNAL, PROCESS, SERVICE or PROCEDURE. As an AS_1 data type definition (*Data-type-definition*$_1$) has no identifier on its own, the *Qualifer*$_1$ denoting the scope unit where it is defined is used instead.

In addition, *Entity-dict* contains information of how signals from the environment of the system can be routed. ENVIRONMENT is explained below.

A descriptor is either a descriptor of a type, a sort, a syntype, a literal or operator, a variable, a signal, a process, a service, or a procedure. Note that some of the entities of SDL identifiers are excluded (e.g. channels and blocks).

Furthermore, *Entity-dict* contains some extra objects which have to be known by the underlying system and/or the sdl processes or services. Those objects are accessed via some *Quot* values:

ENVIRONMENT	When applied on *Entity-dict* the result is the routing information (for SDL signals) (*Reachabilities*) originating from the environment.
EXPIREDF	When applied on *Entity-dict* the result is a function used by *input-port* processor instances for timer handling.
SYSTEMLEVEL	When applied on *Entity-dict* the result is the AS_1 qualifier denoting the system level.
PIDSORT	When applied on *Entity-dict* the result is the AS_1 identifier of the Pid sort.
NULLVALUE	When applied on *Entity-dict* the result is an AS_1 ground term representing the Pid value Null.
TRUEVALUE	When applied on *Entity-dict* the result is an AS_1 ground term representing the Boolean value True.

FALSEVALUE	When applied on *Entity-dict* the result is an AS_1 ground term representing the Boolean value False.
SCOPEUNIT	When applied on *Entity-dict* the result is the qualifier denoting the current scopeunit.
SELF	When applied on *Entity-dict* the result is the SDL Pid value of either the SDL process using the *Entity-dict* or the owning SDL process of the service using the *Entity-dict.*
PARENT	When applied on *Entity-dict* the result is the SDL Pid value of either the parent of the SDL process using the *Entity-dict* or the owning SDL process of the service using the *Entity-dict.*
OFFSPRING	When applied on *Entity-dict* the result is a pointer to a Meta-IV variable holding the SDL Pid value of the most recent offspring of either the SDL process using the *Entity-dict* or the owning SDL process of the service using the *Entity-dict.*
SENDER	When applied on *Entity-dict* the result is a pointer to a Meta-IV variable holding the SDL Pid value of the most recent sender of either the SDL process using the *Entity-dict* or the owning SDL process of the service using the *Entity-dict.*
ADMIN	When applied on *Entity-dict* the result is the II value of the Meta-IV process (i.e. *process-set-admin*) administrating the process set to which the SDL process belongs, or the *sdl-process* which manages the SDL service.
PORT	When applied on *Entity-dict* the result is the II value of the *input-port* of the SDL process, or the *sdl-process* which "looks like" an *input-port* from the SDL service.

3.1 The Type Descriptor

1	*TypeDD*	::	*Term-reduce-map Sortmap* $Equations_1$
2	*Term-reduce-map*	=	*Term-class* $\underset{m}{\rightarrow}$ *Term*
3	*Term-class*	=	*Term*-**set**
4	*Term*	=	$Ground\text{-}term_1$ | $Error\text{-}term_1$
5	*Sortmap*	=	$Sort\text{-}identifier_1$ $\underset{m}{\rightarrow}$ *Term-class*-**set**

The first field (*Term-reduce-map*) contains all equivalence classes (*Term-class*) of all sorts visible in the scopeunit enclosing the data type definition. The *Term-reduce-map* maps each equivalence class to a canonical term (*Term*) which has been chosen to represent that term. If an equivalence class contains the error term, *Term-reduce-map* always maps it to the error term; else if the equivalence class represents a value which must be recognizable by the Meta-IV formulas when interpreting an SDL system (e.g. the Boolean values True and False), *Term-reduce-map* maps it to the same value as given by *Entity-dict* (entries TRUEVALUE and FALSEVALUE for the Boolean values); otherwise an arbitrary term is chosen when building the *Entity-dict*, and thereafter the equivalence class will always be represented by that term.

The second field is a map (*Sortmap*) of all $Sort\text{-}identifier_1$s visible in the scopeunit enclosing the data type definition into the set of equivalence classes existing for the sort. The sort map is only used while building the *Entity-dict* for an SDL system.

The third field is the equations ($Equations_1$) from which the equivalence classes are derived.

3.2 The Sort Descriptor

1	*SortDD*	::	( )
2	*SyntypeDD*	::	*Parent-sort-identifier*$_1$ *Range-condition*$_1$

SortDD and *SyntypeDD* are descriptors of newtypes and syntypes respectively. A newtype descriptor contains no information but is there any way in order to have all used sort identifiers in the *Entity-dict.*

A syntype descriptor also contains the identifier of the parent newtype and an AS_1 range condition.

3.3 The Operator and Literal Descriptor

1	*OperatorDD*	::	*Argument-list Result*
2	*Argument-list*	=	*Sort-reference-identifier*$_1$*
3	*Result*	=	*Sort-reference-identifier*$_1$

OperatorDD is a descriptor of an operator or a literal. It contains the list of sorts or syntypes of the arguments and the sort or syntype of the result.

3.4 The Variable Descriptor

1	*VarDD*	::	*Variable-identifier*$_1$ *Sort-reference-identifier*$_1$ [*Ground-expression*$_1$] [REVEALED] **ref** *Stg*

VarDD is a descriptor of a variable. It contains the variable identifier, the sort or syntype identifier, the initialization expression, if any, the REVEALED attribute and a reference to a process-, service- or procedure-local storage. Each time a procedure is invoked, *Entity-dict* is overwritten with the descriptors representing the formal parameters and local declarations. For an **in**/**out** formal parameter, the descriptor contains the *Variable-identifier*$_1$ of the associated actual parameter and a reference to the storage where the value of the actual parameter can be found, i.e. because SDL allows recursive procedures, there may exist several storages containing variables with the same *Variable-identifier*$_1$, one for each recursive call.

3.5 The View Descriptor

1	*ViewDD*	::	*Sort-reference-identifier*$_1$

ViewDD is a descriptor of a view definition. It contains the sort or syntype identifier of the view.

3.6 The Signal Descriptor

1	*SignalDD*	::	*Sort-reference-identifier*$_1$* [REVERSE]

SignalDD is a descriptor of a signal. It contains the list of sort or syntype identifiers attached to the signal and, in case it is a subsignal, whether or not it goes in the reverse direction of its parent signal.

3.7 The Process Descriptor

1	*ProcessDD*	::	*ParameterDD* Initial Maximum* [*Process-graph*$_1$] *Reachabilities*
2	*ParameterDD*	=	*Variable-identifier*$_1$
3	*Initial*	=	*Intg*
4	*Maximum*	=	[*Intg*]
5	*Reachabilities*	=	*Reachability*-**set**
6	*Reachability*	=	*Reachability-endp Signal-identifier*$_1$-**set** *Path*
7	*Reachability-endp*	=	ENVIRONMENT | *Process-identifier*$_1$ | *Service-identifier*$_1$
8	*Path*	=	*Path-element**
9	*Path-element*	=	*Path-identifier Path-direction* [NODELAY]
10	*Path-identifier*	=	*Identifier*$_1$
11	*Path-direction*	=	FORWARD | REVERSE

ProcessDD is a descriptor of a process. It contains the parameter list (*ParameterDD*), the number of process instances created at system start-up time (*Initial*), the maximum number of allowed processes (*Maximum*), the process graph, and *Reachabilities.* A formal parameter descriptor is the *Variable-identifier*$_1$ of the parameter. A *Reachability* defines a destination *Reachability-endp* (*Process-identifier*$_1$, *Service-identifier*$_1$ or the ENVIRONMENT) which may be reached from the process in the sending of a signal in *Signal-identifier*$_1$-**set** using a certain *Path.* The *Path is* identified by a list of path elements (*Path-element*) each of which contains a channel or signal route identifier (*Path-identifier*), a path direction (*Path-direction*) which is used to identify each direction in a bidirectional channel/signal route, and an indication of whether the path element has a delay or not (a channel may or may not have a delay, a signal route never has a delay). *Path* is empty in the cases where *Process-identifier*$_1$ (or *Service-identifier*$_1$, see below under the description of service descriptors) is both the sender and the receiver.

3.8 The Service Descriptor

1	*ServiceDD*	::	*Service-graph*$_1$ *Input-signal-set Reachabilities*
2	*Input-signal-set*	=	*Signal-identifier*$_1$-**set**

ServiceDD is a descriptor of a service. It contains the service graph, the set of valid input signals *Signal-identifier*$_1$-**set** of the service, and the *Reachabilities* of the service.

3.9 The Procedure Descriptor

1	*ProcedureDD*	::	*FormparmDD* Procedure-graph*$_1$
2	*FormparmDD*	=	*InparmDD* | *InoutparmDD*
3	*InparmDD*	::	*Variable-identifier*$_1$
4	*InoutparmDD*	::	*Variable-identifier*$_1$

ProcedureDD is a descriptor of a procedure. It contains a list of formal parameter descriptors and the procedure graph. A formal parameter is either an **in** parameter or an **in**/**out** parameter and it contains the *Variable-identifier*$_1$.

4 The Underlying System

4.1 System Processor

This processor is the entry point for interpretation of an SDL system. All other processes are started (directly or indirectly) from this process. It is started from *definition-of-SDL*, defined in Annex F.2: Static Semantics.

The processor internally uses the following domains:

```
Process-set-admin-map   = (ENVIRONMENT | Process-identifier1) m→
                          II (process-set-admin)
Path-map                = Path m→ II (path)
Inst-map                = Pid-Value m→
                          (ENVIRONMENT | Process-identifier1)
```

The domain *Process-set-admin-map* maps the identifier of each SDL process instance set to the II value of the *process-set-admin* instance which interprets it. Furthermore, as all SDL process instances running in the environment are assumed to be managed by the same *process-set-admin* instance running in the meta-environment, the map also contains a map from ENVIRONMENT to this instance. The domain is used for routing of SDL signals and creating instance requests.

The domain *Path-map* maps each delaying path to its corresponding instance of the *path* processor. A delaying path is a list of (delaying) channel paths traversed by a signal instance when an **output** node has been interpreted. It is necessary to distinguish possible delaying paths since preservation of signal order is only guaranteed when following the same sequence of delaying channels.

The domain *Inst-map* maps each Pid value of an alive or dead SDL process instance to the identifier of the process set to which it belongs (or to ENVIRONMENT for each SDL process instance alive or dead in the environment). That is, entries are never removed from the map. The domain is used for routing of SDL signals and for keeping track of which SDL Pid values have already been used such that new, unique Pid values can be generated whenever needed.

4.1.1 The Processor

system **processor** (*as*$_1$*tree*, *subset*, *auxinf*) $\triangleq$ (4.1.1.1)

```
(dcl adminmap type Process-set-admin-map;
 dcl pathmap type Path-map;
 dcl instmap := [] type Inst-map;
 (let (timeinf, terminf, expiredf, delayf) = auxinf in
  let dict = extract-dict(as1tree, subset, expiredf, terminf) in
  start view();
  start timer(timeinf)(dict);
  start-process-set-admins(delayf)(dict);
  start-paths(delayf)(dict);
  start-initial-processes(dict);
  handle-inputs(dict)))
```

type: *System-definition*$_1$ *Block-identifier*$_1$-**set** *Auxiliary-information* ⇒

Objective Interpret the SDL system.

Parameters

as$_1$*tree* The AS_1 definition of the system.

subset The consistent subset selected.

auxinf Contains the following (see line 4):

timeinf Information required by the *timer* processor. It contains a function which updates the current **now** on each tick in the *timer* processor and the start value of the system time. The domain is defined in Annex F.2 and it is further described in the definition of the *timer* processor.

terminf A closure containing the AS_1 identifier of the Pid sort and three AS_1 ground terms chosen to represent each of the following values: The Pid value Null and the Boolean values True and False.

expiredf A function delivering **true** if a given timer has expired.

delayf A function delivering a *Bool* value at random. Used in the *path* processor for modelling delay on channels, and in the *input-port* processor for modelling unstability of SDL states containing spontaneous transitions.

Algorithm

Line 1-3 Declare the variables needed by the *system* processor. The purpose of the variables has already been explained below the domain definitions above.

Line 5 Build the *Entity-dict* corresponding to the given SDL system, the selected subset and the necessary parts of *Auxiliary-information.*

Line 6 Start one instance of the *view* processor.

Line 7 Start one instance of the *timer* processor with actual parameters for the handling of **now** (further explained in the definition of *timer*).

Line 8 Start one instance of the *process-set-admin* processor for each process definition present in the SDL system (or rather in the selected consistent subset). The actual parameter *delayf* will be used for handling of spontaneous transitions.

Line 9 Start one instance of the *path* processor for each sequence of delaying channel paths which can be traversed by at least one SDL signal type.

Line 10 Perform the system start up creation of SDL process instances.

Line 11 Handle all further meta-communication to and from the *system.*

start-process-set-admins(*delayf*)(*dict*) ≜ (4.1.1.2)

((**input** **mk**-*Environment-admin*(*envadmin*) **from** …
  ⇒ adminmap := [ENVIRONMENT ↦ *envadmin*]);
(**def** *adminmap-delta* : [*prid* ↦ **start** *process-set-admin*(*prid, delayf*)(*dict*) |
    (*prid*, PROCESS) ∈ **dom** *dict*];
adminmap := **c** adminmap + *adminmap-delta*))

type: *DelayF* → *Entity-dict* ⇒

Objective Start one *process-set-admin* processor instance for each process definition present in (the selected consistent subset of) the SDL system.
Enter information about the started processor instances in adminmap.

Parameters

delayf A function delivering a *Bool* value at random. Used to model the unstability of SDL states containing spontaneous transitions.

Algorithm

Line 1-2 Obtain the II value of the *process-set-admin* instance which is assumed to run in the meta-environment. Enter this instance in adminmap.

Line 3-5 Start one *process-set-admin* instance for each process definition in the SDL system and compute the adminmap contribution from this (lines 3-4). Update adminmap with this contribution (line 5).

start-paths(*delayf*)(*dict*) ≜ (4.1.1.3)

```
1   (let reaches = dict(ENVIRONMENT) ∪
2        union {s-Reachabilities(dict((prid, PROCESS))) |
3                  (prid, PROCESS) ∈ dom dict} ∪
4        union {s-Reachabilities(dict((servid, SERVICE))) |
5                  (servid, SERVICE) ∈ dom dict} in
6    let delaypaths = {delaying-path(path) | (, , path) ∈ reaches} in
7    pathmap := [delaypath ↦ start path(delayf) | delaypath ∈ delaypaths \ {⟨⟩}])
```

type: *DelayF* → *Entity-dict* ⇒

Objective Start one *path* instance for each sequence of delaying channel paths which can be traversed by at least one SDL signal type. Enter information about the started processor instances in pathmap.

Parameters

delayf A function delivering a *Bool* value at random. Used to model the delay on channels.

Algorithm

Line 1-5 Extract all existing *Reachabilities* in the SDL system. The total *Reachability* set consists of all *Reachabilities* originating from the system environment (line 1), all *Reachabilities* originating from SDL process instance sets not partitioned into services (line 2-3) and all *Reachabilities* originating from services (line 4-5).

Line 6 For each *Reachability* in the SDL system, extract the sequence of delaying channel paths contained in *Path.*

Line 7 Start one *path* instance for each (non-empty) sequence of delaying channel paths which connects two leaf blocks (in the selected consistent subset) or one leaf block and the SDL system environment. Enter these instances in pathmap.

start-initial-processes(*dict*) ≜ (4.1.1.4)

```
for all (prid, PROCESS) ∈ dom dict do
(let mk-ProcessDD(parmddl, initno, , ,) = dict((prid, PROCESS)) in
 let vl = ⟨UNDEFINED | 1 ≤ i ≤ len parmddl⟩,
      parent = dict(NULLVALUE) in
 for i = 1 to initno do
 handle-create-instance-request(prid, vl, parent, nil)(dict))
```

type: *Entity-dict* ⇒

Objective Perform the system start up creation of SDL process instances.

Algorithm

Line 1	For each process instance set in the SDL system do the following:
Line 2	Obtain information about the formal parameters and initial number of instances for the process instance from the *dict.*
Line 3-4	All actual parameters to a process instance which is created at system start up are “undefined” (line 3). The **parent** value for such an instance is Null (line 4).
Line 5-6	Create *initno* instances of the process instance set. The fourth actual parameter in line 6 is **nil** to indicate that there is no SDL process or service instance waiting for response about the process instance creation.

handle-inputs(*dict*) $\triangleq$ (4.1.1.5)

```
1   cycle {input mk-Create-Instance-Request(prid, vl, parent) from parbody
2             ⇒  handle-create-instance-request(prid, vl, parent, parbody)(dict),
3          input mk-Create-Pid( ) from se
4             ⇒  handle-create-in-env(se)(dict),
5          input mk-Send-Signal(sid, vl, seid, se, re, via) from ...
6             ⇒  handle-send-signal(sid, vl, seid, se, re, via)(dict)}
```

type: *Entity-dict* ⇒

Objective Handle all meta-communication of *system* after initializations.

Algorithm

Line 1	Start a loop forever. In each iteration of that loop one of the mentioned inputs will be elaborated (on a non-deterministic basis). The handling of each input is described in a specific handling function.

handle-create-instance-request(*prid*, *vl*, *parent*, *parbody*)(*dict*) $\triangleq$ (4.1.1.6)

```
(def offspring : getpid(dom c instmap)(dict);
 def offspradmin : c adminmap(prid);
 output mk-Create-Instance-Request1(vl, parent, offspring) to offspradmin;
 input mk-Create-Instance-Answer1(exceed) from offspradmin
    ⇒   (if ¬exceed then
             instmap := c instmap + [offspring ↦ prid]
           else
             I;
          if parbody ≠ nil then
             (let offspring′ = if ¬exceed then offspring else dict(NULLVALUE) in
             output mk-Create-Instance-Answer(offspring′) to parbody)
           else
             I))
```

type: *Process-identifier*$_1$ *Value-List Pid-Value* [*Body-processor*] → *Entity-dict* ⇒

Objective Handle creation of SDL process instances.

Parameters

prid	The SDL process identifier of the process instance to be started.
vl	The list of actual parameter values.
parent	The SDL Pid value of the creating process instance.

parbody The II value of the processor which interprets the creating SDL process or service instance. This parameter is **nil** if the function is called during system initialization.

Algorithm

Line 1 Create a unique SDL Pid value.

Line 2 Get the II value of the *process-set-admin* instance for the SDL process to be created.

Line 3 Output a create instance request to the *process-set-admin.*

Line 4 Wait for response from the *process-set-admin.* The input parameter *exceed* indicates whether or not a new SDL process instance could be created due to the maximum number of instances.

Line 5-8 If a new SDL process instance was created, the instance map (instmap) is updated with the new instance.

Line 9-13 If the create was caused by a **create** node, then send a response to the creating SDL process or service instance as follows:

Line 10 If the create request succeeded, then the **offspring** value should be the one generated in line 1, otherwise it should be the Pid value Null.

Line 11 Send this **offspring** value to the creator.

handle-create-in-env(*se*)(*dict*) ≜ (4.1.1.7)

```
1    (def offspring : getpid(dom c instmap)(dict);
2     instmap := c instmap + [offspring ↦ ENVIRONMENT];
3     output mk-Pid-Created(offspring) to se)
```

type: *II* → *Entity-dict* ⇒

Objective Handle the creation of SDL Pid values in the environment. Update maps within the system and return the Pid value to the environment. The communication is not exactly like the one in handling of **create** nodes within the system. However, one cannot suppose the environment to contain **create** nodes (!). The general idea is to make as few assumptions about the environment as possible while still having a consistent model.

Parameters

se The II value of “the sender”.

Algorithm

Line 1 Create a unique Pid value.

Line 2 Update the map of living SDL process instances with the new instance.

Line 3 Return the Pid value to the environment.

handle-send-signal(*sid*, *vl*, *seid*, *se*, *re*, *via*)(*dict*) ≜ (4.1.1.8)

```
1    (let reaches =
2        (seid = ENVIRONMENT
3          → dict(ENVIRONMENT),
4        (seid, PROCESS) ∈ dom dict
5          → s-Reachabilities(dict((seid, PROCESS))),
6        (seid, SERVICE) ∈ dom dict
7          → s-Reachabilities(dict((seid, SERVICE)))) in
8    let reaches′ = restrict-to-signal(reaches, sid) in
9    let reaches″ =
10       if via = { }
11         then reaches′
12         else restrict-to-via(reaches′, via) in
13   def (reaches‴, re′) : (re = nil
14                           → (reaches″, nil),
15                         (re, PROCESS) ∈ dom dict
16                           → (restrict-to-destprcs-or-env(reaches″, re)(dict), nil),
17                         ⊤ → (restrict-to-destpid(reaches″, re, c instmap)(dict), re));
18   if reaches‴ ≠ { } then
19     (let (reidorenv, , path) ∈ reaches‴ in
20      let delaypath = delaying-path(path) in
21      def readmin : c adminmap(process-or-env(reidorenv)(dict));
22      if delaypath = 〈〉 then
23        output mk-Signal-Delivered(sid, vl, se, re′) to readmin
24       else
25        (def path′ : c pathmap(delaypath);
26         output mk-Queue-Signal(sid, vl, se, readmin, re′) to path′ ))
27    else
28     I)
```

type: *Signal-identifier*$_1$ *Value-List Sender-Id Sender-Value* [*Receiver*] *Direct-via*$_1$ → *Entity-dict* ⇒

Objective Routing of SDL signals.

Parameters

sid	Signal being sent.
vl	List of values carried by the signal.
seid	The SDL identifier of the process or service sending the signal (or ENVIRONMENT if the signal is sent from the environment).
se	The SDL Pid value of the sender.
re	The optional SDL Pid value or process identifier of the (intended) receiver of the signal from the **to** clause.
via	Set of signal route and channel identifiers from the optional **via** clause. If the **via** clause was absent, this set is empty.

Algorithm

Line 1-7	Obtain the set of *Reachabilities* originating from the sender. The sender can either be the environment (line 1-3), an instance of a process which is not decomposed into services (line 4-5), or a service instance (line 6-7). The remaining part of the function consecutively restricts the *Reachabilities* of the sender (until line 17).
Line 8	Restrict to those *Reachabilities* which may convey the signal.
Line 9-12	Restrict to the signal routes and channels mentioned in the **via** clause, if any.

Line 13-17 Restrict to the Pid value or process identifier of the **to** clause, if any, and get a resulting optional receiver Pid value as follows:

If the **to** clause was absent, no further restrictions are made on the *Reachabilities*, and the optional receiver Pid value is **nil** (line 13-14).

If the **to** clause contained a process identifier, the *Reachabilities* are restricted to this process identifier, and the optional receiver Pid value is **nil** (line 15-16).

If the **to** clause contained a Pid expression, the *Reachabilities* are restricted to the process set which contains the destination process instance, and the receiver Pid value is this Pid value (line 17). Note that if the Pid expression evaluated to Null, to a Pid value of a not yet existing process instance, or to a Pid value of an instance which cannot be reached via the given *Reachabilities*, the remaining *Reachability* set will be empty.

Line 18,28 If the remaining *Reachability* set is empty, the signal is discarded.

Line 19 Select an arbitrary *Reachability* from the remaining *Reachability* set and decompose it into a destination endpoint and a communication path.

Line 20 Obtain the delaying part of the chosen communication path.

Line 21 Obtain the II value of the *process-set-admin* instance which should receive the signal. If the destination endpoint of the *Reachability* is a service, then use the identifier of its enclosing process definition as key to the adminmap.

Line 22-23 If the chosen communication path contains no delaying channel paths, the signal is sent directly to the receiving *process-set-admin* instance.

Line 25-26 Obtain the II value of the *path* instance which should convey the signal, and output the signal to this instance.

4.1.2 Auxiliary Functions

restrict-to-signal(*reaches*, *sid*) $\triangleq$ (4.1.2.1)

1 {(, *sigset*,) ∈ *reaches* | *sid* ∈ *sigset*}

type: *Reachability*-**set** *Signal-identifier*$_1$ → *Reachability*-**set**

Objective Restrict a set of *Reachabilities* to the set of *Reachabilities* which are able to convey a given signal.

Parameters

reaches The original set of *Reachabilities.*

sid The identifier of the signal.

Result The restricted set of *Reachabilities.*

Algorithm

Line 1 Select those *Reachabilities* whose signal set contain the given signal.

restrict-to-via(*reaches*, *via*) $\triangleq$ (4.1.2.2)

1 {(, , *path*) ∈ *reaches* | *is-in-via*(*path*, *via*)}

type: *Reachability*-**set** *Direct-via*$_1$ → *Reachability*-**set**

Objective Restrict a set of *Reachabilities* to the set of *Reachabilities* which are mentioned in a given **via** set.

Parameters

reaches The original set of *Reachabilities.*

via The **via** set.

Result The restricted set of *Reachabilities.*

Algorithm

Line 1 Select those *Reachabilities* which contain a signal route or channel mentioned in the **via** set.

is-in-via(*path*, *via*) $\triangleq$ (4.1.2.3)

1 (**let** *srchids* = {*id* | (*id*, ,) ∈ **elems** *path*} **in**
2 *srchids* ∩ *via* ≠ {})

type: *Path Direct-via*$_1$ → *Bool*

Objective Test whether a given communication path contains a signal route or channel identifier mentioned in a given **via** set.

Parameters

path The communication path.

via The **via** set.

Result **true** if the path is mentioned, **false** otherwise.

Algorithm

Line 1 Extract the set of signal route and channel identifiers in the communication path.

Line 2 The communication path is mentioned in the **via** clause if the intersection of the **via** set and the set of signal routes/channels is non-empty.

restrict-to-destprcs-or-env(*reaches*, *repridorenv*)(*dict*) ≜ (4.1.2.4)

```
1    {(reachendp, ,) ∈ reaches | process-or-env(reachendp)(dict) = repridorenv}
```

type: *Reachability*-**set** (ENVIRONMENT | *Process-identifier*$_1$) → *Entity-dict* → *Reachability*-**set**

Objective Restrict a set of *Reachabilities* to the set of *Reachabilities* which lead to a given SDL process instance set.

Parameters

reaches The original set of *Reachabilities.*

repridorenv The SDL identifier of the process instance set, or ENVIRONMENT if the desired destination endpoint is the system environment.

Result The restricted set of *Reachabilities.*

Algorithm

Line 1 Select those *Reachabilities* which have *repridorenv* as destination endpoint. If a *Reachability* has a service as destination endpoint, the identifier of the enclosing process definition is used as key for the selection.

restrict-to-destpid(*reaches*, *re*, *instmap*)(*dict*) ≜ (4.1.2.5)

```
1    if re ∈ dom instmap then
2       (let repridorenv = instmap(re) in
3         restrict-to-destprcs-or-env(reaches, repridorenv)(dict))
4      else
5       {}
```

type: *Reachability*-**set** *Receiver-Value Inst-map* → *Entity-dict* → *Reachability*-**set**

Objective Restrict a set of *Reachabilities* to the set of *Reachabilities* which lead to an SDL process instance with a given Pid value.

Parameters

reaches The original set of *Reachabilities.*

re The Pid value of the desired receiver.

instmap The map of SDL Pid values of living process instances.

Result The restricted set of reachabilities.

Algorithm

Line 1,6 If the Pid value is Null or denotes a not yet created SDL process instance, the resulting set of *Reachabilities* is empty.

Line 2 Obtain the identifier of the SDL process instance set to which the given process instance belongs (or ENVIRONMENT if the process instance belongs to the environment).

Line 3 Restrict the set of *Reachabilities* to the obtained process instance set.

delaying-path(*path*) $\triangleq$ (4.1.2.6)

```
1    ⟨path[i] | 1 ≤ i ≤ len path ∧ (let(, , nodelay) = path[i] in
2                                   nodelay = nil)⟩
```

type: *Path* → *Path*

Objective Extract the delaying part of a communication path.

Parameters

path The original communication path.

Result The delaying part of the communication path.

Algorithm

Line 1 Delete all signal route and channel paths which have no delay.

process-or-env(*reachendp*)(*dict*) $\triangleq$ (4.1.2.7)

```
1    (reachendp = ENVIRONMENT
2        → ENVIRONMENT,
3     (reachendp, PROCESS) ∈ dom dict
4        → reachendp,
5     (reachendp, SERVICE) ∈ dom dict
6        → enclosing-scopeunit(reachendp))
```

type: *Reachability-endp* → *Entity-dict* → (ENVIRONMENT | $Process\text{-}identifier_1$)

Objective If a reachability endpoint denotes a service, then convert it to the identifier of the enclosing SDL process.

Parameters

reachendp The reachability endpoint.

Result The converted reachability endpoint.

Algorithm

Line 1-4 If the reachability endpoint denotes the environment or an SDL process definition, then return it unchanged.

Line 5-6 If the reachability endpoint denotes a service definition, then return the identifier of the enclosing SDL process definition.

getpid(*pidsinuse*)(*dict*) $\triangleq$ (4.1.2.8)

```
1    (let newpid ∈ values-of-sort(dict(PIDSORT))(dict)
2          be s.t. newpid ≠ dict(NULLVALUE) ∧ newpid ∉ pidsinuse in
3     newpid)
```

type: *Pid-Value*-**set** → *Entity-dict* → *Pid-Value*

Objective Extract a *Pid-Value* not used yet. The Unique! operator defined for the Pid sort in Z.100 ensures that there exists an infinite number of *Pid-Values.* I.e. the values for the Pid sort are Null, Unique!(Null), Unique!(Unique!(Null)), etc. The set of Pid values is found in *dict.*

Parameters

pidsinuse The set of Pid values which are already in use.

Result An unused *Pid-Value.*

Algorithm

Line 1 Take a Pid value from the set of possible Pid values such that the Pid value is neither Null nor has been used before.

Line 3 Return the Pid value.

4.2 View Processor

This processor uses the internal domain *Reveal-map* which maps triples of SDL Pid values, variable identifiers and variable sorts/syntypes to revealed values. For variables revealed by service instances, the Pid value is that of the enclosing process instance.

```
Reveal-map          = Reveal-map-key ⇒m (Value | UNDEFINED)
Reveal-map-key      = Pid-Value Variable-identifier1
                      Sort-reference-identifier1
```

4.2.1 The Processor

view **processor** ( ) ≜ (4.2.1.1)

```
 1  (dcl revealmap := [] type Reveal-map;
 2   trap exit( ) with error in
 3   (cycle {input mk-Reveal(varid, sortid, pid, value) from . . .
 4              ⇒  revealmap := c revealmap + [(pid, varid, sortid) ↦ value],
 5           input mk-View-Request(viewid, sortid, revealpid) from body
 6              ⇒  (def revealvars : revealed-variables(viewid, sortid, revealpid, c revealmap);
 7                   if revealvars ≠ {} then
 8                     (let revealvar ∈ revealvars in
 9                       output mk-View-Answer(c revealmap(revealvar)) to body)
10                     else
11                      exit("§5.4.4.4: No revealed variable access can be made")),
12           input mk-Die(pid, ownerid) from ...
13              ⇒  (def deadvars : {(pid′, varid,); ∈ dom c revealmap |
14                                  pid′ = pid ∧ enclosing-scopeunit(varid) = ownerid};
15                   revealmap := c revealmap \ deadvars)}))
```

type: ( ) ⇒

Objective Interpret the concept of **view** and **reveal**.

Algorithm

Line 1	Declare a (meta-)variable holding all revealed variable instances in the SDL system at any time.
Line 3	Handle the *Reveal* input.
Line 4	Update the map with the new value.
Line 5	Handle a **view** from an SDL process or service instance.
Line 6	Obtain the set of revealed variables matching the **view**.
Line 7-9	If there are any matching revealed variables then respond with the value of one of these.
Line 11	Define the error that no revealed variable access can be made.
Line 12	Handle the notice of a stopped SDL process or service instance.
Line 13-14	Obtain all revealed variables of the stopped SDL process or service instance.
Line 15	Delete all revealed variables of the stopped SDL process or service instance from the map.

revealed-variables(*viewid, sortid, revealpid, revealmap*) $\triangleq$ (4.2.1.2)

1 $\{(\mathit{pid}, \mathit{varid}, \mathit{sortid}') \in \mathbf{dom}\ \mathit{revealmap} \mid$
2 $(\mathit{revealpid} \neq \mathbf{nil} \supset \mathit{pid} = \mathit{revealpid}) \wedge$
3 $\mathit{enclosing\text{-}block}(\mathit{varid}) = \mathit{enclosing\text{-}block}(\mathit{viewid}) \wedge$
4 $\mathbf{s}\text{-}\mathit{Name}_1\,(\mathit{varid}) = \mathbf{s}\text{-}\mathit{Name}_1\,(\mathit{viewid}) \wedge$
5 $\mathit{sortid}' = \mathit{sortid}\}$

type: $\mathit{View\text{-}identifier}_1\ \mathit{Sort\text{-}reference\text{-}identifier}_1\ [\mathit{Pid\text{-}Value}]\ \mathit{Reveal\text{-}map}$
$\rightarrow \mathit{Reveal\text{-}map\text{-}key}$

Objective Obtain the set of revealed variables matching a specific **view** request.

Parameters

viewid The view identifier of the variable.

sortid The sort or syntype of the viewed variable.

revealpid The optional Pid value resulting from the optional Pid expression in the **view** expression.

revealmap The map of currently living revealed variables.

Result The set of revealed variables matching the **view** request.

Algorithm

Line 2 If a Pid expression was present in the **view** expression, the matching revealed variables are all revealed by the process instance (or contained service instances) having the Pid value resulting from the Pid expression. Otherwise any process instance revealing the variable can be used.

Line 3-5 The revealed variables must be in the same block as the view definition and have the same name and sort/syntype.

4.3 Timer Processor

This processor has been introduced to interpret the concept of global time in SDL. It results in a very simple communication with an external *tick* processor.

timer **processor** (*timeinf*)(*dict*) ≜ (4.3.1)

```
1  (let (timef, startt) = timeinf in
2   dcl time-now := startt type Value;
3   cycle {input mk-Time() from tick
4             ⇒   time-now  := timef (c time-now),
5          input mk-Time-Request() from p
6             ⇒   (def time-now′ : reduce-term(c time-now, dict(SYSTEMLEVEL))(dict);
7                  output mk-Time-Answer(time-now′) to p)})
```

type: *Time-information* → *Entity-dict* ⇒

Objective Interpret the timer-handling in underlying system.

Parameters The object *timeinf* contains two components (line 1) generated in Annex F.2:

timef A function being called on each "tick" from the environment. The *timef* function thus encapsulates two problems: interpretation of "+" for the Time sort and the resolution of time values within the system (i.e. what is the increment in **now** for each "tick").

startt The initial value of **now**.

Algorithm

Line 2 Let time-now denote the (only one) global time of the system. By using a model which includes the start time for interpretation (*startt*) and the updating (the function *timef*) it is hoped to give a correct description of SDL's time-concept.

Line 4 Update the time.

Line 6-7 Return **now**. In line 6 the ground term stored in time-now is reduced to the ground term which has been chosen to represent this time value in the rest of the system.

4.4 Informal Tick Processor

tick **processor** () ≜ (4.4.1)

```
cycle {(output mk-Time() to timer;
        /* models informally the interval between consecutive ticks */)}
```

type: () ⇒

4.5 Path Processor

This processor uses the internal domain *Path-queue* to represent the internal queue of signals. Each *Path-queue-item* contains the SDL identifier of the signal, the list of SDL data values carried by the signal, the sender Pid value, the II value of the receiving *process-set-admin* instance, and an optional receiver Pid value.

```
Path-queue            = Path-queue-item*
Path-queue-item       = Signal-identifier₁ Value-List Sender-Value
                          Receiver-Admin [Receiver-Value]
Receiver-Admin        = II(process-set-admin)
```

4.5.1 The Processor

path **processor** (*delayf*) ≜ (4.5.1.1)

```
1   (dcl pqueue := ⟨⟩ type Path-queue;
2    cycle {input mk-Queue-Signal(sid, vl, se, readmin, re) from system
3              ⇒    (pqueue := c pqueue ⁀ ⟨(sid, vl, se, readmin, re)⟩),
4           (if c pqueue ≠ ⟨⟩ ∧ delayf ( ) then
5               (def (sid, vl, se, readmin, re) : hd c pqueue
6                output mk-Signal-Delivered(sid, vl, se, re) to readmin;
7                pqueue := tl c pqueue)
8               else
9                I)})
```

type: *DelayF* ⇒

Objective Interpret the potential delay in a communication path. An instance exists for each sequence of delaying channel paths originating from some SDL process or service or from the system environment.

Parameters

delayf A function delivering a *Bool* value at random. Used for modelling delay on channels

Algorithm

Line 3 Insertion of a signal into the queue of the path.

Line 4 This clause models the non-deterministic delay on the path. The delivery of a signal may only take place if pqueue is non-empty and *delayf* yields **true**. Otherwise a new iteration of the **cycle** is initiated.

Line 5-6 Deliver the first signal in the queue to the *process-set-admin* instance.

Line 7 Remove the output signal from the queue.

4.6 Process Set Administrating Processor

This processor is the entry point for interpretation of an SDL process instance set and manages directly or indirectly all other processor instances concerned with interpreting the given SDL process instance set.

process-set-admin **processor** (*prid, delayf*)(*dict*) $\triangleq$ (4.6.1)

1 (**dcl** pidno := 0 **type** N_0;
2 **dcl** instancemap := [] **type** *II*(*sdl-process*) $\underset{m}{\rightarrow}$ *Pid-Value*;
3 **dcl** queuemap := [] **type** *Pid-Value* $\underset{m}{\rightarrow}$ *II*(*input-port*);
4 **cycle** {**input mk**-*Create-Instance-Request*1(*vl, par, offspr*) **from** *system*
5 $\Rightarrow$ *handle-create-instance-request*1(*prid, vl, par, offspr, delayf*)(*dict*),
6 **input mk**-*Stop-Instance*( ) **from** *body*
7 $\Rightarrow$ *handle-stop-instance*(*body*),
8 **input mk**-*Signal-Delivered*(*sid, vl, se, re*) **from** ...
9 $\Rightarrow$ *handle-signal-delivered*(*sid, vl, se, re*)})

type: *Process-identifier*$_1$ *DelayF* $\rightarrow$ *Entity-dict* $\Rightarrow$

Objective Interpret an SDL process instance set.

Parameters

prid The identifier of the SDL process instance set.

delayf A function delivering a *Bool* value at random. The function is used to model the unstability of SDL states containing spontaneous transitions.

Algorithm

Line 1 Declare a variable for keeping track of the number of living process instances in the SDL process instance set. The variable is used for ensuring that the maximum number of instances is never exceeded.

Line 2 Declare a variable mapping the II value of each *sdl-process* instance to the Pid value of the SDL process instance that it interprets. The variable is only used when an SDL process instance stops.

Line 3 Declare a variable mapping the Pid value of each SDL process instance to the II value of the *input-port* processor which models its input port queue.

Line 4-9 Handle all meta-communication of *process-set-admin* after initialisation. The handling of each input is described in a specific handling function.

*handle-create-instance-request*1(*prid*, *vl*, *parent*, *offspring*, *delayf*)(*dict*) $\triangleq$ (4.6.2)

```
 1  (let omax = s-Maximum(dict((prid, PROCESS))) in
 2   def exceed : omax ≠ nil ∧ c pidno = omax;
 3   if ¬exceed then
 4      (def inport : start input-port(prid, offspring, delayf, self)(dict);
 5       def body : start sdl-process(prid, vl, parent, offspring)(dict + [ADMIN ↦ self]);
 6       output mk-Body-Created(body) to inport;
 7       output mk-Inport-Created(inport) to body;
 8       input mk-Instance-Created( ) from body
 9          ⇒   (pidno := c pidno + 1;
10               instancemap := c instancemap + [body ↦ offspring];
11               queuemap := c queuemap + [offspring ↦ inport]))
12     else
13      I;
14   output mk-Create-Instance-Answer1(exceed) to system)
```

type: $+$*Process-identifier*$_1$ *Value-List Parent-Value Offspring-Value*-**set** *DelayF* → *Entity-dict* ⇒

Objective Handle incoming create instance request.

Parameters

prid	The identifier of the SDL process instance set.
vl	The list of actual parameter values.
parent	The Pid value of the creating process instance (Null for a system start up create request).
offspring	The Pid value of the new process instance if it can be created.
delayf	The function for modelling the unstability of SDL states containing spontaneous transitions.

Algorithm

Line 1-2	Obtain the optional maximum number of instances for this process instance set and check whether creation of a new instance would violate this maximum. If *omax* is **nil** the number of instances is unbounded.
Line 3-13	If the maximum number of instances already exists, then do not create a new instance.
Line 4-5	Start one *input-port* instance and one *sdl-process* instance. The *dict* is updated with the II value of the *process-set-admin* before it is transferred to the *sdl-process* instance.
Line 6-7	Send the II value of the *sdl-process* to the *input-port* and vice versa such that they are able to address each other when they want to communicate with each other.
Line 8	Wait for an initialization acknowledgement from the *sdl-process.*
Line 9-11	Update the process set administrating variables with the new SDL process instance.
Line 14	Tell the *system* whether or not a new SDL process instance could be created.

handle-stop-instance(*body*) $\triangleq$ (4.6.3)

```
(def pid : c instancemap(body);
 pidno := c pidno – 1;
 instancemap := c instancemap \ {body};
 queuemap : = c queuemap \ {pid})
```

type: *II*(*sdl-process*) ⇒

Objective Handle the stopping of an SDL process instance belonging to the process instance set.

Parameters

body The II value of the *sdl-process* which interprets the body of the stopping SDL process instance.

Algorithm

Line 1 Get the SDL Pid value of the stopping process instance.

Line 2-4 Remove the process instance from the process set administrating variables.

handle-signal-delivered(*sid, vl, se, re*) ≜ (4.6.4)

```
1   (def re′ : get-receiver(re, dom c queuemap);
2    if re′ ≠ nil then
3       output mk-Queue-Signal1(sid, vl, se) to c queuemap(re′)
4     else
5      I)
```

type: *Signal-identifier*$_1$ *Value-List Sender-Value* [*Receiver-Value*] ⇒

Objective Find a receiver of an incoming signal or discard it.

Parameters

sid The signal identifier.

vl The list of data values carried with the signal.

se The sender Pid value.

re The optional receiver Pid value.

Algorithm

Line 1 Obtain a possible receiver, if any, of the signal.

Line 2-3 If there is a possible receiver, then deliver the signal to the input port of the chosen receiver.

Line 5 Otherwise discard the signal.

get-receiver(*re, pids*) ≜ (4.6.5)

```
if re = nil then
   (if pids ≠ {} then
       (let re′ ∈ pids in
        re′)
     else
       nil)
  else
   (if re ∈ pids then
       re
     else
       nil)
```

type: [*Receiver-Value*] *Pid-Value*-**set** → [*Pid-Value*]

Objective Obtain the Pid value of a possible receiver, if any, of a signal which conveys an optional receiver Pid value.

Parameters

re The optional receiver Pid value conveyed with the signal.

pids The set of Pid values of process instances currently alive.

Result If a possible receiver exists, then its Pid value, else **nil**.

Algorithm

Line 1 Two cases are distinguished: The case where the signal does not carry an explicit receiver Pid value is handled by lines 2-6; the case where the signal carries an explicit receiver Pid value is handled by lines 8-11.

Line 2-3 If the process instance set currently contains any living instances, then select an arbitrary one as receiver.

Line 6 Otherwise indicate that no receiver can be found.

Line 8-9 If the intended receiver of the signal is alive, then return its Pid value.

Line 11 Otherwise indicate that the intended receiver is not alive.

4.7 Input-Port Processor

This processor implements the unbounded buffers of SDL process instances, and timers. Furthermore, for model-technical reasons (the need to avoid deadlock between an *input-port* instance and an *sdl-process* instance belonging together) the *input-port* processor also handles the concept of spontaneous transitions.

The *input-port* processor uses internally some auxiliary domains.

1	*Inport-queue*	= *Inport-queue-item**
2	*Inport-queue-item*	= $\mathit{Signal\text{-}identifier}_1$ *Value-List Sender-Value*

The domain *Inport-queue* is used to represent the internal queue of signals in the input port. Each *Inport-queue-item* contains the SDL identifier of the signal, the list of SDL data values carried by the signal, and the sender Pid value.

The domain *Inport-queue* is handled by functions which have been defined separately from the input port processor functions.

3	*Timer-table*	= $(\mathit{Timer\text{-}identifier}_1\ \mathit{Arglist}) \underset{m}{\rightarrow} [\mathit{Timeout\text{-}Value}]$

The domain *Timer-table* is used to keep track of active timers. Each ($\mathit{Timer\text{-}identifier}_{1,}$ *Arglist*) pair represents one active timer instance and is mapped to the expiration time value of the timer instance ([*Timeout-Value*]). The [*Timeout-Value*] becomes **nil**, when the timer instance expires and the corresponding signal is placed in the input port queue. The timer instance is removed from the timer table when the corresponding signal is consumed by the SDL process body.

4.7.1 The Processor

input-port **processor** (*prid, selfpid, delayf, admin*)(*dict*) ≙ (4.7.1.1)

```
1   (dcl queue := empty-inport-queue( ) type Inport-queue;
2    dcl timers := [] type Timer-table;
3    dcl waiting := false type Bool;
4    dcl saveset type Signal-identifier1 -set
5    dcl spont type Spontaneous-Present;
6    (input mk-Body-Created(body) from admin
7        ⇒   (let mk-Identifier1(qual, nm) = prid,
8                  level = qual ⌢ ⟨mk-Process-qualifier1(nm)⟩ in
9             let dict′ = dict + [SCOPEUNIT ↦ level,
10                                SELF      ↦ selfpid] in
11            cycle {input mk-Stop-Input-Port( ) from body
12                       ⇒   stop,
13                   input mk-Queue-Signal1(sid, vl, se) from admin
14                       ⇒   handle-queue-signal1(sid, vl, se, delayf, body),
15                   input mk-Next-Signal(saveset′, spont′) from body
16                       ⇒   handle-next-signal(saveset′, spont′, delayf, body),
17                   (handle-spontaneous-transition(delayf, body)),
18                   input mk-Set-Timer(tid, al, tv) from body
19                       ⇒   handle-set-timer(tid, al, tv, delayf, body)(dict′),
20                   input mk-Reset-Timer(tid, al) from body
21                       ⇒   handle-reset-timer(tid, al),
22                   input mk-Active-Request(tid, al) from body
23                       ⇒   handle-active-request(tid, al, body),
24                   (output mk-Time-Request( ) to timer;
25                    handle-time-request(delayf, body)(dict′))}))))
```

type: *Process-identifier*$_1$ *Pid-Value DelayF II*(*process-set-admin*) → *Entity-dict* ⇒

Objective Model the input port of an SDL process instance. One *input-port* instance exists for each SDL process instance.

Parameters

prid The SDL identifier of the process instance set, to which the SDL process instance owning the input port belongs.

selfpid The Pid value of the SDL process instance owning the input port.

delayf *Bool* function used to model the unstability of SDL states containing spontaneous transitions.

admin The II value of the *process-set-admin* instance administrating this *input-port* instance.

Algorithm

Line 1 Let queue denote the unbounded buffer of the SDL process instance and initialise it to the empty queue.

Line 2 Let timers denote the table of active timer instances and initialise it to the empty table.

Line 3 Let waiting denote whether *sdl-process* is waiting for reply after a request for *Next-Signal* which could not be answered immediately because queue was empty, or because all signals present in the queue had to be saved. Initially the *sdl-process* does not wait for a reply.

Line 4 Let saveset denote the **save** signal set when the *sdl-process* is ready to receive the next signal. The contents of this variable only makes sense when the variable waiting is **true**.

Line 5 Let spont denote whether the SDL state in which the *sdl-process* is waiting contains spontaneous transitions. The contents of this variable only makes sense when the variable waiting is **true**.

Line 6 Obtain the II value of the *sdl-process* instance with which this *input-port* instance should communicate.

Line 7-10 Construct the qualifier denoting the process instance set and insert this qualifier in the *Entity-dict* together with the Pid value of the SDL process instance owning the input port.

Line 11 Is the entry of the main cycle of *input-port.*

Line 15 Note: this input cannot always be answered immediately. The reason for introducing the variables waiting, saveset and spont is the **save** construct. If a pure queue structure, then an input guard could be used to exclude communication of *Next-Signal* in case of an empty queue.

Line 17 This **cycle** branch models the unstability of SDL states containing spontaneous transitions. As this branch is not guarded, it can be taken at any time independently of the other branches. See *handle-spontaneous-transition* for further details on the handling of spontaneous transitions.

Line 24 Include one output in this scheme. It is the repeated request for the current time from the *timer*.

*handle-queue-signal*1(*sid, vl, se, delayf, body*) $\triangleq$ (4.7.1.2)

```
1   (queue := add-signal-inport-queue( c queue, (sid, vl, se));
2    if c waiting then
3       try-to-make-transition(delayf, body)
4      else
5       I)
```

type: *Signal-identifier*$_1$ *Value-List Pid-Value DelayF II*(*sdl-process*) $\Rightarrow$

Objective A signal has been received from some SDL process instance, or a timer instance has expired. Put the signal in the input port queue. Thereafter, if the SDL process body is waiting in a state, then make it perform a transition if possible.

Parameters

sid Signal to be inserted.

vl Its optional list of values.

se Sender Pid value of the signal.

delayf The *Bool* function modelling the unstability of SDL states containing spontaneous transitions. Used if the SDL process body is waiting in a state and this state has spontaneous transitions.

body The II value of the *sdl-process* instance interpreting the SDL process body.

Algorithm

Line 1 Concatenate the signal to queue.

Line 2-3 If the SDL process body is waiting in a state then make it perform a transition if possible.

handle-next-signal(*saveset′*, *spont′*, *delayf*, *body*) ≙ (4.7.1.3)

```
1    (waiting := true;
2     saveset := saveset′;
3     spont := spont′;
4     try-to-make-transition(delayf, body))
```

type: *Signal-identifier*$_1$-**set** *Spontaneous-Present DelayF II*(*sdl-process*) ⇒

Objective The SDL process body has entered a new state. Make it perform a transition if possible.

Parameters

saveset' The **save** set for the state.

spont' An indication of whether the state contains spontaneous transitions.

delayf The *Bool* function modelling the unstability of SDL states containing spontaneous transitions.

body The II value of the *sdl-process* instance interpreting the SDL process body.

Algorithm

Line 1 Set waiting to **true** to indicate that the SDL process body is waiting in a state.

Line 2-3 Keep track of the **save** set of the SDL state, and whether it has spontaneous transitions.

Line 4 Make the SDL process body perform a transition if possible.

handle-spontaneous-transition(*delayf*, *body*) ≙ (4.7.1.4)

```
1    if c waiting ∧ c spont ∧ delayf() then
2       deliver-spontaneous-signal(body)
3      else
4       I
```

type: *DelayF II*(*sdl-process*) ⇒

Objective Model the unstability of SDL states containing spontaneous transitions.

Parameters

delayf The *Bool* function modelling the unstability.

body The II value of the *sdl-process* instance interpreting the SDL process body.

Algorithm

Line 1-2 If the SDL process body is waiting in a state, this state has spontaneous transitions, and the "unstability" function *delayf* yields **true**, then make the SDL process body perform a spontaneous transition.

try-to-make-transition(*delayf*, *body*) ≜ (4.7.1.5)

```
1    (def possible-actions : (if next-signal-inport-queue(c queue, c, saveset) ≠ nil then
2                                  {INPUTSIGNAL}
3                                 else
4                                  {}) ∪
5        (if c spont ∧ delayf( ) then {SPONTSIGNAL} else {});
6     if possible-actions ≠ {} then
7       (let action ∈ possible-actions in
8        cases action:
9         (INPUTSIGNAL   →  deliver-input-signal(body),
10         SPONTSIGNAL  →  deliver-spontaneous-signal(body)))
11      else
12       I)
```

type: *DelayF II*(*sdl-process*) ⇒

Objective The SDL process body is waiting in a state. Make it perform a transition if possible.

Parameters

delayf The *Bool* function modelling the unstability of SDL states containing spontaneous transitions.

body The II value of the *sdl-process* instance interpreting the SDL process body.

Algorithm

Line 1-5 Based on the contents of the variables queue, saveset, spont and the result of calling the "unstability" function *delayf*, compute a set of possible actions as follows:

Line 1 If the input port queue contains a signal which is not in the **save** set, the input port is able to deliver a signal to the SDL process body.

Line 5 If the SDL state has spontaneous transitions, and *delayf* yields **true**, the input port is able to (make the process body) initiate a spontaneous transition.

Line 6,12 If no actions are possible, the SDL process body keeps waiting.

Line 7 Select (one of) the possible action(s).

Line 9 If the chosen action is the delivery of a signal to the process body, then perform this action.

Line 10 If the chosen action is the initiation of a spontaneous transition, then perform this action.

deliver-input-signal(*body*) ≜ (4.7.1.6)

```
(def (sid, vl, se) : next-signal-inport-queue(c queue, c saveset);
 output mk-Input-Signal(sid, vl, se) to body;
 queue := remove-signal-inport-queue(c queue, c saveset);
 if (sid, vl) ∈ dom c timers then
    timers := c timers \ {(sid, vl)}
   else
    I;
 waiting := false)
```

type: *II*(*sdl-process*) ⇒

Objective The SDL process body is waiting in a state, and the input port has decided to deliver a signal to the body. Deliver the signal.

Parameters

body The II value of the *sdl-process* instance interpreting the SDL process body.

Algorithm

Line 1 Get the signal to be delivered, taking into account the **save** set.

Line 2 Deliver the signal.

Line 3 Remove the signal from the input port queue.

Line 4-5 If the signal is a timer signal, then remove it from the table of active timer instances.

Line 8 Indicate that the SDL process body is no longer waiting in a state.

deliver-spontaneous-signal(*body*) $\triangleq$ (4.7.1.7)

```
1    (output mk-Spontaneous-Signal( ) to body;
2     waiting := false)
```

type: *II*(*sdl-process*) ⇒

Objective The SDL process body is waiting in a state containing spontaneous transitions, and the input port has decided to initiate one of these. Do this.

Parameters

body The II value of the *sdl-process* instance interpreting the SDL process body.

Algorithm

Line 1 Make the process body perform a spontaneous transition.

Line 2 Indicate that the SDL process body is no longer waiting in a state.

handle-set-timer(*tid*, *al*, *tv*, *delayf*, *body*)(*dict*) $\triangleq$ (4.7.1.8)

```
(handle-reset-timer(tid, al);
 timers := c timers + [(tid, al) ↦ tv];
 output mk-Time-Request( ) to timer;
 handle-time-request(delayf, body)(dict))
```

type: *Timer-identifier*$_1$ *Arglist Timeout-Value DelayF II*(*sdl-process*) → *Entity-dict* ⇒

Objective Set a timer instance.

Parameters

tid Identifier of the timer.

al Argument value list of the timer.

tv Expiration time.

delayf The *Bool* function used to model the unstability of SDL states having spontaneous transitions. Although a timer can only be set when a transition is being performed this argument is necessary because other functions are called which really require this argument.

body The II value of the *sdl-process* instance interpreting the SDL process body.

Algorithm

Line 1 Reset the timer instance if it is already active.

Line 2 Update the map of active timers.

Line 3-4 Query the current time and make the timer instance expire immediately if its expiration time is less than or equal to **now**.

handle-reset-timer(*tid*, *al*) $\triangleq$ (4.7.1.9)

1 (timers := **c** timers \ {(*tid*, *al*)};
2 queue := *remove-timer-signal*(*tid*, *al*, **c** queue))

type: *Timer-identifier*$_1$ *Arglist* ⇒

Objective Reset a timer instance.

Parameters

tid Identifier of the timer.

al Argument value list of the timer.

Algorithm

Line 1 Remove the timer instance from the table of active timers.

Line 2 Remove the corresponding timer signal from the input port queue if it has been placed there.

handle-active-request(*tid*, *al*, *body*) $\triangleq$ (4.7.1.10)

1 (**def** *stat* : (*tid*, *al*) ∈ **dom c** timers;
2 **output mk**-*Active-Answer* (*stat*) **to** *body*)

type: *Timer-identifier*$_1$ *Arglist II*(*sdl-process*) ⇒

Objective Supply the answer to a timer **active** expression.

Parameters

tid Identifier of the timer.

al Argument value list of the timer.

body The II value of the *sdl-process* instance interpreting the SDL process body.

Algorithm

Line 1 Let *stat* denote **true** if the specified timer is active, otherwise **false**.

Line 2 Use this value as parameter in the output to *sdl-process.*

handle-time-request(*delayf*, *body*)(*dict*) ≙ (4.7.1.11)

```
1   (input mk-Time-Answer(t) from timer
2     ⇒   for all (tid, al) ∈  dom c timers do
3           (def expt : c timers((tid, al));
4            if expt ≠ nil ∧
5             reduce-term(dict(EXPIREDF)(expt, t), dict(SCOPEUNIT))(dict) = dict(TRUEVALUE) then
6              (timers := c timers + [(tid, al)  ↦ nil];
7               handle-queue-signal1(tid, al, dict(SELF), delayf, body))
8             else
9              I))
```

type: *DelayF II*(*sdl-process*) → *Entity-dict* ⇒

Objective Handle the comparison with the current time for all active, not yet expired timer instances. Place all expired timer instances in the input port queue.

Parameters

delayf The function modelling the unstability of SDL states containing spontaneous transitions.

body The II value of the *sdl-process* instance interpreting the SDL process body.

Algorithm

Line 1 Obtain the current time from the *timer* processor instance.

Line 2 Start the examination of all active timer instances. For each active timer instance do the following:

Line 3 Obtain the optional expiration time of the timer instance.

Line 4-5 If the timer instance has not already expired but should do this now, then do the following:

Line 6 Clear the expiration time for the timer instance.

Line 7 Enqueue the timer signal in the input port queue.

4.7.2 Input Port Queue Auxiliary Functions

empty-inport-queue() $\triangleq$ (4.7.2.1)

$\langle\rangle$

type: $\rightarrow$ *Inport-queue*

Objective Return an empty input port queue.

Result The empty queue.

add-signal-inport-queue(*q*, *qi*) $\triangleq$ (4.7.2.2)

$q \frown \langle qi \rangle$

type: *Inport-queue Inport-queue-item* $\rightarrow$ *Inport-queue*

Objective Enqueue a signal in an input port queue.

Parameters

q The old queue.

qi The new signal.

Result The queue including the new signal.

next-signal-inport-queue(*q*, *saveset*) $\triangleq$ (4.7.2.3)

1 $(q = \langle\rangle$
2 $\quad\rightarrow$ **nil**,
3 **s**-$\textit{Signal-identifier}_1$(**hd** *q*) $\notin$ *saveset*
4 $\quad\rightarrow$ **hd** *q*,
5 $\top \rightarrow$ *next-signal-inport-queue*(**tl** *q*, *saveset*))

type: *Inport-queue* $\textit{Signal-identifier}_1$-**set** $\rightarrow$ [*Inport-queue-item*]

Objective Obtain the next signal, which is not to be saved, from an input port queue.

Parameters

q The queue.

saveset The **save** set.

Result The next signal to be delivered from the queue, if any, otherwise **nil**.

Algorithm

Line 1-2 If the queue is empty, no signal can be obtained.

Line 3-4 If the first signal in the queue is not in the **save** set, then return this signal.

Line 5 Otherwise examine the rest of the queue.

remove-signal-inport-queue(*q*, *saveset*) $\triangleq$ (4.7.2.4)

1 $(\mathbf{s}\text{-}\mathit{Signal\text{-}identifier}_1(\mathbf{hd}\ q) \notin \mathit{saveset}$
2 $\quad \rightarrow \mathbf{tl}\ q,$
3 $\top \rightarrow \langle \mathbf{hd}\ q \rangle \frown \mathit{remove\text{-}signal\text{-}inport\text{-}queue}(\mathbf{tl}\ q, \mathit{saveset}))$

type: $\mathit{Inport\text{-}queue}\ \mathit{Signal\text{-}identifier}_1\text{-}\mathbf{set} \rightarrow \mathit{Inport\text{-}queue}$

Objective Remove the next signal from an input port queue, taking into consideration a **save** set. The function assumes that the queue contains signals not to be saved.

Parameters

q The old queue.

saveset The **save** set.

Result The queue where the signal has been removed.

Algorithm

Line 1-2 If the first signal in the queue is not in the **save** set, then remove this signal.

Line 3 Otherwise keep the first signal and remove a signal from the rest of the queue.

remove-timer-signal(*tid*, *al*, *q*) $\triangleq$ (4.7.2.5)

1 $\langle q[i] \mid 1 \leq i \leq \mathbf{len}\ q \wedge$
2 $\quad (\mathbf{let}\ (\mathit{sid}, \mathit{vl},) = q[i]\ \mathbf{in}$
3 $\quad \neg(\mathit{sid} = \mathit{tid} \wedge \mathit{vl} = \mathit{al}))\rangle$

type: $\mathit{Timer\text{-}identifier}_1\ \mathit{Arglist}\ \mathit{Inport\text{-}queue} \rightarrow \mathit{Inport\text{-}queue}$

Objective Remove a timer signal, if present, from an input port queue because the corresponding timer instance is being reset.

Parameters

tid The identifier of the timer.

al The argument value list of the timer instance.

q The queue.

Result The queue where the timer signal has been removed.

Algorithm

Line 1 Select all queue signals which fulfil the condition in line 2-3. Note that the nature of SDL and the formal model implies that at most one signal will be removed from the queue.

Line 2 Obtain the signal identifier and value list of each queue signal.

Line 3 Keep the queue signal if it does not denote the same timer instance as the one to be removed.

5 The SDL-Process and SDL-Service

This section describes how the META-IV processors *sdl-process* and *sdl-service* interpret (the body of) an SDL process instance resp. an SDL service instance.

Each *sdl-process* and *sdl-service* instance has a local storage, the type of which is given by:

```
Stg                         = Identifier1 →m (Value | UNDEFINED)
```

5.1 The sdl-process Processor

An instance of the *sdl-process* processor is created by its managing *process-set-admin* instance each time an SDL process instance is created. The *sdl-process* instance first performs the necessary initial setup and then interprets the process graph or service decomposition. When the SDL process instance ceases to exist the necessary cleanup is performed, and the *sdl-process* instance ceases to exist.

If the SDL process is *not* decomposed into services the interpreting *sdl-process* instance interprets its process graph. Otherwise it creates one instance of the *sdl-service* processor for each contained service and manages these *sdl-service* instances. In the latter case all meta-communication between *process-set-admin* and *input-port* on one side and *sdl-service* on the other goes through the *sdl-process* instance.

sdl-process **processor** (*prid*, *actparml*, *parentp*, *selfp*)(*dict*) ≜ (5.1.1)

```
(dcl sender := dict(NULLVALUE) type Pid-Value;
 dcl offspring := dict(NULLVALUE) type Pid-Value;
 dcl stg := [] type Stg;
 dcl servinstmap := [] type Π(sdl-service) →m Service- identifier1;
 dcl savemap := [] type Service- identifier1 →m Signal- identifier1-set;
 dcl spontmap := [] type Service- identifier1 →m Spontaneous-Present;
 (input mk-Inport-Created(inport) from dict(ADMIN)
       ⇒ (let mk-identifier1(qual, nm) = prid,
                 level = qual ⌒ ⟨mk-Process-qualifier1(nm)⟩ in
             def dict′ : dict + [SCOPEUNIT ↦ level,
                                  SELF       ↦ selfp,
                                  PARENT     ↦ parentp,
                                  OFFSPRING  ↦ offspring,
                                  SENDER     ↦ sender,
                                  PORT       ↦ inport];
             def dict″ : modify-process-vardds(prid, stg)(dict′);
             trap exit () with error in
             (create-process-vars(prid, actparml)(dict″);
             int-process-graph-or-service-decomp(prid)(dict″)))))
```

type: *Process-identifier*$_1$ *Value-List Pid-Value Pid-Value* → *Entity-dict* ⇒

Objective Interprets the body of an SDL process instance.

Parameters

prid	The SDL identifier of this process instance set.
actparml	The list of actual parameter values.
parentp	The SDL Pid value of the process instance that created this one.
selfp	The SDL Pid value of this process.

Algorithm

Line 1-2	Declare the variables sender and offspring, both initialized to the Pid value Null.
Line 3	Declare a variable stg which is to be the local storage of this SDL process instance and initialize it to be empty.
Line 4-6	These three variables are only used if the SDL process is decomposed into services. Their purpose is:
Line 4	The variable servinstmap contains at any time the set of living service instances owned by this SDL process instance. It maps each II value of an interpreting *sdl-service* instance to the SDL identifier of the service which it interprets. The map is used to direct SDL signals from the input port to the right service.
Line 5	The variable savemap contains a map from the SDL identifier of each living service instance to the save set of the state in which it is currently waiting.
Line 6	The variable spontmap contains a map which for each SDL identifier of a living service instance tells whether or not it is waiting in a state having spontaneous transitions.
Line 7	Obtain the II value of the *input-port* instance with which this *sdl-process* instance should communicate.
Line 8-9	Construct the qualifier for the SDL process set.
Line 10-15	Enter the following information into the *Entity-dict:* The current scope unit, the Pid value of the SDL process instance (**self**), the Pid value of its **parent**, a pointer to each of the meta-variables holding the Pid values of its **offspring** and **sender**, and the II value of the *input-port* instance used. The reason that the metavariables sender and offspring are accessed via pointers is that if the SDL process is decomposed into services these meta-variables will be shared between several *sdl-service* instances.
Line 16	For all variables declared in this SDL process (including process formal parameters), modify their descriptors such that they can be used for interpreting the process graph/service decomposition.
Line 17	Trap any **exit** with **error**.
Line 18	Create all process local SDL variables in the local storage.
Line 19	Interpret the process graph/service decomposition of the SDL process.

modify-process-vardds(*prid*, *stgref*)(*dict*) ≜ (5.1.2)

```
1   (let allvars = {varid | (varid, VALUE) ∈ dom dict ∧ enclosing-scopeunit(varid) = prid ∧
2                           is-VarDD(dict((varid, VALUE)))} in
3   dict + [(varid, VALUE) ↦ mk-VarDD(varid, sort, oinit, rev, stgref) |
4          varid, ∈ allvars ∧ mk-VarDD(, sort, oinit, rev,) = dict((varid, VALUE))])
```

type: *Process-identifier*$_1$ **ref** *Stg* → *Entity-dict* → *Entity-dict*

Objective Modify the *Entity-dict* descriptors for the variables (including process formal parameters) local to a given SDL process such that they can be used for interpretation of its process graph/service decomposition.

Parameters

prid	The identifier of the SDL process.
stgref	A pointer to the storage where the variables will be stored.

Result An *Entity-dict* where the descriptors have been updated.

Algorithm

Line 1-2 Obtain the set of all variables (including process formal parameters) which are declared in the SDL process.

Line 3-4 For each variable in this set, update its descriptor such that it points to the storage where its value will be stored, and the variable identifier itself will be used as “address” for its value in the storage.

create-process-vars(*prid*, *actparml*)(*dict*) ≜ (5.1.3)

```
1   (let mk-ProcessDD(parmddl, , , ,)  = dict((prid, PROCESS)),
2        allvars = {varid | (varid, VALUE) ∈  dom dict ∧ enclosing-scopeunit(varid) = prid ∧
3                            is-VarDD(dict((varid, VALUE)))} in
4    for i = 1 to len parmddl do
5    update-stg(parmddl[i], actparml[i])(dict);
6    create-local-vars(allvars \ elems parmddl)(dict))
```

type: *Process-identifier*$_1$ *Value-List* → *Entity-dict* ⇒

Objective Create all process local variables (including process formal parameters) in their storage. Process formal parameters are initialized with the corresponding actual parameter values.

Parameters

prid The identifier of the SDL process.

actparml The list of actual parameter values.

Algorithm

Line 1 Obtain the list of formal parameter descriptors for the process.

Line 2-3 Obtain the set of all variables declared in the SDL process.

Line 4-5 Create each formal parameter in the storage with the corresponding actual parameter value as initial value.

Line 6 Create all “purely local” variables in the storage.

create-local-vars(*vars*)(*dict*) ≜ (5.1.4)

```
1   for all varid ∈  vars do
2   (let mk-varDD(, , oinit, ,) = dict((varid, VALUE)) in
3    let init =  eval-ground-expression(oinit)(dict) in
4    update-stg-dcl(varid, init)(dict))
```

type: *Variable-identifier*$_1$-**set** → *Entity-dict* ⇒

Objective Create all “purely process local” variables in their storage, possibly initialized with some value.

Parameters

vars The set of local variables.

Algorithm

Line 1 For each variable do the following:

Line 2-3 Evaluate the optional initialisation expression for the variable.

Line 4 Create the variable in the storage with the initialization value or “undefined” as initial value. If the initial value is outside the range of the sort/syntype of the variable, its initial value becomes “undefined” rather than giving rise to a range check error.

int-process-graph-or-service-decomp(*prid*)(*dict*) ≜ (5.1.5)

```
1  (output mk-Instance-Created( ) to dict(ADMIN);
2   (let-mk-ProcessDD(, , , ograph,) = dict((prid, PROCESS)) in
3    if ograph ≠ nil then
4      int-process-graph(ograph)(dict)
5     else
6      int-service-decomp(prid)(dict));
7   output mk-Stop-Instance( ) to dict(ADMIN);
8   output mk-Die(dict(SELF), prid) to view)
```

type: *Process-identifier*$_1$ → *Entity-dict* ⇒

Objective Interpret the process graph/service decomposition of an SDL process.

Parameters

prid The SDL identifier of the process.

Algorithm

Line 1 Send an initialization acknowledgement to the *process-set-admin* instance managing the process.

Line 2 Obtain the (optional) process graph of the SDL process.

Line 3-6 If the process graph is present then interpret it (line 4). Otherwise the process is decomposed into services and these are interpreted (line 6).

Line 7 Tell the managing *process-set-admin* instance that this SDL process instance is stopping.

Line 8 Tell the *view* processor that it should remove any variables revealed by the stopping SDL process instance.

int-process-graph(*graph*)(*dict*) ≜ (5.1.6)

```
1  (trap-exit(STOP) with I in
2   int-graph(graph)(dict))
```

type: *Process-graph*$_1$ → *Entity-dict* ⇒

Objective Interpret the body of an SDL process which is *not* decomposed into services.

Parameters

graph The process graph.

Algorithm

Line 1-2 Start interpretation of the graph nodes. A **stop** node in the graph will cause an **exit**(STOP) to be performed which will be trapped in line 1.

int-service-decomp(*prid*)(*dict*) ≜ (5.1.7)

```
(start-services(prid)(dict);
 exec-service-starts(dict);
 exec-service-states(dict))
```

type: *Process-identifier*$_1$ → *Entity-dict* ⇒

Objective Interpret the body of an SDL process which is decomposed into services. The function does not perform the execution itself but creates and manages the *sdl-service* instances which are required for interpreting the services.

Parameters

prid The SDL identifier of the process.

Algorithm

Line 1 Start one instance of each service and wait until they are all ready to execute their start transitions.

Line 2 Manage the execution of the start transitions of the services.

Line 3 Manage the execution of the state transitions of the services as long as there are still SDL services alive.

start-services(*prid*)(*dict*) ≜ (5.1.8)

```
1  (let servset = {servid | (servid, SERVICE) ∈ dom dict ∧ enclosing-scopeunit(servid) = prid } in
2   for all servid ∈ servset do
3   (let dict′ = dict + [ADMIN ↦ self,
4                         PORT   ↦ self] in
5    def servbody : start sdl-service(servid)(dict′);
6    servinstmap := c servinstmap + [servbody ↦ servid];
7    input mk-Instance-Created( ) from servbody
8        ⇒ I);
9   output mk-Instance-Created( ) to dict(ADMIN))
```

type: *Process-identifier*$_1$ → *Entity-dict* ⇒

Objective Create SDL service instances for a new SDL process instance.

Parameters

prid The SDL identifier of the process.

Algorithm

Line 1 Obtain the set of identifiers for all services defined in the SDL process.

Line 2 For each service do the following (line 3-7):

Line 3-4 For use by the service both the ADMIN and PORT entries in *Entity-dict* should contain the II value of the managing *sdl-process.* This is because the meta-communication which in case of interpretation of a process graph is done directly with the associated *process-set-admin* and *input-port* instances in case of interpretation of a service graph should go through the *sdl-process* instance. Thus the interpretation functions for graph nodes do not need to distinguish between process and service graph nodes.

Line 5 Start the *sdl-service* instance which will interpret the SDL service.

line 6 Update the service instance map to include the new service.

Line 7 Wait for an initialization acknowledgement from the service.

Line 9 When all service instances have been created and initialized, then send an initialization acknowledgement for the whole process instance to its managing *process-set-admin* instance.

exec-service-starts(*dict*) ≜ (5.1.9)

```
1    for all servbody ∈ c dom servinstmap do
2    (output mk-Execute-Start( ) to servbody;
3     exec-service-transition(servbody)(dict))
```

type: *Entity-dict* ⇒

Objective Manage the initial execution of service transitions until each service has either entered its first state or stopped. Note that the first state of a service may be inside a procedure. No two initial service transitions may be executed at the same time, and all initial transitions must have been executed before any signal input or spontaneous transition is made in any service.

Algorithm

Line 1 For each service instance do the following:

Line 2 Instruct the service instance to execute its initial transition.

Line 3 Wait until the service reaches a state (possibly in a procedure) or stops.

exec-service-states(*dict*) ≜ (5.1.10)

```
1    while c servinstmap ≠ [] do
2    ((def saveset′ : union rng c savemap,
3             sponrt′ : true ∈ rng c spontmap;
4      output mk-Next-Signal(saveset′, spont′) to dict(PORT));
5     {input mk-Input-Signal(sid, vl, se) from dict(PORT)
6          ⇒  if (∃servid ∈ rng c servinstmap)(sid ∈ s-Input-signal-set(dict((servid, SERVICE)))) then
7                 (def servid ∈ rng c servinstmap s.t. sid ∈ s-Input-signal-set(dict((servid, SERVICE)));
8                  def servbody ∈ dom c servinstmap s.t. c servinstmap(servbody) = servid;
9                  output mk-Input-Signal(sid, vl, se) to servbody;
10                 exec-service-transition(servbody)(dict))
11               else
12                I,
13     input mk-Spontaneous-Signal( ) from dict(PORT)
14         ⇒  (def-servid ∈ dom c spontmap s.t. c spontmap(servid);
15             def-servbody ∈ dom c servinstmap s.t. c servinstmap(servbody) = servid;
16             output mk-Spontaneous-Signal( ) to servbody;
17             exec-service-transition(servbody)(dict))})
```

type: *Entity-dict* ⇒

Objective Manage the execution of service state transitions. Note that the execution of a state transition may start in a procedure and/or end in the same or another procedure. No two service state transitions (in two different services) may be executed at the same time.

Algorithm

Line 1 One iteration of this loop is performed for each execution of a service transition. At the beginning of each iteration of the loop all service instances still alive are waiting in a state, ie. each interpreting *sdl-service* instance is waiting for input after outputting *Next-Signal* to this *sdl-process*. The loop terminates when all service instances have stopped.

Line 2 From all **save** sets of service instances still alive, obtain the total save signal set to be sent to the *input-port* instance.

Line 3 If at least one service is in a state containing spontaneous transitions the *input-port* instance should be able to provoke this.

Line 4 Request the next signal from the input port, taking the *saveset'* and *spont'* into consideration.

Line 5 Covers the delivery of a signal to some service.

Line 6-12 If the service which should receive this signal is no longer alive the signal is discarded.

Line 7 Obtain the SDL identifier of the service instance which should receive the signal.

Line 8 Obtain the II value of the *sdl-service* instance interpreting this service instance.

Line 9 Deliver the signal to the service.

Line 10 Wait until the service has completed the execution of the transition.

Line 13 Covers the triggering of a spontaneous transition in some service.

Line 14 Obtain the SDL identifier of an arbitrary service instance which is currently able to perform a spontaneous transition.

Line 15 Obtain the II value of the *sdl-service* instance interpreting this service instance.

Line 16 Instruct the chosen service to execute a spontaneous transition.

Line 17 Wait until the service has completed the execution of the transition.

exec-service-transition(*servbody*)(*dict*) ≜ (5.1.11)

```
1   (trap exit (ENDTRANS) with I in
2    cycle {input mk-Stop-Instance( ) from servbody
3           ⇒ (def servid : c servinstmap(servbody);
4                servinstmap : = c servinstmap \ {servbody};
5                savemap := c savemap \ {servid};
6                spontmap := c spontmap \ {servid};
7                exit (ENDTRANS)),
8         input mk-Next-Signal(saveset′, spont′) from servbody
9           ⇒ (def servid : c servinstmap)(servbody);
10               savemap := c savemap + [servid ↦ saveset′];
11               spontmap := c spontmap + [servid ↦ spont′];
12               exit (ENDTRANS)),
13        input mk-Set-Timer(tid, argl, expt) from servbody
14          ⇒ output mk-Set-Timer(tid, argl, expt) to dict(PORT),
15        input mk-Reset-Timer(tid, argl) from servbody
16          ⇒ output mk-Reset-Timer(tid, argl) to dict(PORT),
17        input mk-Active-Request(tid, argl) from servbody
18          ⇒ (output mk-Active-Request(tid, argl) to dict(PORT);
19               input mk-Active-Answer(stat) from dict(PORT)
20                 ⇒ output mk-Active-Answer(stat) to servbody)})
```

type: *II*(*sdl-service*) → *Entity-dict* ⇒

Objective Manage the execution of a service transition and relay the timer communication between the interpreting *sdl-service* and the *input-port*.

Parameters

servbody The II value of the *sdl-service* instance interpreting the transition.

Algorithm

Line 1-2 The function enters a **cycle** which exits with **exit**(ENDTRANS) when the execution of the service transition has finished. This **exit** is trapped by line 1.

Line 2	Handle the case where the execution of the service transition is terminated by a **stop** node.
Line 3	Obtain the SDL identifier of the stopping service instance.
Line 4-6	Delete the stopping service from the service administration maps of the process instance.
Line 7	Exit the **cycle**.
Line 8	Handle the case where the execution of the service transition is terminated by a **nextstate** node.
line 9	Obtain the SDL identifier of the service instance.
Line 10-11	Insert the new **save** signal set and spontaneous-indication in the save set and spontaneous transition maps.
Line 12	Exit the **cycle**.
Line 13-20	Relay the timer handling meta-communication between the service graph and the input port.

5.2 The sdl-service Processor

An instance of the *sdl-service* processor is created by its managing *sdl-process* instance for each service in the interpreted SDL process. The *sdl-service* instance first performs the necessary initial setup and then interprets the service graph. When the SDL service instance ceases to exist the necessary cleanup is performed, and the *sdl-service* instance ceases to exist.

sdl-service **processor** (*servid*)(*dict*) $\triangleq$ (5.2.1)

```
1   (dcl servstg := [] type Stg;
2    (let mk-Identifier1(qual, nm) = servid,
3         level = qual ⌢ ⟨mk-Service-qualifier1(nm)⟩ in
4     let dict′ = dict + [SCOPEUNIT ↦ level] in
5     def dict″ : modify-service-vardds(servid, servstg)(dict′);
6     trap exit () with error in
7     (create-service-vars(servid)(dict″);
8      int-service-graph(servid)(dict″))))
```

type: *Service-identifier*$_1$ → *Entity-dict* ⇒

Objective Interprets (the body of) an SDL service.

Parameters

servid The SDL identifier of the service.

Algorithm

Line 1 Declare a variable servstg which is to be the local storage of this SDL service instance and initialize it to be empty.

Line 2-3 Construct the qualifier for the service.

Line 4 Enter the current scope unit into the *Entity-dict.*

Line 5 For all variables declared in this service, modify their descriptors such that they can be used for interpreting the service graph.

Line 6 Trap any **exit** with **error**.

Line 7 Create all service local variables in the storage.

Line 8 Interpret the service graph.

modify-service-vardds(*servid, stgref*)(*dict*) $\triangleq$ (5.2.2)

```
1   modify-process-vardds(servid, stgref)(dict)
```

type: *Service-identifier*$_1$ **ref** *Stg* → *Entity-dict* → *Entity-dict*

Objective Modify the *Entity-dict* descriptors for the variables local to a given service such that they can be used for interpretation of its service graph.

Parameters

servid The identifier of the service.

stgref A pointer to the storage where the variables will be stored.

Result An *Entity-dict* where the descriptors have been updated.

Algorithm

Line 1 Service variable descriptors are updated in the same way as process variable descriptors.

create-service-vars(*servid*)(*dict*) ≜ (5.2.3)

```
1   (let allvars = {varid | (varid, VALUE) ∈ dom dict ∧ enclosing-scopeunit(varid) = servid ∧
2                              is-VarDD(dict((varid, VALUE)))} in
3    create-local-vars(allvars)(dict))
```

type: *Service-identifier*$_1$ → *Entity-dict* ⇒

Objective Create all service local variables in their storage.

Parameters

servid The identifier of the service.

Algorithm

Line 1-2 Obtain the set of all variables declared in the service.

Line 3 Create the variables in the storage.

int-service-graph(*servid*)(*dict*) ≜ (5.2.4)

```
1   (output mk-Instance-Created( ) to dict(ADMIN);
2    (let mk-ServiceDD(graph, ,) = dict((servid, SERVICE)) in
3     trap-exit(STOP) with I in
4     int-graph(graph)(dict));
5    output mk-Stop-Instance( ) to dict(ADMIN);
6    output mk-Die(dict(SELF), servid) to view)
```

type: *Service-identifier*$_1$ → *Entity-dict* ⇒

Objective Interpret a service graph.

Parameters

servid The identifier of the containing service.

Algorithm

Line 1 Send an initialization acknowledgement to the *sdl-process* instance managing the service.

Line 2 Obtain the service graph of the service.

Line 3-4 Start interpretation of the graph nodes. A **stop** node in the graph will cause an **exit**(STOP) to be performed which will be trapped in line 3.

Line 5 Tell the managing *sdl-process* instance that the service instance is stopping.

Line 6 Tell the *view* processor that it should remove any variables revealed by the stopping service instance.

5.3 Interpretation of a Procedure

Describes the interpretation of a procedure after its actual parameters have been evaluated.

int-procedure(*prid*, *actparml*)(*dict*) ≜ (5.3.1)

1 (**dcl** prcdstg := [] **type** *Stg*;

2 (**let mk**-*Identifier*$_1$(*qual*, *nm*) = *prid*,

3 *level* = *qual* $\frown$ ⟨**mk**-*Procedure-qualifier*$_1$(*nm*)⟩ **in**

4 **let** *dict′* = *dict* + [SCOPEUNIT] ↦ *level*] **in**

5 **def** *dict″* : *modify-procedure-vardds*(*prid*, *actparml*, prcdstg)(*dict′*);

6 *create-procedure-vars*(*prid*, *actparml*)(*dict″*);

7 *int-procedure-graph*(*prid*)(*dict″*)))

type: *Procedure-identifier*$_1$ (*Variable-identifier*$_1$ | *Value* | UNDEFINED)* → *Entity-dict* ⇒

Objective Interprets a procedure.

Parameters

prid The SDL identifier of the procedure.

actparml The list of actual parameter values. For an **in**/**out** parameter the parameter “value” is the identifier of the actual parameter variable.

Algorithm

Line 1 Declare a variable prcdstg which is to be the local storage of this procedure instance and initialize it to be empty.

Line 2-3 Construct the qualifier for the procedure.

Line 4 Enter the current scope unit into the *Entity-dict*.

Line 5 For all variable declared in this procedure (including **in** formal parameters), modify their descriptors such that they can be used for interpreting the procedure graph.

Line 6 Create all procedure local variables in the storage.

Line 7 Interpret the procedure graph.

modify-procedure-vardds(*prid*, *actparml*, *stgref*)(*dict*) ≜ (5.3.2)

(**let mk**-*ProcedureDD*(*parmddl*,) = *dict*((*prid*, PROCEDURE)),

allvars′ = {*varid* | (*varid*, VALUE) ∈ **dom** *dict* ∧ *enclosing-scopeunit*(*varid*) = *prid* ∧

is-*VarDD*(*dict*((*varid*, VALUE)))} **in**

dict + [(*fvarid*, VALUE) ↦ *dict*((*actparml*[*i*], VALUE)) |

i ∈ **ind** *parmddl* ∧ **is**-*InoutparmDD*(*parmddl*[*i*]) ∧

mk-*InoutparmDD*(*fvarid*) = *parmddl*[*i*]]

+ [(*varid*, VALUE) ↦ **mk**-*VarDD*(*varid*, *sort*, *oinit*, *rev*, *stgref*) |

varid ∈ *allvars′* ∧ **mk**-*VarDD*(, *sort*, *oinit*, *rev*,) = *dict*((*varid*, VALUE))])

type: *Procedure-identifier*$_1$ (*Variable-identifier*$_1$ | *Value* | UNDEFINED)* **ref** *Stg* → *Entity-dict* → *Entity-dict*

Objective Modify the *Entity-dict* descriptors for the variables (including **in** formal parameters) local to a given procedure such that they can be used for interpretation of its procedure graph.

Parameters

prid The identifier of the procedure.

actparml The list of actual parameter values/variables.

stgref A pointer to the storage where the variables will be stored.

Result An *Entity-dict* where the descriptors have been updated.

Algorithm

Line 1 Obtain the list of formal parameter descriptors for the procedure.

Line 2-3 Obtain the set of all variables (including **in** formal parameters) which are declared in the procedure.

Line 4-6 The variable descriptor for each **in**/**out** formal parameter becomes the same as that of the corresponding actual parameter variable. This means that the descriptor of the formal parameter will point to the same storage as that of the actual parameter variable, and that it will use the same "address" as the actual parameter for accessing or changing its value in the storage.

Line 7-8 For each variable in the set *allvars'*, update its descriptor such that it points to the storage where its value will is stored, and the variable identifier itself will be used as "address" for its value in the storage.

create-procedure-vars(*prid*, *actparml*)(*dict*) $\triangleq$ (5.3.3)

1 (**let mk**-*ProcedureDD*(*parmddl*,) = *dict*((*prid*, PROCEDURE)),
2 *allvars'* = {*varid* | (*varid*, VALUE) ∈ **dom** *dict* ∧ *enclosing-scopeunit*(*varid*) = *prid* ∧
3 **is**-*VarDD*(*dict*((*varid*, VALUE)))},
4 *invars* = {*varid* | **mk**- *InparmDD*(*varid*) ∈ **elems** *parmddl*} **in**
5 **for** *i* = 1 **to len** *parmddl* **do**
6 **if is**- *InparmDD*(*parmddl*[*i*]) **then**
7 *update-stg*(**s**- *Variable-identifier*$_1$ (*parmddl*[*i*]), *actparml*[*i*])(*dict*)
8 **else**
9 **I**;
10 *create-local-vars*(*allvars'* \ *invars*)(*dict*))

type: *Procedure-identifier*$_1$ (*Variable-identifier*$_1$ | *Value* | UNDEFINED* → *Entity-dict* ⇒

Objective Create all procedure local variables (including **in** formal parameters) in their storage. Procedure in formal parameters are initialized with the corresponding actual parameter values.

Parameters

prid The identifier of the procedure.

actparml The list of actual parameter values/variables.

Algorithm

Line 1 Obtain the list of formal parameter descriptors for the procedure.

Line 2-3 Obtain the set of all variables (except of **in**/**out** formal parameters) declared in the procedure.

Line 4 Obtain the set of **in** formal parameter variables.

Line 5-9 Create each **in** formal parameter in the storage with the corresponding actual parameter value as initial value.

Line 10 Create all "purely local" variables in the storage.

int-procedure-graph(*prid*)(*dict*) $\triangleq$ (5.3.4)

1 (**let mk**-*ProdecureDD*(, *graph*) = *dict*((*prid*, PROCEDURE)) **in**
2 **trap exit** (RETURN) **with I in**
3 *int-graph*(*graph*)(*dict*))

type: *Procedure-identifier*$_1$ → *Entity-dict* ⇒

Objective Interpret a procedure graph.

Parameters

servid The identifier of the containing procedure.

Algorithm

Line 1 Obtain the procedure graph of the procedure.

Line 2-3 Start interpretation of the graph nodes. A **return** node in the graph will cause an **exit**(RETURN) to be performed which will be trapped in line 2.

5.4 Storage Handling

update-stg-dcl(*id*, *val*)(*dict*) $\triangleq$ (5.4.1)

1 *update-stg′*(*id*, *val*, DCLASSIGN)(*dict*)

type: *Variable-identifier*$_1$ (*Value* | UNDEFINED) → *Entity-dict* ⇒

Objective Assign an initial value to a variable declared by **dcl**. If the value is outside the range of the sort/syntype of the variable its initial value becomes "undefined". Reveal the initial value of the variable if it has the **revealed** attribute.

Parameters

id The identifier of the variable.

val The initial value of the variable.

Algorithm

Line 1 Call a general-purpose function to update variables in their storages.

update-stg(*id*, *val*)(*dict*) $\triangleq$ (5.4.2)

1 *update-stg′* (*id*, *val*, OTHERASSIGN)(*dict*)

type: *Variable-identifier*$_1$ (*Value* | UNDEFINED) → *Entity-dict* ⇒

Objective Assign an initial *value* to a process formal parameter or procedure **in** formal parameter, or assign a new value to any kind of variable. If the value is outside the range of the sort/syntype of the variable a range check error occurs. Reveal the new value of the variable if it has the **revealed** attribute.

Parameters

id The identifier of the variable.

val The new value of the variable.

Algorithm

Line 1 Call a general-purpose function to update variables in their storages.

update-stg′ (*id*, *val*, *asgnkind*)(*dict*) ≜ (5.4.3)

```
1   (let mk-VarDD(vid, sid, , revealed, stg′ ) = dict((id, VALUE)) in
2    let val′ = (range-check(sid, val)(dict)
3                 → val,
4               asgnkind = DCLASSIGN
5                 → UNDEFINED,
6               asgnkind = OTHERASSIGN
7                 → exit(“§5.3.1.9: Value is not within the range of the syntype”)) in
8    stg′ := c stg′ + [vid ↦ val′];
9    if revealed = REVEALED then
10     output mk-Reveal(vid, sid, dict(SELF), val′) to view
11    else
12     I)
```

type: *Identifier*$_1$ (*Value* | UNDEFINED) (DCLASSIGN | OTHERASSIGN) → *Entity-dict* ⇒

Objective Assign an initial or new value to any kind of variable. The parameter *asngkind* determines what happens if the value is outside the range of the sort/syntype of the variable. Reveal the initial/new value of the variable if it has the **revealed** attribute.

Parameters

id	The identifier of the variable.
val	The initial/new value.
asgnkind	Determines what happens if the value is outside the range of the sort/syntype of the variable.

Algorithm

Line 1	Lookup the description of the variable identifier.
Line 2-7	Perform a range check on the value and obtain the value which will be assigned to the variable. If the value is within the range of the sort/syntype of the variable it gets this value (line 2-3). Otherwise, if the value is the result of the evaluation of an initializer expression in the declaration of the variable, the variable becomes “undefined” (line 4-5). Otherwise, a range check error occurs (line 6-7).
Line 8-9	The referenced storage is overwritten with the new variable – value pair.
Line 9-12	If the variable is **revealed** the initial/new value is sent to the *view* processor.

5.5 Interpretation of a Process, Service or Procedure Graph

Describes the interpretation of a behaviour graph divided into an interpretation function for each type of graph node.

int-graph(*graph*)(*dict*) ≜ (5.5.1)

```
1    (let (start, statenodes) = decomp-graph(graph) in
2     tixe [statenm ↦ int-state-node(statenode)(dict) |
3            statenode ∈ statenodes ∧ s-State-name_1(statenode) = statenm] in
4     int-start-node(start)(dict))
```

type: (*Process-graph*$_1$ | *Service-graph*$_1$ | *Procedure-graph*$_1$) → *Entity-dict* ⇒

Objective Interprets a process, service or procedure graph.

Parameters

graph The process/service/procedure graph.

Algorithm

Line 1 Partition of the graph into a start node and a set of states.

Line 2 Traps all **exit**(*statenm*) from *int-state-node* and *int-transition* by interpreting the associated *State-node*$_1$. The **tixe** construct is a very convenient way to model the "goto"s used in the nextstate nodes. The keyword **tixe** is followed by a map from state names into call of int-state-node with the state-node associated to state name as actual parameter. If an **exit**(*statenm*) is encountered within the dynamic scope of the **tixe** construct, that is either in the range of the **tixe** map (*int-state-node*) or in *int-start-node*, the interpretation of the process continues with the *State-node*$_1$ having the name *statenm.*

Line 4 Interpretation of the start node.

int-start-node (*start*)(*dict*) ≜ (5.5.2)

```
1    (let trans = decomp-start-node(start) in
2     if is-Service-start-node_1 (start) then
3        (input mk-Execute-Start( ) from dict(ADMIN)
4           ⇒ int-transition(trans)(dict))
5      else
6       int-transition(trans)(dict))
```

type: (*Process-start-node*$_1$ | *Service-start-node*$_1$ | *Procedure-start-node*$_1$) → *Entity-dict* ⇒

Objective Interprets a process, procedure or service start node.

Parameters

start The start node.

Algorithm

Line 1 Extract the start transition from the start node.

Line 2-3 If the start node is a service start node then wait until the managing *sdl-process* instance instructs this *sdl-service* instance to interpret the start node. This prevents the simultaneous execution of several service start transitions belonging to the same SDL process instance.

Line 4 Interpret the start transition.

Line 6 If the start node is a process or procedure start node its interpretation starts immediately.

int-start-node(**mk**-*State-node*$_1$(, **mk**-*Save-signalset*$_1$(*saveset*), *inputset*, *spontrset*))(*dict*) ≜ (5.5.3)

```
1    (output mk-Next-Signal(saveset, spontrset ≠ {}) to dict(PORT);
2     {input mk-Input-Signal(sid′, actparml, sender′) from dict(PORT)
3         ⇒ (dict(SENDER) := sender′;
4             (let mk-Input-node1(sid, formparml, trans) ∈ inputset be s.t. sid = sid′ in
5              for i = 1 to len formparml do
6              if formparml[i] ≠ nil
7                then update-stg(formparml[i], actparml[i])(dict)
8                else I;
9              int-transition(trans)(dict))),
10    input mk-Spontaneous-Signal( ) from dict(PORT)
11        ⇒ (dict(SENDER) := dict(SELF);
12            (let mk-Spontaneous-transition1(trans) ∈ spontrset in
13             int-transition(trans)(dict)))})
```

type: State-*node*$_1$ $\rightarrow$ *Entity-dict* $\Rightarrow$

Objective Interprets a state node.

Parameters

state-node Composed of a *saveset* which is a set of signals to be saved by the input port, an *inputset* which is a set of signals and associated transitions, and *spontrset* which is a (possibly empty) set of spontaneous transitions.

Algorithm

Line 1 Request the input port to output a signal which is not in the *saveset*, and to save all signals belonging to the saveset. If the state contains spontaneous transitions the input port may choose to provoke a spontaneous transition instead.

Line 2 Receive a signal composed of a signal identifier, a list of data values and the SDL Pid value of the sender.

Line 3 Update the **sender** value.

Line 4 Select the input node that has the same signal identifier as the received signal.

Line 5-8 For all the formal parameters: if the formal parameter is present (different from **nil**), then the storage is updated with its associated variable and the value of the actual parameter.

Line 9 Interpret the selected transition.

Line 10 Initiate a spontaneous transition. The input port can only respond with this answer if the second parameter of *Next-Signal* was **true**.

Line 11 The **sender** value becomes the same as **self**.

Line 12 Select an arbitrary spontaneous transition.

Line 13 Interpret the contained transition.

int-transition(**mk**-*Transition*$_1$(*nodel*, *termordec*))(*dict*) ≜ (5.5.4)

```
1  (for i = 1 to len nodel do
2   int-graph-node(nodel[i])(dict);
3   cases termordec:
4    (mk-Nextstate-node_1(nm)  → exit(nm),
5     mk-Stop-node_1( )        → exit(STOP),
6     mk-Return-node_1( )      → exit(RETURN),
7     mk-Decision-node_1(, ,)  → int-decision-node(termordec)(dict)))
```

type: *Transition*$_1$ → *Entity-dict* ⇒

Objective Interprets a transition.

Parameters

nodel The list of action nodes.

termordec A terminator node or a decision node.

Algorithm

Line 1-2 Interpret the action nodes sequentially.

Line 4 A nextstate node is interpreted by **exit** with the name of the next state.

Line 5 A stop node by **exit** with STOP.

Line 6 A return node by **exit** with RETURN.

Line 7 A decision node by calling the *int-decision-node* function.

int-graph-node(*graphnode*)(*dict*) ≜ (5.5.5)

```
cases graphnode:
 (mk-Task-node_1(asgnortxt)      → int-task-node(asgnortxt)(dict),
  mk-Output-node_1(, , ,)        → int-output-node(graphnode)(dict),
  mk-Create-request-node_1(,)    → int-create-node(graphnode)(dict),
  mk-Call-node_1(,)              → int-call-node(graphnode)(dict),
  mk-Set-node_1(, ,)             → int-set-node(graphnode)(dict),
  mk-Reset-node_1(,)             → int-reset-node(graphnode)(dict))
```

type: *Graph-node*$_1$ → *Entity-dict* ⇒

Objective Interprets a graph node.

Parameters

graphnode The graph node to be interpreted.

int-task-node(*asgnortxt*)(*dict*) ≜ (5.5.6)

```
cases asgnortxt:
 (mk-Assignment-statement_1(,)  → int-assign-stmt(asgnortxt)(dict),
  mk-Informal-text_1( )         → int-informal-text(asgnortxt))
```

type: (*Assignment-statement*$_1$ | *Informal-text*$_1$) → *Entity-dict* ⇒

Objective Interprets a task node.

Parameters

asgnortxt An assignment statement or informal text.

Algorithm

Line 1 The *asgnortxt* is interpreted as either an assignment or as informal text.

int-assign-stmt(**mk**-*Assignment-statement*$_1$(*vid*, *exp*))(*dict*) ≜ (5.5.7)

```
1   (def val : eval-expression(exp)(dict);
2    update-stg(vid, val)(dict))
```

type: *Assignment-statement*$_1$ → *Entity-dict* ⇒

Objective Interprets an assignment statement.

Parameters

vid The target variable.

exp The expression.

Algorithm

Line 1 Evaluate the value of the expression.

Line 2 Update the storage with *vid* and value of the expression.

int-informal-text(**mk**-*Informal-text*$_1$( )) ≜ (5.5.8)

```
1   (/* This informal Meta-IV text denotes the interpretation of informal text */)
```

type: *Informal-text*$_1$ ⇒

int-output-node(**mk**-*Output-node*$_1$(*sid*, *exprl*, *dest*, *via*))(*dict*) ≜ (5.5.9)

```
1   (let mk-SignalDD(sortl,) = dict((sid, SIGNAL)) in
2    def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
3    def destval : (dest = nil
4                     → nil,
5                   (dest, PROCESS) ∈ dom dict
6                     → dest,
7                   ⊤ → eval-expression(dest)(dict));
8    let senderid = process-or-service-scopeunit(dict(SCOPEUNIT)) in
9    if (∀ i ∈ ind vall)(range-check(sortl[i], vall[i](dict))
10     then output mk-Send-Signal(sid, vall, senderid, dict(SELF), destval, via) to system
11     else exit("§5.3.1.9: Value is not within the range of the syntype"))
```

type: *Output-node*$_1$ → *Entity-dict* ⇒

Objective Interprets an output node.

Parameters

sid The identifier of the signal to be sent.

exprl The actual parameters for the signal.

dest An optional Pid expression or process identifier denoting the process to which the signal should be sent.

Via An optional set of signal route/channel identifiers at least one of which should be used to convey the signal.

Algorithm

Line 2 Evaluate the list of actual parameters.

Line 3-7 Evaluate the optional signal destination. If it is absent or is a process identifier (line 3-6) it will be handed on to the *system* processor unchanged. If it is a Pid expression (line 7) this expression is evaluated.

Line 8 Obtain the SDL identifier of the process or service instance which sends the signal.

Line 9 Perform a range check on the actual parameter values.

Line 10 Send the signal.

int-create-node(**mk**-*Create-request-node*$_1$(*prid*, *exprl*))(*dict*) ≜ (5.5.10)

```
1  (let mk-ProcessDD(formparms, , , ,) = dict((prid, PROCESS)) in
2   let sortl = ⟨s-Sort-reference-identifier1(dict((formparms[i], VALUE))) | 1 ≤ i ≤ len formparms⟩ in
3   def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
4   if (∀i ∈ ind sortl)(range-check(sortl[i], vall[i])(dict)) then
5     (output mk-Create-Instance-Request(prid, vall, dict(SELF)) to system;
6      input mk-Create-Instance-Answer(offspring′) from system
7          ⇒  dict(OFFSPRING) := offspring′)
8    else
9     exit("§5.3.1.9: Value is not within the range of the syntype"))
```

type: *Create-request-node*$_1$ → *Entity-dict* ⇒

Objective Interprets a create node.

Parameters

prid The identifier of the process to be created.

exprl The list of actual parameters.

Algorithm

Line 1-2 Establish the list of sort reference identifiers of the formal parameters.

Line 3 Evaluate the list of actual parameters.

Line 4 Perform a range check on the actual parameters.

Line 5 Issue the create instance request.

Line 6 Wait for a response on the create request. The response carries the SDL Pid value of the new process instance.

Line 7 Update the **offspring** value.

int-call-node(**mk**-*Call-node*$_1$(*prid*, *exprl*))(*dict*) ≜ (5.5.11)

```
1  (let mk-ProcedureDD(parmddl,) = dict((prid, PROCEDURE)) in
2   def actparml : ⟨(is-InparmDD(parmddl[i])
3                      → eval-expression(exprl[i])(dict),
4                    is-InoutparmDD(parmddl[i])
5                      → exprl[i]) |
6                    1 ≤ i ≤ len parmddl⟩;
7   int-procedure(prid, actparml)(dict))
```

type: *Call-node*$_1$ → *Entity-dict* ⇒

Objective Interpret a procedure call node.

Parameters

prid The identifier of the procedure to be called.

exprl The actual parameters for the procedure call.

Algorithm

Line 1 Obtain the list of formal parameter descriptors for the procedure.

Line 2-6 Evaluate the list of actual parameters. If an actual parameter is an **in** parameter it is an expression which should be evaluated (line 2-3). If an actual parameter is an **in**/**out** parameter its "evaluation result" is the SDL identifier of the actual parameter variable (line 4-5).

Line 7 Interpret the procedure.

int-set-node(**mk**-*Set-node*$_1$(*texp*, *tid*, *exprl*))(*dict*) ≜ (5.5.12)

```
1  (let mk-SignalDD(sortl,) = dict((tid, SIGNAL)) in
2   def val : eval-expression(texp)(dict);
3   def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
4   if (∀i ∈ ind vall)(range-check(sortl[i], vall[i])(dict))
5      then output mk-Set-Timer(tid, vall, val) to dict(PORT)
6      else exit("§5.3.1.9: Value is not within the range of the syntype"))
```

type: *Set-node*$_1$ → *Entity-dict* ⇒

Objective Interprets a set node.

Parameters

texp The expiration time expression.

tid The identifier of the timer to be set.

exprl The actual parameters for the timer.

Algorithm

Line 2 Evaluate the expiration time expression.

Line 3 Evaluate the list of actual parameters.

Line 4 Perform a range check on the actual parameter values.

Line 5 Instruct the *input-port* to set the timer.

int-reset-node(**mk**-*Reset-node*$_1$(*tid*, *exprl*))(*dict*) ≜ (5.5.13)

```
(let mk-SignalDD(sortl,) = dict((tid, SIGNAL)) in
 def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
 if (∀i ∈ ind vall)(range-check(sortl[i], vall[i](dict))
    then output mk-Reset-Timer(tid, vall) to dict(PORT)
    else exit("§5.3.1.9: Value is not within the range of the syntype"))
```

type: *Reset-node*$_1$ → *Entity-dict* ⇒

Objective Interprets a reset node.

Parameters

tid The identifier of the timer to be reset.

exprl The actual parameters for the timer.

Algorithm

Line 2 Evaluate the list of actual parameters.

Line 3 Perform a range check on the actual parameter values.

Line 4 Instruct the *input-port* to reset the timer.

int-decision-node(**mk**-*Decision-node*$_1$(*quest*, *answset*, *elseansw*))(*dict*) ≜ (5.5.14)

```
1   (def questval : (is-Expression1(quest)
2                        → eval-expression(quest)(dict),
3                    is-Informal-text1(quest)
4                        → quest);
5    let answset′ = matching-answer(questval, answset)(dict) in
6    (answset′ ≠ {}
7       →(let {mk-Decision-answer1(, trans)} = answset′ in
8           int-transition(trans)(dict)),
9    elseansw ≠ nil
10      → (let mk-Else-answer1(trans) = elseansw in
11          int-transition(trans)(dict)),
12   T → exit("§2.7.5: No matching answer")))
```

type: *Decision-node*$_1$ → *Entity-dict* ⇒

Objective Interprets a decision node.

Parameters

quest The question of the decision.

answset The set of answers and associated transitions.

Elseansw The optional **else** transition.

Algorithm

Line 1-3 Evaluate the decision question.

Line 5 Extract the set of answers which match the decision question value.

Line 6-8 If the extracted set of answers is not empty then it contains exactly one answer (it is checked during the building of the *Entity-dict* that the answers do not overlap). The transition associated with the selected answer is interpreted.

Line 9-11 If no matching answer was found, and an **else** transition is present, this transition is interpreted.

Line 12 If no matching answers is found and no **else** answer is present an error occurs.

matching-answer(*questval*, *answset*)(*dict*) ≜ (5.5.15)

```
{mk-Decision-answer1(valsetortext,) ∈ answset |
 (is-Range-condition1(valsetortext) ∧ is-Value(questval)
     → (let branchcond = eval-range-condition(questval, valsetortext)(dict) in
         branchcond = dict(TRUEVALUE)),
  T → text-equality (questval, valsetortext))}
```

type: (*Value* | *Informal-text*$_1$) *Decision-answer*$_1$-**set** → *Entity-dict* → *Decision-answer*$_1$-**set**

Objective Find the set of answers in the supplied set of answers which match the supplied question value.

Parameters

quest The question value of the decision.

answset The set of answers and associated transitions.

Result The matching answer and its associated transition.

Algorithm

Line 2-4 If neither the question nor the answer is informal then the range condition is evaluated w.r.t. the question value.

Line 5 If the question or the answer is informal the equality is tested by the informal function *text-equality*.

text-equality(*value-text*, *valueset-text*) $\triangleq$ (5.5.16)

```
(/∗ This informal Meta-IV text denotes the equality test ∗/;
 /∗ between informal question and/or informal answer ∗/)
```

type: $(Informal\text{-}text_1 \mid Value)\ (Informal\text{-}text_1 \mid Range\text{-}condition_1) \rightarrow Bool$

5.6 Expression Evaluation

This section defines the functions for expression evaluation.

eval-expression(*exp*)(*dict*) $\triangleq$ (5.6.1)

1 **if** *exp* = **nil then**
2 UNDEFINED
3 **else**
4 **cases** *exp*:
5 (**mk**-*Ground-expression*$_1$( )
6 → *eval-ground-expression*(*exp*)(*dict*),
7 **mk**-*Identifier*$_1$(,)
8 → *eval-variable-identifier*(*exp*)(*dict*),
9 **mk**-*Operator-application*$_1$(,)
10 → *eval-operator-application*(*exp*)(*dict*),
11 **mk**-*Conditional-expression*$_1$(, ,)
12 → *eval-conditional-expression*(*exp*)(*dict*),
13 **mk**-*View-expression*$_1$(,)
14 → *eval-view-expression*(*exp*)(*dict*),
15 **mk**-*Timer-active-expression*$_1$(,)
16 → *eval-timer-active-expression*(*exp*)(*dict*),
17 **mk**-*Anyvalue-expression*$_1$( )
18 → *eval-anyvalue-expression*(*exp*)(*dict*),
19 **mk**-*Now-expression*$_1$( )
20 → *eval-now-expression*( ),
21 **mk**-*Self-expression*$_1$( )
22 → *dict*(SELF),
23 **mk**-*Parent-expression*$_1$( )
24 → *dict*(PARENT),
25 **mk**-*Offspring-expression*$_1$( )
26 → **c** *dict*(OFFSPRING),
27 **mk**-*Sender-expression*$_1$( )
28 → **c** *dict*(SENDER),
29 **mk**-*Error-term*$_1$( )
30 → **exit**("§5.4.2.1: Attempt to evaluate error expression"))

type: [*Expression*$_1$] → *Entity-dict* ⇒ (*Value* | UNDEFINED)

Objective Evaluate an expression.

Parameters

exp The expression.

Result The value of the expression.

Algorithm

Line 1-2 If the expression is absent (typically an omitted actual parameter) its value is "undefined" .

Line 21-24 If the expression is **self** or **parent** its value is looked up in the *Entity-dict.*

Line 25-28 If the expression is **offspring** or **sender** a META-IV variable holding its current value is accessed via a pointer which is looked up in the *Entity-dict.*

Line 29-30 If the expression is **error** an error occurs.

5.6.1 Ground Expression Evaluation

eval-ground-expression(*gexpr*)(*dict*) ≜ (5.6.1.1)

```
1   if gexpr = nil then
2     UNDEFINED
3    else
4     (let mk-Ground-expression1(gterm) = gexpr in
5      eval-ground-term(gterm)(dict))
```

type: [*Ground-expression*$_1$] → *Entity-dict* → (*Value* | UNDEFINED)

Objective Evaluate a ground expression.

Parameters

gexpr The ground term.

Result The value of the ground expression.

Algorithm

Line 1-2 If the ground expression is absent its value is “undefined”.

Line 4-5 Obtain the contained ground term (line 4) and evaluate it (line 5).

eval-ground-term(**mk**-*Ground-term*$_1$(*contents*))(*dict*) ≜ (5.6.1.2)

```
1   (is-Identifier1(contents)
2      → (let resterm = mk-Ground-term1(contents) in
3          reduce-term(resterm, dict(SCOPEUNIT))(dict)),
4    is-Conditional-term1(contents)
5      → (let mk-Conditional-term1)(cond, cons, alt) = contents in
6          let condval = eval-ground-term(cond)(dict) in
7          (condval = dict(TRUEVALUE)
8              → eval-ground-term(cons)(dict),
9          condval = dict(FALSEVALUE)
10             → eval-ground-term(alt)(dict))),
11   ⊤ → (let (opid, arglist) = contents in
12         let vallist = ⟨eval-ground-term(arglist[i])(dict) | 1 ≤ i ≤ len arglist⟩ in
13         eval-ground-term-opapp(opid, vallist)(dict)))
```

type: *Ground-term*$_1$ → *Entity-dict* → *Value*

Objective Evaluate a ground term.

Parameters

contents The “contents” of the ground term (a literal identifier, conditional ground expression or operator application on a list of ground terms).

Result The value of the ground term.

Algorithm

Line 1 Handle the case where the ground term is a literal identifier.

Line 2 Build a ground term representing the resulting value.

Line 3 Obtain the ground term which has been chosen to represent the value in the rest of the system.

Line 4 Handle the case where the ground term is a conditional term.

Line 5 Decompose the conditional term into its components.

Line 6 Evaluate the condition.

Line 7-10 If the condition is True (line 7) then evaluate the consequence (line 8). If the condition is False (line 9) then evaluate the alternative (line 9). No other possibilities exist as the wellformedness of the Boolean data sort has been checked during the building of the *Entity-dict.*

Line 11 Handle the case where the ground term is an operator application. Decompose the operator application into an operator identifier and an argument list.

Line 12 Evaluate the argument list.

Line 13 Perform the operator application on the list of argument values.

eval-ground-term-opapp(*opid, vallist*)(*dict*) ≜ (5.6.1.3)

```
1   (let mk-OperatorDD)(sortlist, sort) = dict((opid, VALUE)) in
2    if (∀i ∈ ind sortlist)(range-check(sortlist[i], vallist[i])(dict)) then
3       (let resterm = mk-Ground-term1((opid, vallist)) in
4        let resval = reduct-term(resterm, dict(SCOPEUNIT))(dict) in
5        if range-check(sort, resval)(dict) then
6           resval
7         else
8           exit("§5.3.1.9: Value is not within the range of the syntype"))
9     else
10     exit("§5.3.1.9: Value is not within the range of the syntype"))
```

type: *Operator-identifier*$_1$ *Value*$^+$ → *Entity-dict* → *Value*

Objective Apply an SDL operator to a list of argument values.

Parameters

opid The SDL operator identifier.

vallist The list of argument values.

Result The resulting value of the operator application.

Algorithm

Line 1 Obtain the argument sort list and the result sort of the operator.

Line 2 Perform a range check on the list of argument values.

Line 3 Build a ground term representing the resulting value.

Line 4 Obtain the ground term which has been chosen to represent the value in the rest of the system.

Line 5 Perform a range check on the resulting value.

Line 6 Return the resulting value.

5.6.2 Active Expression Evaluation

eval-variable-identifier(*id*)(*dict*) ≜ (5.6.2.1)

```
1   (let mk-VarDD(vid, , , , stg) = dict((id, VALUE)) in
2    if c stg(vid) ≠ UNDEFINED
3       then c stg(vid)
4       else exit("§5.4.2.2: Value of accessed variable is undefined"))
```

type: *Identifier*$_1$ → *Entity-dict* ⇒ *Value*

Objective Evaluate a variable identifier.

Parameters

id The variable identifier.

Result The contents, if any, of that variable.

Algorithm

Line 1 Gets the referenced variable identifier and a pointer to its storage (the variable *id* could be a procedure **in**/**out** formal parameter).

Line 4 If the contents of storage for the referenced identifier is undefined an error occurs.

Line 3 The contents of storage for the referenced identifier is returned.

eval-operator-application(**mk**-*Operator-application*$_1$(*opid*, *expl*))(*dict*) ≜ (5.6.2.2)

```
1   (def vall : ⟨eval-expression(expl[i])(dict) | 1 ≤ i ≤ len expl⟩;
2    eval-ground-term-opapp(opid, vall)(dict))
```

type: *Operator-application*$_1$ → *Entity-dict* ⇒ *Value*

Objective Evaluate an operator application.

Parameters

opid Identifier of the operator.

expl Argument list for the application.

Result The value of the operator application.

Algorithm

Line 1 Evaluate the list of arguments.

Line 2 Perform the operator application on the list of argument values.

eval-view-expression(**mk**-*View-expression*$_1$(*id*, *exp*))(*dict*) ≜ (5.6.2.3)

```
1   (let mk-ViewDD(sortid) = dict((id, VALUE)) in
2    def pid : if exp = nil then nil else eval-expression(exp)(dict);
3    output mk-View-Request(id, sortid, pid) to view;
4    input mk-View-Answer(val) from view
5      ⇒      if val ≠ UNDEFINED
6                then val
7                else exit("§5.4.2.2: The viewed value is undefined"))
```

type: *View-expression*$_1$ → *Entity-dict* ⇒ *Value*

Objective Evaluate a **view** expression.

Parameters

id The identifier of the viewed variable.

exp An optional Pid expression.

Result The value of the **view** expression.

Algorithm

Line 1 Get the sort or syntype of the viewed variable.

Line 2 Evaluate the Pid expression if present.

Line 3 Request the *view* processor to obtain the value of one of the possible revealed variable instances.

Line 4 Wait for a response from the *view* processor.

Line 5 Check that the contents of the viewed variable instance is not "undefined".

Line 6 Return the viewed value.

eval-conditional-expression(**mk**-*Conditional-expression*$_1$(*cond*, *cons*, *alt*))(*dict*) $\triangleq$ (5.6.2.4)

```
1   (def condval : eval-expression(cond)(dict);
2    (condval = dict(TRUEVALUE)
3         → eval-expression(cons)(dict),
4     condval = dict(FALSEVALUE)
5         → eval-expression(alt)(dict)))
```

type: *Conditional-expression*$_1$ → *Entity-dict* ⇒ *Value*

Objective Evaluate a conditional expression.

Parameters

cond The condition expression.

cons The consequence expression.

alt The alternative expression.

Result The value of either the consequence or the alternative expression depending on the condition.

Algorithm

Line 1 Evaluate the condition.

Line 2-5 If the condition is True (line 2) then evaluate the consequence expression (line 3). If the condition is False (line 4) then evaluate the alternative expression (line 5). No other possibilities exist as the wellformedness of the Boolean data sort has been checked during the building of the *Entity-dict*.

eval-timer-active-expression(**mk**-*Timer-active-expression*$_1$(*timer*, *exprl*))(*dict*) $\triangleq$ (5.6.2.5)

```
(let mk-SignalDD(sortl,) = dict((timer, SIGNAL)) in
 def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
 if (∀i ∈ ind vall)(range-check(sortl[i], vall[i])(dict)) then
    (output mk-Active-Request(timer, vall) to dict(PORT);
     input mk-Active-Answer(b) from dict(PORT)
         ⇒   if b then dict(TRUEVALUE) else dict(FALSEVALUE))
   else
    exit("§5.3.1.9: Value is not within the range of the syntype"))
```

type: *Timer-active-expression*$_1$ → *Entity-dict* ⇒ *Value*

Objective Evaluate a timer active expression.

Parameters

timer The identifier of the timer.

exprl The arguments of the timer.

Result The SDL Boolean value of the timer **active** expression.

Algorithm

Line 1 Establish the sort list of the timer.

Line 2 Evaluate the timer arguments.

Line 3 Perform a range check on the list of argument values.

Line 4 Request the input port to examine if the timer instance is active.

Line 5 Receive a response from the input port with a parameter *b* denoting the “activeness” of the timer instance.

Line 6 Return the SDL value True or False depending on the answer from the input port.

eval-anyvalue-expression(**mk**-*Anyvalue-expression*$_1$(*sortref*))(*dict*) ≙ (5.6.2.6)

```
1   (let sortid = sort-or-parent-sort(sortref)(dict) in
2    let values = {val ∈ values-of-sort(sortid)(dict) | range-check(sortref, val)(dict)} in
3    if values ≠ {} then
4       (let val ∈ values in
5        val)
6      else
7       exit("§5.4.4.6: Attempt to evaluate an anyvalue expression for an empty sort or syntype"))
```

type: *Anyvalue-expression*$_1$ → *Entity-dict* → *Value*

Objective Evaluate an anyvalue expression.

Parameters

sortref The contained sort/syntype identifier of the anyvalue expression.

Result The (arbitrary) value of the anyvalue expression.

Algorithm

Line 1 If the sort/syntype identifier is a syntype identifier then obtain its parent sort.

Line 2 Obtain the set of all values belonging to the sort/syntype.

Line 3-7 It is an error to apply **any** to a sort or syntype containing no values.

Line 4-5 Select an arbitrary value from the value set and return it.

eval-now-expression( ) ≙ (5.6.2.7)

```
(output mk-Time-Request( ) to timer;
 input mk-Time-Answer(val) from timer
     ⇒  val)
```

type: ( ) ⇒ *Value*

Objective Evaluate the **now** expression.

Result The current value of **now**.

Algorithm

Line 1 Request the *timer* processor to get the current time.

Line 2 Wait for a response from *timer*.

Line 3 Return the result.

5.7 Range Check and Range Condition Evaluation

This section defines functions for range checks and for evaluation of range conditions w.r.t. given SDL data values.

Range-check(*sortref*, *value*)(*dict*) ≜ (5.7.1)

```
1   if value = UNDEFINED then
2     true
3    else
4     cases dict((sortref, SORT)):
5      (mk-SyntypeDD(, rangecond)
6        → (let testval = eval-range-condition(value, rangecond)(dict) in
7             testval = dict(TRUEVALUE)),
8      mk-SortDD( )
9        → true
```

type: *Sort-reference-identifier*$_1$ (*Value* | UNDEFINED) → *Entity-dict* → *Bool*

Objective Test whether a value is within the range of a sort/syntype.

Parameters

sortref The sort/syntype identifier.

value The value.

Result **true** if the value is within the range, else **false**.

Algorithm

Line 1-2 If the value is "undefined" (typically an omitted actual parameter) it is considered to be in the range of any sort/syntype.

Line 4 Look up the sort/syntype in the *Entity-dict.*

Line 5-7 Handle the case where the sort/syntype is a syntype. The associated range condition is retrieved (line 5) and evaluated w.r.t. the value to be checked (line 6). The range check is **true** if the range condition evaluation result is the SDL value True (line 7).

Line 8-9 If the sort/syntype is a sort the range check is always **true**.

eval-range-condition(*value*, **mk**-*Range-condition*$_1$(*orid*, *cset*))(*dict*) ≜ (5.7.2)

```
1   eval-condition-item-set(value, orid, cset)(dict)
```

type: *Value Range-condition*$_1$ → *Entity-dict* → *Value*

Objective Evaluate a range condition w.r.t. a given value.

Parameters

value The value.

orid The *Or-operator-identifier*$_1$ of the range condition.

cset The condition items of the range condition.

Result The SDL Boolean evaluation result.

Algorithm

Line 1 Call a function which evaluates each condition item w.r.t. the value and takes the SDL Boolean **or** of the results.

eval-condition-item-set(*value*, *orid*, *cset*)(*dict*) $\triangleq$ (5.7.3)

```
1    (let cond ∈ cset in
2     let condval = eval-condition-item(value, cond)(dict) in
3     if card cset = 1 then
4        condval
5      else
6        (let restval = eval-condition-item-set(value, orid, cset \ {cond})(dict) in
7         eval-ground-term-opapp(orid, ⟨condval, restval⟩)(dict)))
```

type: *Value Or-operator-identifier*$_1$ *Condition-item*$_1$-**set** $\rightarrow$ *Entity-dict* $\rightarrow$ *Value*

Objective Evaluate a set of range condition items w.r.t. a given value and take the SDL Boolean **or** of the results.

Parameters

value The value.

orid The AS_1 identifier for the SDL predefined Boolean **or** operator.

cset The (non-empty) set of range condition items.

Result The SDL Boolean evaluation result.

Algorithm

Line 1 Pick a condition item from the condition items set.

Line 2 Evaluate this condition item w.r.t. the *value.*

Line 3-4 If the picked condition item is the only one in the condition item set then return the evaluation result obtained in line 2.

Line 6 Evaluate the remaining set of condition items w.r.t. the *value* and take the SDL Boolean **or** of the results.

Line 7 Apply the SDL Boolean **or** operator to the two sub-evaluation results.

eval-condition-item(*value*, *cond*)(*dict*) $\triangleq$ (5.7.4)

```
cases cond:
 (mk-Open-range1(relopid, gexpr)
     → (let gval = eval-ground-expression(gexpr)(dict) in
          eval-ground-term-opapp(relopid, ⟨value, gval⟩)(dict)),
  mk-Closed-range1(andid, orng1, orng2)
     → (let mk-Open-range1(relopid1, gexpr1) = orng1,
              mk-Open-range1(relopid2, gexpr2) = orng2 in
          let gval1 = eval-ground-expression(gexpr1)(dict),
              gval2 = eval-ground-expression(gexpr2)(dict) in
          let condval1 = eval-ground-term-opapp(relopid1, ⟨gval1, value⟩)(dict),
              condval2 = eval-ground-term-opapp(relopid2, ⟨value, gval2⟩)(dict) in
          eval-ground-term-opapp(andid, ⟨condval1, condval2⟩)(dict)))
```

type: *Value Condition-item*$_1$ $\rightarrow$ *Entity-dict* $\rightarrow$ *Value*

Objective Evaluate a range condition item w.r.t. a given value.

Parameters

value The value.

cond The condition item.

Result The SDL Boolean evaluation result.

Algorithm

Line 2	Handle the case where the condition item is an open range. Decompose the open range into its contained (relational) operator identifier and ground expression.
Line 3	Evaluate the ground expression.
Line 4	Apply the relational operator to the *value* and the ground expression value.
Line 5	Handle the case where the condition item is a closed range. Decompose it into the AS_1identifier for the SDL predefined Boolean **and** operator and the two contained open ranges.
Line 6-7	Decompose the two open ranges.
Line 8-9	Evaluate the ground expressions contained in the two open ranges.
Line 10-11	Apply each of the two relational operators to the *value* and its corresponding ground expression value.
Line 12	Apply the SDL predefined Boolean **and** operator to the evaluation results of the two open ranges.

6 Construction of *Entity-dict* and Handling of Abstract Data Types

This section contains the functions which build the *Entity-dict* (see the domain definition of *Entity-dict).* The *Entity-dict* is used by almost all processors. The *system* processor builds it by calling *extract-dict* below.

The section is divided into five subsections:

1. The creation of simple self-contained descriptors such as descriptors for variables, signals etc. Also the descriptors for processes and services (i.e. *ProcessDDs* resp. *ServiceDDs*) are created but with empty *Reachability* sets.

   Descriptors are created for entities regardless of whether or not they are defined in a scopeunit included in the consistent subset. The reason for this is that the consistency checks on the data types applies for all scopeunits.

2. Creation of the descriptors for the data type definitions (*TypeDD*). For each scopeunit, this descriptor is created after the descriptors for the sorts *(SortDD)* and syntypes (*SyntypeDD*) are created.
3. Selection of the consistent subset.
4. Creation of the *Reachabilities* for the processes (i.e. creation of all possible communication paths for the processes.)
5. Auxiliary functions for simple information extraction from SDL channel and signal route definitions.

The selection of the consistent subset is made after descriptors for all the entities are constructed, by removing the SDL parts which will not be interpreted. With the modified SDL system as basis, descriptors are constructed again, and *Reachabilities* are constructed. The construction of the *Entity-dict* can be regarded as some intermediate level between the static semantics and the dynamic semantics. The error conditions in this section (checks on the consistent subset and on consistency of the abstract data types) can be regarded as some additional static conditions which are placed in the Dynamic Semantics because:

- Consistency checks on equivalence classes and on mutual exclusion of decision answers cannot easily be expressed in terms of AS_1, i.e. these (static) checks are placed in the Dynamic Semantics because construction of the equivalence classes is required.
- The check on selection of a consistent refinement subset requires that selection of a consistent block subset has already been done.

  To be strict, the selection of the consistent (refinement) subset is not an error condition, since it is not part of an SDL specification, but in order to check its properties, consistency checks are made on the set of block identifiers reflecting the consistent subset.

extract-dict(as_1*tree, blockset, expiredf, terminf*) $\triangleq$ (6.1)

(**let** (as_1*pid*, as_1*null*, as_1*true*, as_1*false*) = *terminf* **in**
let *dict* = [EXPIREDF $\mapsto$ *expiredf,*
PIDSORT $\mapsto$ as_1*pid,*
NULLVALUE $\mapsto$ **mk**-*Ground-term*$_1$(as_1*null*),
TRUEVALUE $\mapsto$ **mk**-*Ground-term*$_1$(as_1*true*),

6 FALSEVALUE $\mapsto$ **mk**-*Ground-term*$_1$(*as*$_1$*false*)] **in**
7 **let** *d'* = *make-system-dict*(*as*$_1$*tree*)(*dict*) **in**
8 **let** *as*$_1$*tree'* = *select-consistent-subset*(*as*$_1$*tree*, *blockset*)(*d'*) **in**
9 **let** *dict'* = *make-system-dict*(*as*$_1$*tree'*)(*dict*) **in**
10 **let** *dict''* = *make-reachabilities*(*as*$_1$*tree'*)(*dict'*) **in**
11 *dict''*)

type: *System-definition*$_1$ *Block-identifier*$_1$-**set** *Is-expiredF Term-information* $\rightarrow$ *Entity-dict*

Objective Construct the *Entity-dict* for a given SDL system.

Parameters

as$_1$*tree* The abstract syntax representation of an SDL system, i.e. an object of the domain *System-definition*$_1$.

blockset The (assumed) consistent subset represented by a set of block identifiers and block substructure identifiers. Although the system scopeunit is also in the consistent subset it is not included in *blockset.*

expiredf A function for comparing SDL time values.

terminf Some AS$_1$ identifiers used by the underlying system.

Result The *Entity-dict* for the part (consistent subset) of the SDL system which will be interpreted.

Algorithm

Line 1 Decompose the *Term-information* (defined in Annex F.2) which contains the *Identifiers*$_1$s of the Pid sort, the Null literal, the True literal and the False literal.

Line 2-6 Create the initial *Entity-dict* wherein the time comparison function and the term information are placed.

Line 7 Construct the *Entity-dict* for the entire SDL system.

Line 8 Remove the parts of the SDL system which will not be interpreted.

Line 9 Construct the *Entity-dict* for the modified SDL system.

Line 10 Construct information about all possible communication paths (the *Reachabilities*) in the modified SDL system and insert this information in the process and service descriptors and the ENVIRONMENT entry of the *Entity-dict.*

Line 11 Return the *Entity-dict.*

6.1 Construction of Descriptors for Simple Objects

make-system-dict(**mk**-*System-definition*$_1$(*snm*, *bset*, , *sigset*, *tp*, *synset*))(*dict*) ≜ (6.1.1)

```
1   (let level = ⟨mk-System-qualifier1(snm)⟩ in
2    let dict' =  dict + [ENVIRONMENT    ↦ {},
3                         SYSTEMLEVEL    ↦ level] in
4    let dict'' =  extract-sortdict (tp, synset, level)(dict') in
5    make-entities(sigset ∪ bset, level)(dict''))
```

type: *System-definition*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the *Entity-dict* for a whole SDL system. Note that enclosed signal route definitions, channel definitions and connections are not dealt with here.

Parameters

snm The system name.

bset The contained block definitions.

sigset The system level signal definitions.

tp The system level data type definition.

synset The system level syntype definitions.

Result The *Entity-dict* for the system.

Algorithm

Line 1 Construct the qualifier denoting the system level.

Line 2 Initialize the ENVIRONMENT entry of the *Entity-dict* to an empty *Reachability set*, and insert the system level qualifier.

Line 4 Insert the system level data information in the *Entity-dict.*

Line 5 Insert information about the other system level definitions in the *Entity-dict.*

make-entities(*entities*, *level*)(*dict*) ≜ (6.1.2)

```
1   if entities = {} then
2      dict
3    else
4      (let entity ∈  entities in
5       let dict' = make-entity(entity, level)(dict) in
6       make-entities(entities \ {entity}, level)(dict'))
```

type: *Decl*$_1$-**set** *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Insert information about definitions into an *Entity-dict.*

Parameters

entities The definitions.

level The qualifier denoting the scope unit level containing the definitions.

Algorithm

Line 1-2 If the set of definitions is empty then do not modify the *Entity-dict.*

Line 4 Pick a definition from the definition set.

Line 5 Insert information about the definition in the *Entity-dict.*

Line 6 Insert information about remaining definitions in the *Entity-dict*.

make-entity(*entity*, *level*)(*dict*) ≜ (6.1.3)

```
1   cases entity:
2    (mk-Signal-definition_1(, , )
3        → dict + make-signal-dict(entity, nil, level),
4      mk-Timer-definition_1(nm, sortlist)
5        → dict + [(mk-Identifier_1(level, nm), SIGNAL) ↦ mk-SignalDD(sortlist, nil)],
6      mk-Variable-definition_1(nm, sort, init, rev)
7        → dict + [(mk-Identifier_1(level, nm), VALUE) ↦ mk-VarDD(, sort, init, rev,)],
8      mk-View-definition_1(nm, sort)
9        → dict + [(mk-Identifier_1(level, nm), VALUE) ↦ mk-ViewDD(sort)],
10     mk-Block-definition_1(, , , , , , ,)
11       → make-block-dict(entity, level)(dict),
12     mk-Process-definition_1(, , , , , , , , , ,)
13       → make-process-dict(entity, level)(dict),
14     mk-Service-definition_1(, , , , , , ,)
15       → make-service-dict(entity, level)(dict),
16     mk-Procedure-definition_1(, , , , , ,)
17       → make-procedure-dict(entity, level)(dict),
18     ⊤ → dict)
```

type: *Decl*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Insert information about a definition into an *Entity-dict*.

Parameters

entity The definition.

level A qualifier denoting the scopeunit containing the definition.

Algorithm Construct the contribution for the entity in hand. Note that a timer is treated as a normal signal.

make-signal-dict(**mk**-*Signal-definition*$_1$(*nm*, *sortlist*, *refinement*), *orev*, *level*) ≜ (6.1.4)

```
1   (let d = [(mk-Identifier_1(level, nm), SIGNAL) ↦ mk-SignalDD(sortlist, orev)] in
2    if refinement = nil then
3      d
4     else
5       (let mk-Signal-refinement_1(subsigset) = refinement in
6        let level' = level ⁀ ⟨mk-Signal-qualifier_1(nm)⟩ in
7        d + merge {make-signal-dict(subsigdef, subsigorev, level') |
8                  mk-Subsignal-definition_1(subsigorev, subsigdef) ∈  subsigset}))
```

type: *Signal-definition*$_1$ [REVERSE] *Qualifier*$_1$ → *Entity-dict*

Objective Make the *Entity-dict* contribution for a signal and for its subsignals if any. Note that a signal descriptor does not tell whether a signal is a subsignal or not. This is due to the fact that this information can be derived from the qualifier of the signal.

Parameters

Signal-definition$_1$ The AS_1 signal definition consisting of

nm The name of the signal.

sortlist The sorts of the values conveyed by the signal.

refinement The signal refinement part.

level A qualifier denoting the scopeunit where the signal is defined.

Algorithm

Line 1 Make the contribution for the signal and

Line 5-7 Make the contributions for the sub-signals with the qualifier denoting the scopeunit which is the signal definition.

make-block-dict(*bdef*, *level*)(*dict*) ≜ (6.1.5)

1 (**let mk**-*Block-definition*$_1$(*bnm*, *pdefs*, *sigdefs*, , , *datatype*, *syntype*, *sub*) = *bdef* **in**
2 **let** *level'* = *level* ⁀ ⟨**mk**-*Block-qualifier*$_1$(*bnm*)⟩ **in**
3 **let** *sortd* = *extract-sortdict*(*datatype*, *syntype*, *level'*)(*dict*) **in**
4 **let** *dict'* = *make-entities*(*sigdefs* ∪ *pdefs*, *level'*)(*sortd*) **in**
5 **if** *sub* = **nil then**
6 *dict'*
7 **else**
8 (**let mk**-*Block-substructure-definition*$_1$(*snm*, *bdefs*, , , *sdefs*, *tp*, *syndefs*) = *sub* **in**
9 **let** *level''* = *level'* ⁀ ⟨**mk**-*Block-substructure-qualifier*$_1$(*snm*)⟩ **in**
10 **let** *sortd'* = *extract-sortdict*(*tp*, *syndefs*, *level''*)(*dict'*) **in**
11 *make-entities*(*bdefs* ∪ *sdefs*, *level''*)(*sortd'*)))

type: *Block-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Insert information about a block definition and its contained definitions into an *Entity-dict*. Note that enclosed signal route definitions, channel definitions and connections are not dealt with here.

Parameters

bdef The block definition.

level The qualifier denoting the level where the block is defined.

Algorithm

Line 1 Decompose the block definition.

Line 2 Construct the qualifier which denotes the level of the block.

Line 3 Update the *Entity-dict* to include the data defined in the block.

Line 4 Update the *Entity-dict* to include the signals (*sigdefs*) and processes (*pdefs*) defined in the block

Line 5 If no block substructure is specified then the updated *Entity-dict* is returned.

Line 8 Decompose the block substructure.

Line 9 Construct the qualifier which denotes the level of the block substructure.

Line 10 Update the *Entity-dict* to include the data definitions defined in the block substructure.

Line 11 Update the *Entity-dict* to include the blocks (*bdefs*) and signals (*sdefs*) defined in the block substructure.

make-process-dict(*pdef*, *level*)(*dict*) $\triangleq$ (6.1.6)

1 (**let mk**-*Process-definition*$_1$(*nm*, *inst*, *f*, *pset*, *sigset*, *tp*, *synset*, *vset*, , *tset*, *grordec*) = *pdef* **in**
2 **let mk**-*Number-of-instances*$_1$(*init*, *maxi*) = *inst* **in**
3 **let** *pid* = **mk**-*Identifier*$_1$(*level*, *nm*),
4 *level'* = *level* $\frown$ ⟨**mk**-*Process-qualifier*$_1$(*nm*)⟩ **in**
5 **let** (*parmdds*, *parmd*) = *make-process-formal-parameters*(*f*, *level'*) **in**
6 **let** *dict'* = *extract-sortdict*(*tp*, *synset*, *level'*)(*dict* + *parmd*) **in**
7 **let** *dict''* = *make-entities*(*pset* ∪ *sigset* ∪ *vset* ∪ *tset*, *level'*)(*dict'*) **in**
8 (**is**-*Process-graph*$_1$(*grordec*)
9 → (**let** *grordec'* = *check-graph*(*grordec*, *level'*)(*dict'*) **in**
10 *dict''* + [(*pid*, PROCESS) ↦ **mk**-*Process-DD*(*parmdds*, *init*, *maxi*, *grordec'*, { })]),
11 **is**-*Service-decomposition*$_1$(*grordec*)
12 → (**let mk**-*Service-decomposition*$_1$(*servset*, ,) = *grordec* **in**
13 **let** *dict'''* = *make-entities*(*servset*, *level'*)(*dict'*) **in**
14 *dict'''* + [(*pid*, PROCESS) ↦ **mk**-*Process-DD*(*parmdds*, *init*, *maxi*, **nil**, { })])))

type: *Process-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Insert information about a process definition and its contained definitions into an *Entity-dict*. Note that enclosed signal route definitions and connections are not dealt with here.

Parameters

pdef The process definition.

level A qualifier denoting the scopeunit where the process is defined.

Algorithm

Line 1 Decompose the process definition.

Line 2 Extract the initial number of instances (*init*) and the maximum number of instances (*maxi*).

Line 3 Construct the identifier for the process definition.

Line 4 Construct the qualifier denoting the scopeunit which is the process definition.

Line 5 Construct the formal parameter descriptors and *Entity-dict* contributions for the process formal parameters.

Line6 Make the *Entity-dict* which is updated with information about the data definitions in the process.

Line 7 Make the contributions for the contained procedure definitions (*pset*), signal definitions (*sigset*), variable definitions (*vset*) and timer definitions (*tset*)

Line 8 Handle the case where the process is *not* decomposed into services.

Line 9 Check the wellformedness of the process graph. The function either returns the process graph unchanged or performs an **exit**.

Line 10 Update the constructed *Entity-dict* with the descriptor for the process itself. Note that, at this stage, the *Reachability* set for the process is empty.

Line 11 Handle the case where the process is decomposed into services.

Line 12 Decompose the service decomposition.

Line 13 Update the *Entity-dict* with information about the services.

Line 14 Update the constructed *Entity-dict* with the descriptor for the process itself. The process graph field in the process descriptor is set to nil to indicate that the process is decomposed into services. Note that, at this stage, the *Reachability* set for the process is empty.

make-process-formal-parameters(*parml*, *level*) ≜ (6.1.7)

1 (⟨**mk**-*Identifier*$_1$(*level*, *varnm*) |
2 1 ≤ *i* ≤ **len** *parml* ∧ **mk**-*Process-formal-parameter*$_1$(*varnm*,) = *parml*[*i*]⟩,
3 [(**mk**-*Identifier*$_1$(*level*, *varnm*), VALUE) ↦ **mk**-*VarDD*(, *sortref*, **nil**, **nil**,) |
4 1 ≤ *i* ≤ **len** *parml* ∧ **mk**-*Process-formal-parameter*$_1$(*varnm*, *sortref*) = *parml*[*i*]])

type: *Process-formal-parameter*$_1$* *Qualifier*$_1$ → *ParameterDD** *Entity-dict*

Objective Construct the formal parameter descriptors and *Entity-dict* contribution for a list of process formal parameters.

Parameters

parml The list of process formal parameters.

level The qualifier denoting the process level.

Algorithm

Line 1-2 Construct the list of formal parameter descriptors. Each formal parameter descriptor is simply the identifier of the formal parameter variable.

Line 3-4 Construct the *Entity-dict* contribution for the formal parameters. Note that they are treated as normal variables.

make-service-dict(*servdef*, *level*)(*dict*) ≜ (6.1.8)

1 (**let mk**-*Service-definition*$_1$(*nm*, *pset*, *tp*, *synset*, *vset*, , *tset*, *graph*) = *servdef* **in**
2 **let** *servid* = **mk**-*Identifier*$_1$(*level*, *nm*),
3 *level'* = *level* ⁀ ⟨**mk**-*Service-qualifier*$_1$(*nm*)⟩ **in**
4 **let** *dict'* = *extract-sortdict*(*tp*, *synset*, *level'*)(*dict*) **in**
5 **let** *dict''* = *make-entities*(*pset* ∪ *vset* ∪ *tset*, *level'*)(*dict'*) **in**
6 **let** *graph'* = *check-graph*(*graph*, *level'*)(*dict*) **in**
7 *dict''* + [(*servid*, SERVICE) ↦ **mk**-*ServiceDD*(*graph'*, , {})])

type: *Service-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Insert information about a service and its contained definitions into an *Entity-dict*.

Parameters

servdef The service definition.

level The qualifier denoting the level at which the service is defined.

Algorithm

Line 1 Decompose the service definition.

Line 2 Construct the identifier of the service.

Line 3 Construct the qualifier denoting the service level.

Line 4 Update the *Entity-dict* to include information about the data defined in the service.

Line 5 Update the *Entity-dict* to include information about the procedures (*pset*), variables (*vset*) and timers (*tset*) defined in the service.

Line 6 Check the wellformedness of the graph. The function *check-graph* either returns the service graph unchanged or performs an **exit**.

Line 7 Update the constructed *Entity-dict* with the descriptor for the service itself. Note that, at this stage, the *Reachability* set for the service is empty.

make-procedure-dict(*procdef*, *level*)(*dict*) ≜ (6.1.9)

```
1    (let mk-Procedure-definition1(nm, fp, pset, tp, sset, vset, graph) = procdef in
2     let pid = mk-Identifier1(level, nm),
3         level' = level ⁀ ⟨mk Procedure-qualifier1(nm)⟩ in
4     let (parmddl, fdict) = make-procedure-formal-parameters(fp, level') in
5     let dict' = extract-sortdict(tp, sset, level')(dict + fdict) in
6     let dict'' = make-entities(pset ∪ vset, level')(dict') in
7     let graph' = check-graph(graph, level')(dict'') in
8     dict'' + [(pid, PROCEDURE) ↦ mk-ProcedureDD(parmddl, graph')])
```

type: *Procedure-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Insert information about a procedure and its contained definitions into an *Entity-dict.*

Parameters

procdef The procedure definition.

level The qualifier denoting the scopeunit where the procedure is defined.

Algorithm

Line 1 Decompose the procedure definition.

Line 2 Construct the identifier for the procedure.

Line 3 Construct the qualifier denoting the procedure scopeunit.

Line 4 Construct the procedure formal parameter descriptors for the procedure and the *Entity-dict* contribution for the formal parameters.

Line 5 Update the *Entity-dict* with information about the data definitions in the procedure.

Line 6 Update the *Entity-dict* with information about the contained procedure definitions (*pset*) and variable definitions (*vset*).

Line 7 Check the wellformedness of the procedure graph. The function *check-graph* either returns the procedure graph unchanged or performs an **exit**.

Line 8 Update the constructed *Entity-dict* with the descriptor for the procedure itself.

make-procedure-formal-parameters(*parml*, *level*) ≜ (6.1.10)

```
(⟨cases parml[i]:
   (mk-In-parameter1(varnm,)
       → mk-InparmDD(mk-Identifier1(level, varnm)),
    mk-Inout-parameter1(varnm,)
       → mk-InoutparmDD(mk-Identifier1(level, varnm))) |
 1 ≤ i ≤ len parml⟩,
[(mk-Identifier1(level, varnm), VALUE) ↦ mk-VarDD(, sortref, nil, nil,) |
  mk-In-parameter1(varnm, sortref) ∈ elems parml])
```

type: *Procedure-formal-parameter*$_1$* *Qualifier*$_1$ → *FormparmDD** *Entity-dict*

Objective Construct the formal parameter descriptors and *Entity-dict* contribution for a list of procedure formal parameters.

Parameters

parml The list of procedure formal parameters.

level The qualifier denoting the procedure level.

Algorithm

Line 1-6 Construct the list of formal parameter descriptors.

Line 7-8 Construct the *Entity-dict* contribution for the (**in**) formal parameters. Note that they are treated as normal variables. No entries in *Entity-dict* are made for the **in/out** formal parameters.

check-graph(*graph, level*)(*dict*) ≜ (6.1.11)

```
1   (¬is-wf-assignments(graph, level)(dict)
2      → exit("§5.4.3: Ground expression in assignment statement is out of range"),
3    ¬is-wf-decision-answers(graph, level)(dict)
4      → exit("§2.7.5: Answers in decision actions are not mutually exclusive"),
5    T → graph)
```

type: (*Process-graph*$_1$ | *Service-graph*$_1$ | *Procedure-graph*$_1$) *Qualifier*$_1$ → *Entity-dict*
→ (*Process-graph*$_1$ | *Service-graph*$_1$ | *Procedure-graph*$_1$)

Objective Check the wellformedness of a process, service or procedure graph, i.e. perform a range check on each ground expression constituting the right hand side of an assignment statement, and check that no decision node contains overlapping answers.

Parameters

graph The process, service or procedure graph to be checked.

level The qualifier denoting the process/service/procedure level.

Result If the graph is wellformed, it is returned unchanged, otherwise the function performs an **exit**.

Algorithm

Line 1-2 Perform a range check on each ground expression constituting the right hand side of an assignment statement.

Line 3-4 Check that no decision node contains overlapping answers.

Line 5 Return the graph unchanged.

is-wf-assignments(*graph, level*)(*dict*) ≜ (6.1.12)

```
1   (let (startnode, stateset) = decomp-graph(graph) in
2    (let trans = decomp-start-node(startnode) in
3     is-wf-transition-assignments(trans, level)(dict)) ∧
4    (∀mk-State-node_1(, , inputs, spontrs) ∈ stateset)
5     ((∀mk-Input-node_1(, , trans) ∈ inputs)(is-wf-transition-assignments(trans, level)(dict)) ∧
6      (∀mk-Spontaneous-transition_1(trans) ∈ spontrs)(is-wf-transition-assignment(trans, level)(dict))))
```

type: (*Process-graph*$_1$ | *Service-graph*$_1$ | *Procedure-graph*$_1$) *Qualifier*$_1$ → *Entity-dict*
→ *Bool*

Objective Perform a range check on each ground expression which constitutes the right hand side of some assignment statement in a process, service or procedure graph.

Parameters

graph The process, service or procedure graph.

level The qualifier denoting the process/service/procedure level.

Result **true** if success, else **false**.

Algorithm

Line 1 Decompose the graph into its start node and state node set.

Line 2 Obtain the transition contained in the start node.

Line 3 No ground expression constituting the right hand side in an assignment statement in the start transition may be out of range.

Line 4 For each state it must hold that

Line 5 for each input transition no assignment statement may have an out-of-range ground expression as its right hand side,

Line 6 and for each spontaneous transition no assignment statement may have an out-of-range ground expression as its right hand side.

is-wf-transition-assignments(**mk**-*Transition*$_1$(*actl*, *termordec*), *level*)(*dict*) $\triangleq$ (6.1.13)

1 (∀*act* ∈ **elems** *actl*)
2 (**is**-*Task-node*$_1$(*act*) ⊃ **is**-*wf-task-node*(*act*, *level*)(*dict*)) ∧
3 (**is**-*Decision-node*$_1$(*termordec*) ⊃
4 (**let mk**-*Decision-node*$_1$(, *answerset*, *elsetrans*) = *termordec* **in**
5 (∀**mk**-*Decision-answer*$_1$(, *trans*) ∈ *answerset*)
6 (*is-wf-transition-assignments*(*trans*, *level*)(*dict*)) ∧
7 (*elsetrans* ≠ **nil** ⊃ *is-wf-transition-assignments*(*s-Transition*$_1$(*elsetrans*), *level*)(*dict*))))

type: *Transition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Bool*

Objective Check that no assignment in a transition has an out-of-range ground expression as its right hand side.

Parameters

actl,*termordec* The action list and the terminator/(outermost) decision node in the transition.

level The qualifier denoting the surrounding scope unit.

Result **true** if success, else **false**.

Algorithm

Line 1-2 Check all task nodes in the action list of the transition.

Line 3 If the terminating action of the transition is a decision node, the checks in the lines below should be performed.

Line 4 Decompose the decision node.

Line 5-6 Check the transition contained in each decision answer.

Line 7 If the **else** answer is present, then check its contained transition.

is-wf-task-node(**mk**-*Task-node*$_1$(*asgnortxt*), *level*)(*dict*) $\triangleq$ (6.1.14)

1 **cases** *asgnortxt*:
2 (**mk**-*Assignment-statement*$_1$(*varid*, *expr*)
3 → **is**-*Ground-expression*$_1$(*expr*) ⊃
4 (**let** *dict'* = *dict* + [SCOPEUNIT ↦ *level*] **in**
5 **let mk**-*VarDD*(, *sortref*, , ,) = *dict'*((*varid*, VALUE)),
6 *exprval* = *eval-expression*(*expr*)(*dict'*) **in**
7 *range-check*(*sortref*, *exprval*)(*dict*)),
8 **mk**-*Informal-text*$_1$()
9 → **true**)

type: *Task-node*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Bool*

Objective If a task node contains an assignment statement, then check that its right hand side is not a ground expression which is out of range.

Parameters

asgnortxt The assignment statement or informal text in the task node.

level The qualifier denoting the surrounding scope unit.

Result **true** if success, else **false**.

Algorithm

Line 2 Consider the case where the contents of the task node is an assignment statement.

Line 3 If the right hand side of the assignment statement is not a ground expression, the assignment statement is wellformed.

Line 4-6 Insert the scopeunit in the *Entity-dict*, look up the sort or syntype of the left hand side variable, and evaluate the right hand side.

Line 7 Perform the range check.

Line 8-9 If the contents of the task node is an informal text, the task node is wellformed.

is-wf-decision-answers(*graph, level*)(*dict*) $\triangleq$ (6.1.15)

1 (**let** (*startnode*, *stateset*) = *decomp-graph*(*graph*) **in**
2 (**let** *trans* = *decomp-start-node*(*startnode*) **in**
3 *is-wf-transition-answers*(*trans, level*)(*dict*)) $\wedge$
4 ($\forall$**mk**-*State-node*$_1$(, , *inputs*, *spontrs*) $\in$ *stateset*)
5 (($\forall$**mk**-*Input-node*$_1$(, , *trans*) $\in$ *inputs*)(*is-wf-transition-answers*(*trans, level*)(*dict*)) $\wedge$
6 ($\forall$**mk**-*Spontaneous-transition*$_1$(*trans*) $\in$ *spontrs*)(*is-wf-transition-answers*(*trans, level*)(*dict*))))

type: (*Process-graph*$_1$ | *Service-graph*$_1$ | *Procedure-graph*$_1$) *Qualifier*$_1$ $\rightarrow$ *Entity-dict*
$\rightarrow$ *Bool*

Objective Check that the answers in a decision action of a process, service or procedure graph are mutually exclusive.

Parameters

graph The process, service or procedure graph.

level The qualifier denoting the process/service/procedure level.

Result **true** if success, else **false**.

Algorithm

Line 1 Decompose the graph into its start node and state node set.

Line 2 Obtain the transition contained in the start node.

Line 3 No decision node in the start transition may contain overlapping answers.

Line 4 For each state it must hold that

Line 5 for each input transition no decision node contains overlapping answers,

Line 6 and for each spontaneous transition no decision node contains overlapping answers.

is-wf-transition-answers(**mk**-*Transition*$_1$(, *termordec*), *level*)(*dict*) ≜ (6.1.16)

```
1   is-Decision-node_1(termordec) ⊃
2   (let mk-Decision-node_1(, answerset, elsetrans) = termordec in
3    (∀mk-Decision-answer_1(, trans) ∈ answerset)
4     (is-wf-transition-answers(trans, level)(dict)) ∧
5    (elsetrans ≠ nil ⊃ is-wf-transition-answers(s-Transition_1(elsetrans), level)(dict)) ∧
6    (∀answer1 ∈ answerset)
7     ((∀answer2 ∈ answerset \ {answer1})
8       ((let mk-Decision-answer_1(rngortxt1,) = answer1,
9             mk-Decision-answer_1(rngortxt2,) = answer2,
10            dict' = dict + [SCOPEUNIT ↦ level] in
11        is-Range-condition_1(rngortxt1) ∧ is-Range-condition_1(rngortxt2) ⊃
12        ranges-not-overlapping(rngortxt1, rngortxt2) (dict')))))
```

type: *Transition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Bool*

Objective Check that no decision action in a transition contains overlapping answers.

Parameters

termordec The terminator or (outermost) decision node in the transition.

level The qualifier denoting the surrounding scopeunit.

Result **true** if success, else **false**.

Algorithm

Line 1 The condition is **true** if the terminating action of the transition is not a decision node.

Line 2 Decompose the decision node into a set of answers and an optional **else** answer.

Line 3-4 Check that no decision node in the answers contains overlapping answers.

Line 5 If the **else** answer is present then check that no contained decision node contains overlapping answers.

Line 6-7 For any two different decision answers in the decision node lines 8-12 must hold.

Line 8-9 Obtain the answer range conditions from the two decision answers.

Line 10 Insert the scope unit level of the decision node into the *Entity-dict* in order to enable "static evaluation" of the range conditions.

Line 11-12 If both answer range conditions are really range conditions (i.e. none of them is an informal text) they are not allowed to overlap.

ranges-not-overlapping(*rngcond*1, *rngcond*2)(*dict*) ≜ (6.1.17)

```
1   (let-sort = sort-of-range-condition(rngcond1)(dict) in
2    (∀value ∈ values-of-sort(sort)(dict))
3     ((trap exit( ) with true in
4       let answerval1 = eval-range-condition(value, rngcond1)(dict),
5           answerval2 = eval-range-condition(value, rngcond2)(dict) in
6       answerval1 = dict(FALSEVALUE) ∨ answerval2 = dict(FALSEVALUE))))
```

type: *Range-condition*$_1$ *Range-condition*$_1$ → *Entity-dict* → *Bool*

Objective Check that two given range conditions do not overlap.

Parameters

*rngcond*1 The first range condition.

*ragcond*2 The second range condition.

Result **true** if success, else **false**.

Algorithm

Line 1 Obtain the sort of the values expected by the range conditions. If (some of) the contained condition items expect a syntype the parent sort of this is obtained.

Line 2-6 The range conditions are disjoint exactly if there exists no value for which both range conditions are True. For each value the two range conditions are "statically" evaluated (line 4-5) and it is tested that at least one of the evaluation results is False (line 6). Any exit caused by range checks for syntypes during evaluation of the range conditions is trapped (line 3) since range checks for syntypes should not be applied until the decision is interpreted.

6.2 Handling of Abstract Data Types

This section contains the functions for handling of abstract data types. The entry functions are:

extract-sortdict which is applied during the construction of *Entity-dict* and which creates the type descriptors, sort descriptors, syntype descriptors, literal descriptors and operator descriptors.

values-of-sort which is used to obtain all values of a given sort.

reduce-term which is used to obtain the ground term which has been chosen (during the creation of the *Entity-dict)* to represent the equivalence class to which a given ground term belongs.

sort-of-range-condition which is used to obtain the sort of values which is expected by a range condition. If (some of) the condition items of the range condition expect a syntype the corresponding parent sort is returned.

sort-or-parent-sort which obtains the parent sort of a syntype. If a sort identifier is given to the function this sort identifier is returned.

6.2.1 Entry Functions

extract-sortdict(*typedef*, *syndefs*, *level*)(*dict*) ≜ (6.2.1.1)

```
(let mk-Data-type-definition1(sorts, signatureset, eqs) = typedef in
 let literald = [(id, VALUE) ↦ mk-OperatorDD(⟨⟩, result) |
                    mk-Literal-signature1(nm, result) ∈ signatureset ∧
                    id = mk-Identifier1(level, nm)],
     operatord = [(id, VALUE) ↦ mk-OperatorDD(arglist, result) |
                    mk-Operator-signature1(nm, arglist, result) ∈ signatureset ∧
                    id = mk-Identifier1(level, nm)],
     sortd = [(id, SORT) ↦ mk-SortDD() |
                 nm ∈ sorts ∧ id = mk-Identifier1(level, nm)],
     syntyped = [(id, SORT) ↦ mk-SyntypeDD(parsort, rngcond) |
                    mk-Syn-type-definition1(nm, parsort, rngcond) ∈ syndefs ∧
                    id = mk-Identifier1(level, nm)],
     dict' = dict + literald + operatord + sortd + syntyped in
 let equations = collect-all-equations(eqs, level)(dict'),
     sortmap = make-sortmap(sorts, equations, level)(dict'),
     trmap = make-term-reduce-map(sortmap, level)(dict'),
     dict'' = dict' + [(level, TYPE) ↦ mk-TypeDD(trmap, sormap, equations)] in
 (¬is-wf-literals(level)(dict'')
     → exit("§5.3.1.7: Literal is equivalent to the error term"),
  ¬is-wf-values(level)(dict'')
     → exit("§5.2.1: Generation or reduction of equivalence classes of the enclosing scope unit"),
  T → dict''))
```

type: *Data-type-definition*$_1$ *Syn-type-definition*$_1$-**set** *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Update *Entity-dict* to contain the descriptors for the data definitions (i.e. data type, sorts, syntypes, literals and operators) at a given scope unit level.

Parameters

typedef The data type definition.

syndefs The syntype definitions.

level The level on which they are defined.

Result The updated *Entity-dict.*

Algorithm

Line 1 Decompose the data type definition into its contained sorts, literal and operator signatures, and equations.

Line 2-3 Construct the descriptors for all the literals in the data type definition. They are considered as operators without any arguments.

Line 5-7 Construct the descriptors for all the operators defined in the data type definition.

Line 8-9 Construct the descriptors for all the sorts defined in the data type definition.

Line 10-12 Construct the descriptors for the syntype definitions.

Line 13 Add the above constructed descriptors to the *Entity-dict.*

Line 14 Obtain the set of all equations which apply at this scope unit level.

Line 15 Use the equations to construct the *Sortmap* which applies at this scope unit level.

Line 16 Use the *Sortmap* to construct the *Term-reduce-map* which maps each equivalence class of this scope unit level to a canonical ground term. The choice of these canonical ground terms is made by the function *make-term-reduce-map* according to some criteria which will be explained in the section where *make-term-reduce-map.*

Line 17 Insert a descriptor for the data type definition into the *Entity-dict.* The qualifier of the enclosing scope unit is used as key for looking up this descriptor because a data type definition has no name.

Line 18-19 Check that no literal is equal to the **error**! term.

Line 20-21 Check that no equivalence classes of the scope unit enclosing this one are unified, and that no new equivalence classes are added to sorts visible in the scope unit enclosing this one.

Line 22 Return the updated *Entity-dict.*

values-of-sort(*sortid*)(*dict*) $\triangleq$ (6.2.1.2)

1 (**let** *sortlevel* = **s**-*Qualifier*$_1$(*sortid*) **in**
2 **let mk**-*TypeDD*(*trmap*, ,) = *dict*((*sortlevel*, TYPE)) **in**
3 {*val* ∈ **rng** *trmap* \ {**mk**-*Error-term*$_1$()} | *is-of-this-sort*(*sortid, val*)(*dict*)})

type: *Sort-identifier*$_1$ → *Entity-dict* → *Value*-**set**

Objective Obtain the set of all values belonging to a given sort.

Parameters

sortid The identifier of the sort.

Result The set of values of the sort.

Algorithm

Line 1 Obtain the qualifier of the sort.

Line 2 Use this qualifier to look up the type descriptor for the scope unit where the sort is defined.

Line 3 The range of the *Term-reduce-map* of the scope unit contains all values of all sorts visible in that scope unit, and the error term. Exclude the error term and select those values which belong to the given sort.

reduce-term(*term*, *level*)(*dict*) ≜ (6.2.1.3)

1 (**let mk**-*TypeDD*(*trmap*, ,) = *dict*((*level*, TYPE)) **in**
2 **let** *class* ∈ **dom** *trmap* **be s.t.** *term* ∈ *class* **in**
3 **let** *term'* = *trmap*(*class*) **in**
4 **if is**-*Error-term*$_1$(*term'*) **then**
5 **exit**("§5.3.1.7: Expression, term or value is equivalent to the error term")
6 **else**
7 *term'*)

type: *Ground-term*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Value*

Objective Given a ground term, obtain the canonical ground term which has been chosen to represent its equivalence class in the rest of the system.

Parameters

term The ground term.

level The scope unit level at which the ground term has been built.

Result The canonical ground term.

Algorithm

Line 1 Obtain the *Term-reduce-map* for the scope unit level.

Line 2 Select the equivalence class which contains the ground term.

Line 3 Obtain the canonical ground term from the *Term-reduce-map*.

Line 4-5 It is an error if the ground term is equivalent to the error term.

Line 7 Return the canonical ground term.

sort-of-range-conditiont(**mk**-*Range-condition*$_1$(, *cset*))(*dict*) ≜ (6.2.1.4)

1 (**let** *condit* ∈ *cset* **in**
2 **let** *relopid* = **cases** *condit*:
3 (**mk**-*Open-range*$_1$(*op*,)
4 → *op*,
5 **mk**-*Closed-range*$_1$(, , **mk**-*Open-range*$_1$(*op*,))
6 → *op*) **in**
7 **let mk**-*OperatorDD*(*sortlist*,) = *dict*((*relopid*, VALUE)) **in**
8 *sort-or-parent-sort*(*sortlist*[1])(*dict*))

type: *Range-condition*$_1$ → *Entity-dict* → *Sort-identifier*$_1$

Objective Obtain the sort of the values which are expected by a range condition. If (some of) the condition items in the range condition expect a syntype the parent sort of this is returned.

Parameters

cset The condition items of the range condition.

Result The sort expected by the range condition.

Algorithm

Line 1 Select an arbitrary condition item from the range condition. The static conditions on a range condition ensure that all its condition items expect the same sort/parent sort.

Line 2-6 If the chosen condition item is an open range its relational operator is extracted (line 3-4). If it is a closed range the relational operator of its second open range is extracted (line 5-6).

Line 7 Look up the argument sort list of the operator.

Line 8 The first argument sort/syntype of the operator is the one expected by the condition item. If the argument sort/syntype is a syntype its parent sort is returned.

sort-or-parent-sort(*sortref*)(*dict*) $\triangleq$ (6.2.1.5)

```
1   cases dict((sortref, SORT)):
2    (mk-SortDD()              → sortref,
3     mk-SyntypeDD(parsort,)   → parsort)
```

type: $\textit{Sort-reference-identifier}_1 \rightarrow \textit{Entity-dict} \rightarrow \textit{Sort-identifier}_1$

Objective If a given sort/syntype is a syntype then obtain its parent sort.

Parameters

sortref The sort/syntype identifier.

Result The sort/parent sort identifier.

Algorithm

Line 1 Look up the sort/syntype in the *Entity-dict*.

Line 2 If the sort/syntype is a sort it is returned.

Line 3 If the sort/syntype is a syntype its parent sort is returned.

6.2.2 Equation Collection

collect-all-equations(*eqs*, *level*)(*dict*) ≜ (6.2.2.1)

```
1   (let sureqs =
2       if len level = 1 then
3         {}
4       else
5         (let surlevel = ⟨level[i ] | 1 ≤  i < len level⟩ in
6           s-Equations1(dict((surlevel, TYPE)))) in
7    eqs ∪ sureqs
```

type: *Equations*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Equations*$_1$

Objective Obtain the set of all equations which apply at a given scope unit level.

Parameters

eqs The equations defined in this scope unit.

level This scope unit level.

Result All equations which apply at this scope unit level.

Algorithm

Line 1-6 Obtain the equations visible in the enclosing scope unit. If the current scope unit is the system level the “enclosing” equation set is empty.

Line 7 The equations applying at this scope unit levels are the ones defined at this level together with the “enclosing” ones.

6.2.3 Equivalence Class Generation and Equation Evaluation

make-sortmap(*sorts*, *equations*, *level*)(*dict*) ≙ (6.2.3.1)

```
1   (let sursmap =
2       if len level = 1 then
3           []
4       else
5           (let surlevel = ⟨level[i] | 1 ≤ i < len level⟩ in
6            s-Sortmap(dict((surlevel, TYPE)))) in
7    let sortset = {mk-Identifier1(level, nm) | nm ∈ sorts} ∪ dom sursmap in
8    let initial-sortmap = [sort ↦ make-equivalence-classes(sort)(dict) | sort ∈ sortset] in
9    eval-equations(initial-sortmap, equations)(dict))
```

type: $Sorts_1$ $Equations_1$ $Qualifier_1$ → *Entity-dict* → *Sortmap*

Objective Construct the *Sortmap* which applies at a given scope unit level.

Parameters

sorts The sorts defined in this scope unit.

equations The equations visible in this scope unit.

level The qualifier for this scope unit.

Result The *Sortmap*.

Algorithm

Line 1-6 Obtain the sort map which applies at the enclosing scope unit level. If the current scope unit is the system level the “enclosing” sort map is empty.

Line 7 Obtain the set of all sorts visible in this scope unit.

Line 8 Construct the initial sort map where each possible ground term is in its own equivalence class.

Line 9 Construct equivalence classes according to the equations.

make-equivalence-classes(*sort*)(*dict*) ≙ (6.2.3.2)

```
1   {{term} | term ∈ Ground-term1 ∧ is-of-this-sort(sort, term)(dict)} ∪ {{mk-Error-term1 ()}}
```

type: $Sort\text{-}identifier_1$ → *Entity-dict* → *Term-class*-**set**

Objective For a given sort, construct the initial set of equivalence classes where each ground term is contained in its own equivalence class.

Parameters

sort The identifier of the sort.

Result The initial set of equivalence classes.

Algorithm Select all ground terms which belong to the given sort and put each one in its own equivalence class. An equivalence class containing the error term only is also included.

is-of-this-sort(*sortref*, *t*)(*dict*) ≜ (6.2.3.3)

```
1   (let sortid = sort-or-parent-sort(sortref)(dict),
2        mk-Ground-term1(term) = t in
3    (is-Identifier1(term)
4        → (let entry = (term, VALUE) in
5            entry ∈ dom dict ∧ is-OperatorDD(dict(entry)) ∧
6            (let mk-OperatorDD(sortlist, result) = dict(entry) in
7             sortlist = 〈〉 ∧ result = sortid)),
8     is-Conditional-term1(term)
9        → false,
10    T → (let (opid, arglist) = term in
11           let entry = (opid, VALUE) in
12           entry ∈ dom dict ∧ is-OperatorDD(dict(entry)) ∧
13           (let mk-OperatorDD(sortlist, result) = dict(entry) in
14            len arglist = len sortlist ∧
15            (∀i ∈ ind arglist)(is-of-this-sort(sortlist[i], arglist[i])(dict)) ∧
16            sort-or-parent-sort(result)(dict) = sortid))))
```

type: *Sort-reference-identifier*$_1$ *Ground-term*$_1$ → *Entity-dict* → *Bool*

Objective Test whether a given ground term belongs to a given sort. If the sort given is actually a syntype its parent sort is used.

Parameters

sortref The identifier of the sort/syntype.

t The ground term.

Result **true** if the ground term belongs to the given sort, else **false**.

Algorithm

Line 1 Obtain the sort/parent sort of the sort/syntype.

Line 2 Get the “contents” of the ground term.

Line 3 If the term is an identifier then

Line 5 the identifier must be found in *Entity-dict* as a (literal) operator,

Line 7 the argument list of which is empty in the descriptor, and the result sort must be appropriate according to the result sort found in the descriptor.

Line 8-9 If the term is a conditional term then it does not represent a value (but the consequence and alternative in the conditional term may do).

Line 10 If the term is an operator term then

Line 12 the operator must be found in *Entity-dict*,

Line 14 the number of arguments in the descriptor must be equal to the number of arguments present in the term,

Line 15 each argument term must be of the appropriate sort according to the argument list found in the descriptor,

Line 16 and the result sort must be appropriate according to the result sort found in the descriptor.

eval-equations(*sortmap*, *equations*)(*dict*) ≜ (6.2.3.4)

```
1     (let trueterm = dict(TRUEVALUE),
2          falseterm = dict(FALSEVALUE) in
3      let quanteq = {eq ∈ equations | is-Quantified-equations1(eq)},
4          rest = equations \ quanteq in
5      let unquant = union {eval-quantified-equation(sortmap, eq) | eq ∈ quanteq} in
6      let rest′ = expand-conditional-term-in-equations(rest ∪ unquant, trueterm, falseterm) in
7      let rest″ =
8          union {if is-Conditional-equation1(eq)
9                 then expand-conditional-term-in-conditions({eq}, trueterm, falseterm)
10                else {eq} | eq ∈ rest′} in
11     let unquanteqs = {eq ∈ rest″ | is-Unquantified-equation1(eq)},
12         condeqs = {eq ∈ rest″ | is-Conditional-equation1(eq)} in
13     let sortmap′ = eval-unquantified-equations(sortmap, unquanteqs) in
14     eval-conditional-equations(sortmap′, condeqs))
```

type: *Sortmap Equations*$_1$ → *Entity-dict* → *Sortmap*

Objective Reduce the number of equivalence classes for the sorts visible in a given scopeunit according to a set of equations.

Parameters

sortmap A *Sortmap* containing the equivalence classes which are to be reduced

equations A set of equations.

Result The modified *Sortmap.*

Algorithm

Line 1-2 Extract the AS$_1$ representations for the Boolean literals True and False from *Entity-dict.*

Line 3 Extract the equations which are quantified.

Line 5 Turn the set of quantified equations into a set of unquantified equations

Line 6 Turn all the conditional terms occurring in the modified set of equations (except for those occurring in the conditions of conditional equations) into a set of conditional equations.

Line 7-10 Turn all the conditional equations which contain conditional terms in the condition set, into a set of conditional equations without any conditional terms in the conditions (see example in the text following the function *expand-conditional-term-in-conditions*).

Line 11-12 Split the resulting set of equations (*rest"*) into a set of unquantified equations and a set of conditional equations.

Line 13 Modify *sortmap* in accordance with the set of unquantified equations.

Line 14 Return the *Sortmap* which is *sortmap* modified in accordance with the set of conditional equations.

eval-quantified-equation(*sortmap*, *quanteqs*) ≜ (6.2.3.5)

```
1   (let mk-Quantified-equations1(nmset, sortid, equations) = quanteqs in
2    let nm ∈ nmset in
3    let mk-Identifier1(level, snm) = sortid in
4    let valueid = mk-Identifier1(level ⁀ ⟨mk-Sort-qualifier1(snm)⟩, nm) in
5    let allterms = union sortmap(sortid) \ {mk-Error-term1()} in
6    let equations′ = union{union {insert-term(sortmap, eq, valueid, term) | term ∈ allterms} |
7                                  eq ∈ equations} in
8    if nmset = {nm} then
9      equations′
10    else
11     (let quanteq = mk-Quantified-equations1(nmset \ {nm}, sortid, equations′) in
12       eval-quantified-equation(sortmap, quanteq)))
```

type: *Sortmap* $\mathit{Quantified\text{-}equations}_1 \to \mathit{Equations}_1$

Objective Expand a quantified equation into a set of unquantified equations.

Parameters

sortmap The Sortmap of the enclosing data type definition, wherein the terms (still) are in different equivalence classes

quanteqs The quantified equations.

Result The resulting set of unquantified equations.

Algorithm

Line 2 Take one of the value names in the quantified equation.

Line 4 Make the value identifier corresponding to the value name

Line 5 Make a set (*allterms*) consisting of all possible terms (except the $\mathit{Error\text{-}term}_1$) for the quantifying sort.

Line 6-7 Construct a set of unquantified equations from the set of equations contained in the quantified equation by replacing the value identifier in the set of equations by every term in *allterms*.

Line 8 If every value name has been replaced in the equations then return the equations (*equations′*) else

Line 11-12 Do the same for the rest of the value names in the quantified equation.

insert-term(*sortmap*, *equation*, *vid*, *term*) ≜ (6.2.3.6)

```
cases equation:
 (mk-Unquantified-equation1(term1, term2)
     → {mk-Unquantified-equation1(insert-term-in-term(term1, vid, term),
                                  insert-term-in-term(term2, vid, term))},
  mk-Quantified-equations1(, ,)
     → (let equations = eval-quantified-equation(sortmap, equation) in
          union {insert-term(sortmap, eq, vid, term) | eq ∈ equations}),
  mk-Conditional-equation1(eqs, eq)
     → (let mk-Unquantified-equation1(term1, term2) = eq,
            eqs′ = union {insert-term(sortmap, e, vid, term) | e ∈ eqs} in
          let eq′ = mk-Unquantified-equation1(insert-term-in-term(term1, vid, term),
                                              insert-term-in-term(term2, vid, term)) in
          {mk-Conditional-equation1(eqs′, eq′)}),
  ⊤ → {equation})
```

type: *Sortmap* $\mathit{Equation}_1$ $\mathit{Value\text{-}identifier}_1$ $\mathit{Ground\text{-}term}_1 \to \mathit{Equations}_1$

Objective Replace a value name by a *Ground-term*$_1$ in an equation enclosed by a quantified equation.

Parameters

sortmap A *Sortmap* which is used if the equation (in turn) contains quantified equations

equation The equation to be modified

vid The value identifier which should be replaced

term The *Term*$_1$ by which *vid* should be replaced.

Result A set of equations containing the modified equation. If the equation is quantified equation, the set might contain more that one equation.

Algorithm

Line 2-4 If it is an unquantified equation then replace *vid* by *term* in the two contained terms (*term*1, *term*2).

Line 5-7 If it is a quantified equation then first expand it into a set of unquantified equations and then replace the value identifier in every equation in the set.

Line 8-13 If it is a conditional equation then replace the value identifier by the term in every equation in the restriction and in the restricted equation and construct and return a set containing the modified conditional equation.

Line 14 If it is informal text then do not touch it.

insert-term-in-term(*term*, *vid*, *vterm*) ≙ (6.2.3.7)

```
if is-Ground-term1(term) ∨ is-Error-term1(term) then
  term
 else
  (let mk-Composite-term1(term′) = term in
   (is-Identifier1(term′)
       → if term′ = vid then vterm else term,
    is-Conditional-term1(term′)
       → (let mk-Conditional-term1(cond, t1, t2) = term′ in
            let cond′ = insert-term-in-term(cond, vid, vterm),
                t1′ = insert-term-in-term(t1, vid, vterm),
                t2′ = insert-term-in-term(t2, vid, vterm) in
            let term″ = mk-Conditional-term1(cond′, t1′, t2′) in
            if is-Ground-term1(cond′) ∧ is-Ground-term1(t1′) ∧ is-Ground-term1(t2′) then
              mk-Ground-term1(term″)
             else
              mk-Composite-term1(term″)),
    ⊤ → (let (opid, arglist) = term′ in
            let arglist′) = ⟨insert-term-in-term(arglist[i], vid, vterm) | 1 ≤ i ≤ len arglist⟩ in
            if (∃ arg ∈  elems arglist)(is-Composite-term1(arg)) then
              mk-Composite-term1((opid, arglist′))
             else
              mk-Ground-term1((opid, arglist′)))))
```

type: *Term*$_1$ *Value-identifier*$_1$ *Ground-term*$_1$ $\rightarrow$ *Term*$_1$

Objective Replace a value identifier (*vid*) by a (ground) term (*vterm*) in a term (*term*).

Parameters

term The *Term*$_1$ which should have its value identifier replaced.

vid The value identifier to be replaced

vterm The $Term_1$ which should be inserted instead of the value identifier.

Result The modified term.

Algorithm

Line 1 If it is a ground term or an error term then do not modify it.

Line 5-6 If it is an identifier and it is equal to *vid* then return the new term else do not modify it.

Line 7-12 If it is a conditional term then construct the conditional term wherein occurrences of *vid* in the three contained terms has been replaced by *vterm*.

Line 13-16 If all the three contained terms have become ground terms then return the new conditional term as a ground term else return it as a composite term.

Line 17-22 Else *term* must be an operator term in which case *vid* in the argument terms is replaced by *vterm* and if all the modified argument terms have become ground terms then return the new operator term as a ground term else return it as a composite term.

expand-conditional-term-in-equations(*equations*, *trueterm*, *falseterm*) ≜ (6.2.3.8)

```
1   if equations = {} then
2     {}
3    else
4     (let eq ∈ equations in
5      let (condset, eq′) =
6          cases eq:
7           (mk-Unquantified-equation1(,)
8              → ({}, eq),
9            mk-Conditional-equation1(condeq, eq)
10             → (condeq, eq)) in
11     let mk-Unquantified-equation1(t1, t2) = eq′ in
12     let (t1′, t1″, cond1) = expand-conditional-in-terms(t1),
13         (t2′, t2″, cond2) = expand-conditional-in-terms(t2) in
14     if cond1 = nil ∧ cond2 = nil then
15       {eq} ∪ expand-conditional-term-in-equations(equations \ {eq}, trueterm, falseterm)
16      else
17       (let (cond, term, nterm1, nterm2) be s.t. (cond, term, nterm1, nterm2) ∈
18            {(cond2, t1, t2′, t2″), (cond1, t2, t1′, t1″)} ∧ cond ≠ nil in
19        let eq1 = mk-Unquantified-equation1(cond, trueterm),
20            eq2 = mk-Unquantified-equation1(cond, falseterm) in
21        let condeq1 =
22            mk-Conditional-equation1(condset ∪ {eq1}, mk-Unquantified-equation1(term, nterm1)),
23            condeq2 =
24            mk-Conditional-equation1(condset ∪ {eq2}, mk-Unquantified-equation1(term, nterm2)) in
25        let equations′ = equations ∪ {condeq1, condeq2} \ {eq} in
26        expand-conditional-term-in-equations(equations′, trueterm, falseterm)))
```

type: $Equations_1\ Ground\text{-}term_1\ Ground\text{-}term_1 \rightarrow Equations_1$

Objective Replace every $Conditional\text{-}term_1$ by two $Conditional\text{-}equation_1$s.

Example: The equation

```
if a then b else c fi == d;
```

is expanded into

```
a == True  ==> b == d;
a == False ==> c == d;
```

Parameters

equations	The set of equations to be replaced
trueterm,*falseterm*	The two ground terms denoting the boolean True and False

Result The modified set of equations containing no *Conditional-term*$_1$s

Algorithm

Line 1	When the set of equations is empty, return nothing
Line 4-9	Take a equation from the set and extract the set of restriction (*condset*) and the restricted equation (*eq'*). If it is an unquantified equation, the restriction set is empty.
Line 12-13	Modify the terms in the restricted equation. *cond*1 and *cond*2 are the conditions to be tested upon. A condition is **nil** if the term do not contain any conditional terms. *t*1′, *t*2′ are the original terms (*t*1, *t*2) wherein a conditional term has been replaced by the **then** part of the conditional term and *t*1″ *t*2″ are the original terms wherein a conditional term has been replace by the **else** part of the conditional term.
Line 14-15	If none of the two terms contained any conditional terms then do not change the equation and continue with another equation in *equations*
Line 17	Choose one of the two terms to deal with. The other one will not be changed in this call.
Line 19-20	Construct the two unquantified equations, which must hold for the two modified equations.
Line 21-23	Construct two conditional equations wherein *eq*1 respective *eq*2 has been added as an extra condition. (*condeq*1) contains an equation wherein one of the original terms (*t*1 or *t*2) has been replaced by a term containing the **then** part and (*condeq*2) contains an equation wherein one of the original terms has been replaced by a term containing the **else** part.
Line 26	Include the two new conditional equations in the set of remaining equations to be considered (because one of the terms in *eq* has not been expanded and because the expanded term may contain further conditional terms).

expand-conditional-term-in-conditions(*equations*, *trueterm*, *falseterm*) ≜ (6.2.3.9)

```
 1   if equations = {} then
 2     {}
 3   else
 4     (let eq ∈ equations in
 5      let mk-Conditional-equation1(condset, eq′) = eq in
 6      if (∃cond ∈ condset)
 7          ((let mk-Unquantified-equation1(t1, t2) = cond in
 8            let (, , cond1) =
 9                 expand-conditional-in-terms(t1),
10                 (, , cond2) =
11                 expand-conditional-in-terms(t2) in
12            cond1 ≠ nil ∨ cond2 ≠ nil)) then
13        (let (condeq, cond, term, nterm1, nterm2) be s.t. condeq ∈ condset ∧
14              (let mk-Unquantified-equation1(t1, t2) =
15                   condeq in
16               let (t1′, t1″, cond1) =
17                   expand-conditional-in-terms(t1),
18                   (t2′, t2″, cond2) =
19                   expand-conditional-in-terms(t2) in
20               (cond, term, nterm1, nterm2) = (if cond1 = nil
21                                                 then (cond2, t1, t2′ t2″)
22                                                 else (cond1, t2, t1′ t1″))) in
23          let eq1 = mk-Unquantified-equation1(cond, trueterm),
24              eq2 = mk-Unquantified-equation1(cond, falseterm) in
25          let condset′ = condset \ {condeq} ∪ {eq1, mk-Unquantified-equation1(term, nterm1)},
26              condset″ = condset \ {condeq} ∪ {eq2, mk-Unquantified-equation1(term, nterm2)} in
27          let equations′ = equations \ {eq} ∪ {mk-Conditional-equation1(condset′, eq′),
28                                               mk-Conditional-equation1(condset″, eq′)} in
29          expand-conditional-term-in-conditions(equations′, trueterm, falseterm))
30        else
31        {eq} ∪ expand-conditional-term-in-conditions(equations \ {eq}, trueterm, falseterm))
```

type: *Conditional-equation*$_1$-**set** *Ground-term*$_1$ *Ground-term*$_1$ → *Equations*$_1$

Objective Split the conditional equations in *equations* into two conditional equations if they contain any conditional terms in the *Restriction*$_1$.

Example: The equation

```
if b then c else d fi == e ==> f == g;
```

is expanded into

```
b == True,  c == e == > f == g;
b == False, d == e == > f == g;
```

Parameters

equations The set of conditional equations

trueterm, *falseterm* The two ground terms denoting boolean True and False.

Result The expanded set of equations.

Algorithm

Line 1 When through, return the empty set

Line 4-12 Take a conditional equation from the set and if it does not contain a conditional term in the restriction part then continue with the rest of equations in the set (line 31)

Line 13-21 Extract the unquantified equation from the set of restrictions which contains the conditional term (*condeq*), the condition in the conditional term (*cond*), the **then** version of the term in the unquantified equation containing the conditional term (*nterm*1), the **else** version of the term in the unquantified equation containing the conditional term (*nterm*2) and the other term of the unquantified equation (*term*).

Line 23-24 Construct the two additional restrictions to be included in the respective restriction sets.

Line 25-26 Construct the two modified restriction sets.

Line 27 Replace the old conditional equation by the two new conditional equations in the equation set.

Line 29 Repeat the operation with the modified equation set.

expand-conditional-in-terms(*t*) $\triangleq$ (6.2.3.10)

```
1   if is-Error-term1(t ) then
2     (t, t, nil)
3    else
4     (let mk-Ground-term1(term) = t in
5      cases term:
6       (mk-Identifier1(,)
7           → (t, t, nil),
8        mk-Conditional-term1(cond, t1, t2)
9           → (t1, t2, cond),
10      (id, arglist)
11          → if (∃ arg ∈ elems arglist)
12                ((let (, , cond) =
13                      expand-conditional-in-terms(arg) in
14                  cond ≠ nil)) then
15              (let (i, t1, t2, cond) be s.t. i ∈ ind arglist ∧
16                   cond ≠ nil ∧
17                   expand-conditional-in-terms(arglist [i]) = (t1, t2, cond) in
18               let arglist′ =
19                   ⟨arglist [n] | 1 ≤ n < i⟩ ⁀ ⟨t1⟩ ⁀ ⟨arglist[n] | i < n ≤ len arglist⟩,
20                   arglist″ =
21                   ⟨arglist[n] | 1 ≤ n < i⟩ ⁀ ⟨t2⟩ ⁀ ⟨arglist[n] | i < n ≤ len arglist⟩ in
22               (mk-Ground-term1((id, arglist′)), mk-Ground-term1((id, arglist″)), cond))
23             else
24              (t, t nil)))
```

type: $Term_1 \rightarrow Term_1\ Term_1\ [Ground\text{-}term_1]$

Objective Split a term (*t*) into three terms. If *t* does not contain a conditional term then the two first terms are not relevant and the third one is **nil**. Otherwise the result is *t* modified to contain the **then** part, *t* modified to contain the **else** part and the boolean condition term.

Result The three new terms.

Algorithm

Line 1-6 If it is an error term then do not modify it and indicate that it does not contain a conditional term by returning **nil** as the condition term.

Line 8 If it is a conditional term then return its three parts.

Line 10-14 If it is an operator term and one of its arguments contain a conditional term then

Line 15-17	Take an argument term which contains a conditional term and split it. *i* is the position in the argument list.
Line 18-20	Construct the argument lists for the **then** part (*arglist’*) and for the **else** part (*arglist’’*) and
Line 22	Return the two operator terms corresponding to the **then** part, to the **else** part and the boolean condition in the conditional term in the argument.

eval-unquantified-equations(*sortmap*, *equations*) ≜ (6.2.3.11)

```
1    (if equations = {} then
2       sortmap
3     else
4       (let eq ∈ equations in
5        let mk-Unquantified-equation1(lterm, rterm) = eq in
6        let sort ∈ dom sortmap be s.t. (∃termset ∈ sortmap(sort))(lterm ∈ termset) in
7        let termset1 be s.t. termset1 ∈ sortmap(sort) ∧ lterm ∈ termset1 in
8        let termset2 be s.t. termset2 ∈ sortmap(sort) ∧ rterm ∈ termset2 in
9        if termset1 = termset2 then
10          eval-unquantified-equations(sortmap, equations \ {eq})
11        else
12          (let newset = sortmap(sort) \ {termset1, termset2} ∪ {termset1 ∪ termset2} in
13           let sortmap′ = sortmap + [sort ↦ newset] in
14           let sortmap″ = eval-deduced-equivalence(sortmap′) in
15           eval-unquantified-equations(sortmap″, equations \ {eq}))))
```

type: *Sortmap Equations*$_1$ → *Sortmap*

Objective Modify *sortmap* (the equivalence classes) in accordance with *equations*.

Parameters

Sortmap	A *Sortmap* to be modified.
equations	A set of unquantified equations.

Algorithm

Line 1	When through, return the modified *Sortmap*
Line 4-5	Extract the two *Term*$_1$s from one of the (remaining) equations.
Line 6	Extract the sort of *lterm* (which is the same as the sort of *rterm*).
Line 7	Extract the equivalence class which contains *lterm*.
Line 8	Extract the equivalence class which contains *rterm*.
Line 9	If the terms denote the same equivalence class then do not update *sortmap* else
Line 12	Define a new set of equivalence classes wherein the two equivalence classes has been unified.
Line 13	Modify *sortmap* to contain the new set of equivalence classes
Line 14	Reduce the number of equivalence classes by using the information obtained by the equation
Line 15	Repeat the operation for the rest of the equations.

eval-deduced-equivalence(*sortmap*) ≜ (6.2.3.12)

```
1   if (∃class1, class2, class3 ∈ union rng sortmap)
2       (class1 ≠ class2 ∧
3        (∃term1, term2 ∈ class3)((∃term ∈ class1)(replace-term(term, term1, term2) ∈ class2))) then
4     (let (class1, class2, class3) be s.t. {class1, class2, class3} ⊂ union rng sortmap ∧
5           class1 ≠ class2 ∧
6           (∃term1, term2 ∈ class3)((∃term ∈ class1)(replace-term(term, term1, term2) ∈ class2)) in
7      let sort be s.t. {class1, class2} ⊂ rng sortmap(sort) in
8      let classes = sortmap(sort) in
9      let classes′ = classes \ {class1, class2} ∪ {class1 ∪ class2} in
10     let sortmap′ = sortmap + [sort ↦ classes′] in
11     eval-deduced-equivalence(sortmap′))
12    else
13     sortmap
```

type: *Sortmap* → *Sortmap*

Objective Reduce the number of the equivalence classes for sorts by using the information that two terms of a sort are in the same equivalence class.

Parameters

sortmap A *Sortmap* containing the equivalence classes which are to be modified

Result The *Sortmap* where the number of equivalence classes for some of the sorts has been reduced

Algorithm

Line 1 If there exists three equivalence classes *class*1, *class*2, *class*3 in the *Sortmap* such that *class*1 and *class*2 are disjoint (*class*3 may be equal to *class*1 or *class*2 or it may denote another equivalence class, even of another sort) and there exists two terms (*term*1 and *term*2) in *class*3 such that when replacing *term*1 by *term*2 in a term (*term*) taken from *class*1, a term in *class*2 is obtained then

Line 4-13 *class*1 and *class*2 are merged into one equivalence class

Line 4-6 Let *classl*, *class*2, *class*3 denote three such equivalence classes

Line 7 Let *sort* denote the sort of *class*1 and *class*2. *class*1 and *class*2 cannot be of different sort as line 1-3 in that case would not be satisfied

Line 8-10 Form a new *Sortmap* where the two equivalence classes for the sort have been merged

Line 11 Repeat the operation (with the modified *Sortmap*) until no more equivalence classes can be merged *replace-term*(*term*, *oldterm*, *newterm*) ≜ (6.2.3.13)

```
1   if term = oldterm) then
2     newterm
3    else
4    (let mk-Ground-term1(contents) = term in
5     (is-Identifier1(contents)
6        → term,
7     ⊤ → (let (opid, arglist) = term in
8            if (∃i ∈ ind arglist)(replace-term(arglist[i], oldterm, newterm) ≠ arglist[i]) then
9              (let i ∈ ind arglist be s.t. replace-term(arglist[i], oldterm, newterm) ≠ arglist[i] in
10              let arglist′ = ⟨arglist[n] | 1 ≤ n < i⟩ ⁀
11                  ⟨replace-term(arglist[i], oldterm, newterm)⟩ ⁀
12                  ⟨arglist[n] | i < n ≤ len arglist⟩ in
13              mk-Ground-term1((opid, arglist′)))
14            else
15             term)))
```

type: *Ground-term*$_1$ *Ground-term*$_1$ *Ground-term*$_1$ → *Ground*-term$_1$

Objective Replace an occurrence of *oldterm* in *term* by *newterm* and return the modified term

Algorithm

Line 1 If the entire term is equal to *oldterm* then return the new term

Line 5 If the term is an identifier (and it is different from *oldterm)* then no replacement is made else

Line 7 The term is an operator term (conditional terms cannot occur since *term* is taken from an equivalence class). Let *op* denote the operator identifier and let *arglist* denote the argument list

Line 8 If there exists an argument which contains *oldterm* then

Line 9 Let *i* denote the index to the argument which contains *oldterm*

Line 10-12 Construct the argument list where an occurrence of *oldterm* in element *i* has been replaced by *newterm*

Line 13 Return the modified term

Line 15 If *oldterm* do not occur in the argument list then the term is not changed

eval-conditional-equations(*sortmap*, *condequations*) ≜ (6.2.3.14)

```
if (∃condeq ∈ condequations)(restriction-holds(condeq, sortmap)) then
  (let condeq ∈ condequations be s.t. restriction-holds(condeq, sortmap) in
   let mk-Conditional-equation1(, eq) = condeq in
   let sortmap′ = eval-unquantified-equations(sortmap, {eq}) in
   eval-conditional-equations(sortmap′, condequations \ {condeq}))
 else
   sortmap
```

type: *Sortmap* *Conditional-equation*$_1$-**set** → *Sortmap*

Objective Reduce the number of equivalence classes in a *Sortmap* in accordance with the conditional equations for a scopeunit.

Parameters

sortmap A *Sortmap*

condequations A set of conditional equations

Result The modified *Sortmap*

Algorithm

Line 1 If there exists a conditional equation which holds then

Line 2 Let *condeq* denote the conditional equation which holds

Line 3-4 Update *Sortmap* with the properties reflected by the restricted equation (*eq*)

Line 5 Repeat the operation until there are no more conditional equations in the remaining set which hold.

$\textit{restriction-holds}(\textbf{mk-}\textit{Conditional-equation}_1(\textit{eqs},), \textit{sortmap}) \triangleq$ (6.2.3.15)

1 $(\textbf{let } \textit{termpairs} = \{\{\textit{term}1, \textit{term}2\} \mid \textbf{mk-}\textit{Unquantified-equation}_1(\textit{term}1, \textit{term}2) \in \textit{eqs}\} \textbf{ in}$

2 $(\forall \textit{pair} \in \textit{termpairs})((\exists \textit{class} \in \textbf{union rng } \textit{sortmap})(\textit{pair} \subseteq \textit{class})))$

type: $\textit{Conditional-equation}_1\ \textit{Sortmap} \rightarrow \textit{Bool}$

Objective Test whether the set of restrictions for a conditional equation holds

Parameters

eqs The set of restrictions

sortmap The *Sortmap* used for checking whether the restrictions hold

Result True if success

Algorithm

Line 1 Construct a set of pairs of terms each containing the left-hand side term and the right-hand side term of a restriction in the set of restrictions

Line 2 The restrictions hold if it for each restriction holds that the right-hand side term is in the same equivalence class as the left-hand side term.

6.2.4 Term Reduction Map Generation

make-term-reduce-map(*sortmap*, *level*)(*dict*) ≜ (6.2.4.1)

```
1   (let surtrmap =
2       if len level = 1 then
3         (let recogterms = {dict(TRUEVALUE), dict(FALSEVALUE), dict(NULLVALUE)} in
4          [{t} ↦ t | t ∈ recogterms])
5         else
6         (let surlevel = ⟨level[i] | 1 ≤ i < len level⟩ in
7          s-Term-reduce-map(dict((surlevel, TYPE)))) in
8   let classes = union rng sortmap in
9   [class → (mk-Error-term_1() ∈ class
10                 → mk-Error-term_1(),
11             (∃class′ ∈ dom surtrmap)(class′ ⊆ class)
12                 → (let class′ ∈ dom surtrmap) be s.t. class′ ⊆ class in
13                     surtrmap(class′)),
14             ⊤ → (let term ∈ class in
15                     term)) |
16  class ∈ classes])
```

type: *Sortmap Qualifier*$_1$ → *Entity-dict* → *Term-reduce-map*

Objective Construct the *Term-reduce-map* which applies at a given scope unit level.

Parameters

sortmap The sortmap which applies at the given scope unit level.

sortmap The qualifier for the scope unit level.

Result A *Term-reduce-map* mapping all equivalence classes visible at the given scope unit level to their chosen canonical ground term.

Algorithm

Line 1-7 Obtain the *Term-reduce-map* which applies at the enclosing scope unit level. If the current scope unit is the system level the "enclosing" *Term-reduce-map* is a dummy one (line 3-4) ensuring that the three SDL values which must be recognizable by the interpretation functions (SDL Pid value Null and Boolean values True and False) are always represented by the ground terms found in the *Entity-dict* entries TRUEVALUE, FALSEVALUE and NULLVALUE.

Line 8 Get the set of all equivalence classes visible at the current scope unit level.

Line 9-16 Each canonical ground term is selected according to the following criteria:

Line 9-10 If the equivalence class contains the error term the error term is chosen as canonical term.

Line 11-13 If the value represented by the equivalence class is also visible at the enclosing scope unit level (i.e. there exists an "enclosing" equivalence class such that this class is a subset of the treated equivalence class, line 11), then the canonical term chosen in the enclosing scope unit is also chosen in the current scope unit.

Line 14-15 If the value represented by the equivalence class belongs to a sort local to the current scope unit an arbitrary ground term is chosen as canonical ground term.

6.2.5 Wellformedness Checks

is-wf-literals(*level*)(*dict*) ≜ (6.2.5.1)

```
1   (let sortmap = s-Sortmap(dict((level, TYPE))) in
2    let classes = union rng sortmap in
3    ¬(∃class ∈ classes)
4      ((∃{mk-Ground-term_1(t), mk-Error-term_1 ()} ⊆ class)
5        (is-Identifier_1(t))))
```

type: $Qualifier_1 \rightarrow Entity\text{-}dict \rightarrow Bool$

Objective Check that no literal is equal to the error term.

Parameters

level The qualifier denoting the current scope unit level.

Result **true** if the check succeed, else **false**.

Algorithm

Line 1 Obtain the sort map for the scope unit.

Line 2 Get all equivalence classes visible in the scope unit.

Line 3-5 There must not exist an equivalence class which both contains a literal ground term and the error term.

is-wf-values(*level*)(*dict*) ≜ (6.2.5.2)

```
1   if len level = 1 then
2     (let sortmap = s-Sortmap(dict((level, TYPE))) in
3      is-wf-boolean(sortmap, dict(TRUEVALUE), dict(FALSEVALUE)) ∧
4      is-wf-pid(sortmap(dict(PIDSORT))))
5    else
6     (let surlevel = ⟨level[i] | 1 ≤ i < len level⟩ in
7      let sursortmap = s-Sortmap(dict((surlevel, TYPE))),
8          sortmap = s-Sortmap(dict((level, TYPE))) in
9      (∀sortid ∈ dom sursortmap)
10      ((let survset = sursortmap(sortid),
11            vset = sortmap(sortid) in
12        (∀class ∈ vset)((∃!class′ ∈ survset)(class′ ⊆ class)))))
```

type: $Qualifier_1 \rightarrow Entity\text{-}dict \rightarrow Bool$

Objective Check that no unification or generation of equivalence classes is done for sorts which are visible in the enclosing scope unit.

Parameters

level The qualifier for the current scope unit level.

Result **true** if the check succeeds, else **false**.

Algorithm

Line 1 Distinguish between the system level and other levels.

Line 2 Obtain the sort map of the system level.

Line 3-4 Check the wellformedness conditions on the SDL Boolean and Pid sorts.

Line 6 Obtain the qualifier of the enclosing scope unit level.

Line 7-8	Obtain the sort maps for the enclosing and the current scope unit levels.
Line 9	For all sorts visible in the enclosing scope unit the wellformedness condition in line 10-12 must hold.
Line 10-11	For the sort considered, obtain the equivalence class sets for the enclosing and the current scope unit levels.
Line 12	For each equivalence class in the current scope unit it must hold that it includes all the terms of exactly one equivalence class in the enclosing scope unit.

$\textit{is-wf-boolean}(\textit{sortmap-trueterm}, \textit{falseterm}) \triangleq$ (6.2.5.3)

1 $(\mathbf{let}\ \textit{boolsort} \in \mathbf{dom}\ \textit{sortmap}\ \mathbf{be\ s.t.}\ (\exists \textit{class} \in \textit{sortmap}(\textit{boolsort}))(\textit{trueterm} \in \textit{class})\ \mathbf{in}$

2 $(\forall \textit{class} \in \textit{sortmap}(\textit{boolsort}))$

3 $(\mathbf{mk}\text{-}\textit{Error-term}_1() \notin \textit{class} \supset \mathbf{card}\ (\{\textit{trueterm}, \textit{falseterm}\} \cap \textit{class}) = 1))$

type: $\textit{Sortmap}\ \textit{Ground-term}_1\ \textit{Ground-term}_1 \to \textit{Bool}$

Objective Check the wellformedness of the Boolean sort.

Parameters

sortmap	The (system level) sort map.
trueterm	The canonical ground term for True.
falseterm	The canonical ground term for False.

Algorithm

Line 1	Obtain the AS_1 identifier of the boolean sort.
Line 2-3	Each equivalence classe of the Boolean sort which does not contain the error term must contain exactly one of the Boolean literals True and False.

$\textit{is-wf-pid}(\textit{pidvset}) \triangleq$ (6.2.5.4)

1 $(\mathbf{let}\ \textit{pidvset}' = \{\textit{class} \in \textit{pidvset} \mid \mathbf{mk}\text{-}\textit{Error-term}_1() \notin \textit{class}\}\ \mathbf{in}$

2 $(\forall n \in N_1)((\exists s \subset \textit{pidvset}')(\mathbf{card}\ s > n)))$

type: $\textit{Term-class-}\mathbf{set} \to \textit{Bool}$

Objective Check the wellformedness of the Pid sort.

Parameters

pidvset	The set of equivalence classes for the Pid sort.

Algorithm

Line 1	Obtain the set of Pid equivalence classes *not* containing the error term.
Line 2	The number of equivalence classes (not containing the error term) for the Pid sort must be infinite, i.e. for each natural number *n* there must exist a (finite) subset *s* of the equivalence class set such that the number of elements in *s* is greater than *n*.

6.3 Selection of Consistent Subset

This section defines the functions for checking and selecting a consistent subset according to a given consistent subset selection (the entry function is s*elect-consistent-subset).* This consists of two steps: First, for each (selected) block/subblock in the whole system *either* the contained block substructure *or* the contained process definitions, signal routes and channel to route connections are removed. Second, subsignals used in subchannels in some substructure are propagated to channels connected to this substructure, i.e. if a channel carries a parent signal of some subsignal carried by a connected subchannel, the parent signal is replaced by the subsignals on the channel. Note that this transformation may transform a unidirectional channel to a bidirectional one.

Example: Let an SDL system contain the signal and channel definitions

```
signal s
 refinement
  signal s1, s2;
  reverse signal s3;
 endrefinement;

signal t;

channel c from b1 to b2 with s, t; endchannel;
```

and let the origin block b1 contain a (selected) substructure which contains the subchannel definitions and connection

```
channel c1 from subb1 to env with s1, s2; endchannel;

channel c2 from env to subb2 with s3; endchannel;

channel c3 from subb3 to env with t; endchannel;

connect c and c1, c2, c3;
```

After subsignal propagation the channel c will be defined as

```
channel c from b1 to b2 with s1, s2, t;
          from b2 to b1 with s3;
endchannel;
```

select-consistent-subset(*sysdef*, *subset*)(*dict*) $\triangleq$ (6.3.1)

(**let** *sysdef′* = *select-consistent-subset-sys*(*sysdef*, *subset*) **in**
 let *sysdef″* = *propagate-refinement-sys*(*sysdef′*)(*dict*) **in**
 sysdef″)

type: $\textit{System-definition}_1\ \textit{Block-identifier}_1\textbf{-set} \rightarrow \textit{Entity-dict} \rightarrow \textit{System-definition}_1$

Objective Transform a system definition according to a consistent subset selection.

Parameters

sysdef The system definition to be transformed.

subset The (assumed) consistent subset represented by a set of block identifiers and block substructure identifiers.

Result The transformed system definition.

Algorithm

Line 1 Remove the parts which will not be used (either block substructures or processes, signal routes and channel to route connections).

Line 2 Propagate the use of subsignals on subchannels to channels to which the subchannels are connected.

Line 3 Return the transformed system definition.

6.3.1 Removal of Non-Selected Substructures and Processes

select-consistent-subset-sys(*sysdef, subset*) ≜ (6.3.1.1)

```
1   (let mk-System-definition_1 (snm, bset, cset, sigset, dt, sset) = sysdef in
2    let level = ⟨mk-System-qualifier_1(snm)⟩ in
3    let bset′ = {select-consistent-subset-block(block, subset, level) | block ∈ bset} in
4    mk-System-definition_1(snm, bset′, cset, sigset, dt, sset))
```

type: *System-definition*$_1$ *Block-identifier*$_1$-**set** → *System-definition*$_1$

Objective Select consistent subset in a system definition.

Parameters

sysdef The system definition.

subset The (assumed) consistent subset.

Result The transformed system definition.

Algorithm

Line 1 Decompose the system definition.

Line 2 Construct the qualifier denoting the system level.

Line 3 Transform the system-level blocks.

Line 4 The transformed blocks replace the original ones in the system.

select-consistent-subset-block(*block, subset, level*) ≜ (6.3.1.2)

```
1   (let mk-Block-definition_1 (bnm, pset, sigset, connects, srset, dt, sset, osub) = block in
2    if mk-Identifier_1(level, bnm) ∈ subset then
3      (let level′ = level ⁀ ⟨mk-Block-qualifier_1(bnm)⟩ in
4       let osub′ = select-consistent-subset-osub(osub, subset, level′) in
5       (osub′ ≠ nil
6            → mk-Block-definition_1⟨bnm, {}, sigset, {}, {}, dt, sset, osub′),
7         pset ≠ {}
8            → mk-Block-definition_1(bnm, pset, sigset, connects, srset, dt, sset, nil),
9         ⊤ → exit("§3.2.1: Leaf block contains no processes")))
10    else
11     exit("§3.2.1: Block or subblock is not in consistent subset"))
```

type: *Block-definition*$_1$ *Block-identifier*$_1$-**set** *Qualifier*$_1$ → *Block-definition*$_1$

Objective Select consistent subset in a block definition.

Parameters

block The block definition.

subset The (assumed) consistent subset.

level The qualifier for the system or block substructure containing the block.

Result The transformed block.

Algorithm

Line 1 Decompose the block definition.

Line 2,11 The block or subblock must be in the consistent subset.

Line 3 Construct the qualifier for the block level.

Line 4 Transform the substructure of the block if present and selected.

Line 5-6 If the block substructure is present and selected, it replaces the original substructure. As the processes, signal routes and channel to route connections in the block will not be interpreted, they are removed.

Line 7-9 Otherwise, the block is a leaf block and must contain at least one process definition.

select-consistent-subset-osub(*osub*, *subset*, *level*) ≜ (6.3.1.3)

```
1  if osub = nil then
2    nil
3  else
4    select-consistent-subset-sub(osub, subset, level)
```

type: $[Block\text{-}substructure\text{-}definition_1]$ $Block\text{-}identifier_1$-**set** $Qualifier_1$
$\rightarrow [Block\text{-}substructure\text{-}definition_1]$

Objective Select consistent subset in a block substructure if present and selected.

Parameters

osub The optional block substructure.

subset The (assumed) consistent subset.

level The qualifier denoting the enclosing block.

Result If the block substructure is present and selected, then the transformed block substructure, otherwise **nil**.

Algorithm

Line 1-2 If the block substructure is absent then indicate this.

Line 4 Otherwise, transform the block substructure if selected.

select-consistent-subset-sub(*sub*, *subset*, *level*) ≜ (6.3.1.4)

```
1  (let mk Block-substructure-definition₁ (bsnm, bset, connects, cset, sigset, dt, sset) = sub in
2   if mk-Identifier₁(level, bsnm) ∈ subset then
3     (let level′ = level ⁀ ⟨mk-Block-substructure-qualifier₁(bsnm)⟩ in
4      let bset′ = {select-consistent-subset-block(block, subset, level′) | block ∈ bset} in
5      mk-Block-subtructure-definition₁(bsnm, bset′, connects, cset, sigset, dt, sset))
6    else
7     nil)
```

type: $Block\text{-}substructure\text{-}definition_1$ $Block\text{-}identifier_1$-**set** $Qualifier_1$
$\rightarrow [Block\text{-}substructure\text{-}definition_1]$

Objective Select consistent subset in a block substructure if selected.

Parameters

sub The block substructure.

subset The (assumed) consistent subset.

level The qualifier denoting the enclosing block.

Result If the block substructure is selected, then the transformed block substructure, otherwise **nil**.

Algorithm

Line 1 Decompose the block substructure.

Line 2 If the block substructure is selected, then

Line 3 construct the qualifier denoting the block substructure level,

Line 4 transform the contained subblock definitions,

Line 5 and replace the original subblocks with the transformed ones.

Line 7 If the block substructure is *not* selected, then return **nil** to indicate this.

6.3.2 Subsignal Propagation

propagate-refinement-sys(*sysdef*)(*dict*) $\triangleq$ (6.3.2.1)

1 (**let mk**-*System-definition*$_1$(*snm*, *bset*, *cset*, *sigset*, *dt*, *sset*) = *sysdef* **in**
2 **let** *level* = ⟨**mk**-*System-qualifier*$_1$(*snm*)⟩ **in**
3 **let** *bset′* = {*propagate-refinement-block*(*block*, *level*)(*dict*) | *block* ∈ *bset*} **in**
4 **let** *cset′* = { *propagate-refinement-chan*(*chan*, *bset′*, *level*)(*dict*) | *chan* ∈ *cset*} **in**
5 **mk**-*System-definition*$_1$(*snm*, *bset′*, *cset′*, *sigset*, *dt*, *sset*))

type: *System-definition*$_1$ → *Entity-dict* → *System-definition*$_1$

Objective Propagate subsignals in a system where the consistent subset has already been selected.

Parameters

sysdef The system definition.

Result The system where subsignals have been propagated.

Algorithm

Line 1 Decompose the system definition.

Line 2 Construct the system level qualifier.

Line 3 Propagate subsignals in each block defined at system level.

Line 4 Propagate subsignals on each channel defined at system level.

Line 5 The transformed blocks and channels replace the original ones in the system.

propagate-refinement-block(*block*, *level*)(*dict*) $\triangleq$ (6.3.2.2)

1 (**let mk**-*Block-definition*$_1$(*bnm*, *pset*, *sigset*, *connects*, *srset*, *dt*, *sset*, *osub*) = *block* **in**
2 **let** *level′* = *level* ⌢ ⟨**mk**-*Block-qualifier*$_1$(*bnm*)⟩ **in**
3 **let** *osub′* = **if** *osub* ≠ **nil then** *propagate-refinement-sub*(*osub*, *level′*)(*dict*) **else nil in**
4 **mk**-*Block-definition*$_1$(*bnm*, *pset*, *sigset*, *connects*, *srset*, *dt*, *sset*, *osub′*))

type: *Block-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Block-definition*$_1$

Objective Propagate subsignals in a block.

Parameters

block The block definition.

level The qualifier of the enclosing system or substructure.

Result The transformed block.

Algorithm

Line 1 Decompose the block definition.

Line 2 Construct the block level qualifier.

Line 3 Propagate subsignals in the block substructure if it is present.

Line 4 The transformed block substructure replaces the original one.

propagate-refinement-sub(*sub*, *level*)(*dict*) ≜ (6.3.2.3)

```
1  (let mk-Block-substructure-definition1(bsnm, bset, connects, cset, sigset, dt, sset) = sub in
2   let level′ = level ⁀ ⟨mk-Block-substructure-qualifier1(bsnm)⟩ in
3   let bset′ = {propagate-refinement-block(block, level′)(dict) | block ∈ bset} in
4   let cset′ = {propagate-refinement-chan(chan, bset′, level′)(dict) | chan ∈ cset} in
5   if (∀connect ∈ connects)(is-consistent-chancon(connect, cset′)) then
6      mk-Block-subtructure-definition1(bsnm, bset,′connects, cset′, sigset, dt, sset)
7    else
8      exit("§3.3: Illegal refinement of channel"))
```

type: *Block-substructure-definition*$_1$ *Qualifier*$_1$ → *Entity-dict*
→ *Block-substructure-definition*$_1$

Objective Propagate subsignals in a block substructure.

Parameters

sub The block substructure.

level The qualifier of the enclosing block.

Result The transformed block substructure.

Algorithm

Line 1 Decompose the block substructure.

Line 2 Construct the block substructure level qualifier.

Line 3 Propagate subsignals in each block.

Line 4 Propagate subsignals on each channel.

Line 5-8 For each channel connection at the boundary of the substructure, check that no two signals on different refinement levels can go through this connection.

Line 6 The transformed blocks and channels replace the original ones.

propagate-refinement-chan(*chan*, *bset*, *level*)(*dict*) ≜ (6.3.2.4)

```
(let mk-Channel-definition1(chnm, nodelay, forwpath, orevpath) = chan in
 let chid = mk-Identifier1(level, chnm) in
 let mk-Channel-path1(endp1, endp2, forwsigs) = forwpath in
 let revpath = if orevpath ≠ nil then orevpath else mk-Channel-path1(endp2, endp1, {}) in
 let mk-Channel-path1(, , revsigs) = revpath in
 let forwpath′ = propagate-refinement-cpath(chid, forwpath, revsigs, bset)(dict),
     revpath′ = propagate-refinement-cpath(chid, revpath, forwsigs, bset)(dict) in
 let orevpath′ =
     (let mk-Channel-path1(, , ss) = revpath′ in
      if ss = {} then nil else revpath′) in
 mk-Channel-definition1(chnm, nodelay, forwpath′, orevpath′))
```

type: *Channel-definition*$_1$ *Block-definition*$_1$**-set** *Qualifier*$_1$ → *Entity-dict*
→ *Channel-definition*$_1$

Objective Propagate subsignals to a channel.

Parameters

chan The channel definitions.

bset The set of blocks (where subsignals have already been propagated) defined in the same system or substructure as the channel.

level The qualifier of the enclosing system or substructure.

Result The transformed channel.

Algorithm

Line 1 Decompose the channel definition.

Line 2 Construct the identifier of the channel.

Line 3 Decompose the forward channel path into its endpoints and conveyed signal set.

Line 4 If the channel is unidirectional then construct a “dummy” reverse channel path conveying no signals.

Line 5 Obtain the (possibly empty) set of signals conveyed in the reverse direction.

Line 6-7 Propagate subsignals to each of the channel paths. Signals conveyed in a given direction may contribute with reverse subsignals in the opposite direction (which is the reason for the third parameter of *propagate-refinement-cpath*).

Line 8-7 If the set of signals conveyed on the transformed reverse channel path is empty the reverse channel path is removed.

Line 11 The transformed channel paths replace the original ones.

propagate-refinement-cpath(*chid*, *cpath*, *revsigs*, *bset*)(*dict*) ≜ (6.3.2.5)

```
1    (let mk-Channel-path1(endp1, endp2, forwsigs) = cpath in
2     let foutsigs = inout-going-signals(OUT, chid, endp1, bset),
3         finsigs = inout-going-signals(IN, chid, endp2, bset),
4         routsigs = inout-going-signals(OUT, chid, endp2, bset),
5         rinsigs = inout-going-signals(IN, chid, endp1, bset) in
6     if (∃sig1 ∈ foutsigs ∪ rinsigs, sig2 ∈ finsigs ∪ routsigs, sig ∈ forwsigs ∪ revsigs)
7         (is-sig-or-subsig(sig1, sig) ∧ is-sig-or-subsig(sig2, sig) ⊃
8          is-proper-or-subsig(sig1, sig2) ∨ is-proper-subsig(sig2, sig1)) then
9       exit(“§3.3: Illegal refinement of channel”)
10     else
11      (let forwsig′ =
12           extract-direction-subsignals(forwsigs, foutsigs ∪ finsigs, nil)(dict) ∪
13           extract-direction-subsignals(revsigs, foutsigs ∪ finsigs, REVERSE)(dict) in
14       mk-Channel-path1(endp1, endp2, forwsigs′)))
```

type: $\mathit{Channel\text{-}identifier}_1$ $\mathit{Channel\text{-}path}_1$ $\mathit{Signal\text{-}identifier}_1$**-set** $\mathit{Block\text{-}definition}_1$**-set** $\rightarrow$ *Entity-dict* $\rightarrow$ $\mathit{Channel\text{-}path}_1$

Objective Propagate subsignals to a channel path.

Parameters

chid The identifier of the channel.

cpath The channel path.

revsigs The signals conveyed in the opposite direction on the channel.

bset The set of blocks (where subsignals have already been propagated) defined in the same system or substructure as the channel.

Result The transformed channel path.

Algorithm

Line 1 Decompose the channel path.

Line 2-5 Obtain the set of (sub)signals going out through the origin end point (line 2), in through the destination end point (line 3), out through the destination end point (line 4), and in through the origin end point (line 5).

Line 6-9 If there exists a signal *sig*1 going through the origin connection point and a signal *sig*2 going through the destination connection point of the channel which are both (direct or indirect) (sub)signals of the same signal *sig* conveyed by the channel path, *sig*1 and *sig*2 are not allowed to be on different refinement levels of each other.

Line 11-13 Extract from the set of (sub)signals going out through the origin connection point or in through the destination end point the (sub)signals which can be conveyed by the channel path. Signals going in the opposite direction on the channel may also contribute to the (sub)signal set because they can have reverse subsignals (line 13).

is-consistent-chancon(*connect*, *cset*) ≙ (6.3.2.6)

```
1   (let mk-Channel-connection_1(, subchidset) = connect in
2    let cset′ = {select-channel(subchid, cset) | subchid ∈ subchidset} in
3    let connectsigs = union {direction-signals-chan(chan, FORWARD) ∪
4                             direction-signals-chan(chan, REVERSE) | chan ∈ cset′} in
5    ¬(∃sig1, sig2 ∈ connectsigs)(is-proper-subsig(sig1, sig2)))
```

type: *Channel-connection*$_1$ *Channel-definition*$_1$-**set** → *Bool*

Objective Check that no two signals on different refinement levels can go through a given connection point at the boundary of a block substructure, including the case where one signal goes out and the other goes in.

Parameters

connect The channel connection.

cset The set of (transformed) channel definitions in the same block substructure as the connect.

Result **true** if the condition holds, otherwise **false**.

Algorithm

Line 1 Get the set of identifiers of subchannels connected to the *connect.*

Line 2 Select the connected subchannels.

Line 3-4 Extract all signals (from both directions) conveyed on the connected subchannels.

Line 5 No two signals on the connected subchannels are allowed to be on different refinement levels.

inout-going-signals(*inout, chid, endp, bset*) ≙ (6.3.2.7)

```
if endp = ENVIRONMENT then
  {}
 else
  (let block = select-block(endp, bset) in
   inout-going-signals-block(inout, chid, block))
```

type: (IN | OUT *Channel-identifier*$_1$ (*Block-identifier*$_1$ | ENVIRONMENT)
Block-definition$_1$-**set** → *Signal-identifier*$_1$-**set**

Objective Extract the signals going in or out (indicated by the first function argument) through a connection point of a channel.

Parameters

inout Indicates whether the in- or outgoing signals are wanted.

chid The identifier of the channel the connection point signals of which are wanted.

endp The channel end point where connection point signals are wanted.

bset The set of blocks defined at the same scope unit level as the channel.

Result The set of in- or outgoing signals.

Algorithm

Line 1 If the channel end point is the system environment the set of in/outgoing signals is considered to be empty.

Line 4 Extract the block to which the channel is connected.

Line 5 Extract from the block the set of in-/outgoing signals at the connection point for the channel.

inout-going-signals-block(*inout*, *chid*, **mk**-*Block-definition*$_1$ (, , , *connects*, *srset*, , , *osub*)) ≜ (6.3.2.8)

```
1    if osub ≠ nil then
2       inout-going-signals-sub(inout, chid, osub)
3     else
4       (let mk-Channel-to-route-connection1(chidset, sridset) ∈ connects
5             be s.t. chid ∈ chidset in
6        let srset′ = {select-signalroute(srid, srset) | srid ∈ sridset} in
7        union {inout-going-signals-sigroute(inout, sr) | sr ∈ srset′})
```

type: (IN | OUT) *Channel-identifier*$_1$ *Block-definition*$_1$ → *Signal-identifier*$_1$-**set**

Objective Extract from a block the signals going in or out (indicated by the first function argument) through the connection point of a given channel.

Parameters

inout Indicates whether the in- or outgoing signals are wanted.

chid The identifier of the channel for which the connection point signals are wanted.

connects,*srset*,*osub* The channel to route connections, signal routes and substructure of the block.

Result The set of in- or outgoing signals.

Algorithm

Line 1-2 If the block is substructured the in-/outgoing signals are extracted from the substructure.

Line 4 Obtain the set of identifiers of signal routes connected to the channel.

Line 6 Obtain the set of signal routes connected to the channel.

Line 7 Extract from the connected signal routes the set of in-/outgoing signals.

inout-going-signals-sub(*inout*, *chid*, **mk**-*Block-substructure-definition*$_1$ (, , *connects*, *subchset*, , ,)) $\triangleq$ (6.3.2.9)

1 (**let mk**-*Channel-connection*$_1$(*chidset*, *subchidset*) ∈ *connects* **be s.t.** *chid* ∈ *chidset* **in**
2 **let** *subchset′* = {*select-channel*(*subchid*, *subchset*) | *subchid* ∈ *subchidset*} **in**
3 **union** {*inout-going-signals-chan*(*inout*, *subch*) | *subch* ∈ *subchset′*})

type: (IN | OUT) *Channel-identifier*$_1$ *Block-substructure-definition*$_1$
→ *Signal-identifier*$_1$-**set**

Objective Extract from a block substructure the signals going in or out (indicated by the first function argument) through the connection point of a given channel.

Parameters

inout	Indicates whether the in- or outgoing signals are wanted.
chid	The identifier of the channel the connection point signals of which are wanted.
connects,*subchset*	The channel connections and subchannels of the substructure.

Result The set of in- or outgoing signals.

Algorithm

Line 1	Obtain the set of identifiers of subchannels connected to the channel.
Line 2	Obtain the set of subchannels connected to the channel.
Line 3	Extract from the connected subchannels the set of in-/outgoing signals.

extract-direction-subsignals(*sigs*, *subsigs*, *subsigdir*)(*dict*) $\triangleq$ (6.3.2.10)

1 {*subsig* ∈ *subsigs* |
2 (∃ *sig* ∈ *sigs*)
3 (*is-sig-or-subsig*(*subsig*, *sig*) ∧ *subsig-direction*(*subsig*, *sig*)(*dict*) = *subsigdir*)}

type: *Signal-identifier*$_1$-**set** *Signal-identifier*$_1$-**set** [REVERSE] → *Entity-dict*
→ *Signal-identifier*$_1$-**set**

Objective Extract from a given set of (sub)signals the ones which are direct or indirect (sub)signals of signals in another set of signals. The third parameter of the function indicates whether the (sub)signals going in the same or in the opposite direction of its direct or indirect (parent) signal are wanted.

Parameters

sigs	The set of (parent) signals.
subsigs	The set of (sub)signals.
subsigdir	Indicates whether “forward” or “reverse” (sub)signals are wanted.

Result The extracted set of (sub)signals.

Algorithm

Line 1-3	Select each (sub)signal for which a direct or indirect (parent) signal exists and which has the same/opposite direction as the direct or indirect (parent) signal.

subsig-direction(*subsig*, *sig*)(*dict*) ≜ (6.3.2.11)

```
1    if subsig = sig then
2       nil
3     else
4       (let mk-SignalDD(, dir) = dict(subsig) in
5        let restdir = subsig-direction(parent-signal(subsig), sig)(dict) in
6         cases (dir, restdir):
7          ((nil, nil), (REVERSE, REVERSE)
8               → nil,
9           (nil, REVERSE), (REVERSE, nil)
10              → REVERSE))
```

type: *Signal-identifier*$_1$ *Signal-identifier*$_1$ → *Entity-dict* → [REVERSE]

Objective For two signals of which one is on the same or a different refinement level of the other, indicate whether the two signals go in the same or the opposite direction.

Parameters

subsig The (sub)signal.

sig The (parent) signal.

Result An indication of the relative direction.

Algorithm

Line 1-2 If the two signals are the same they go in the same direction.

Line 4 Find the direction of the subsignal relative to its parent signal.

Line 5 Find the direction of the parent signal of the subsignal relative to the signal *sig*.

Line 7 If the subsignal and *sig* have the same direction relative to the parent signal, they go in the same direction.

Line 9 If the subsignal and *sig* have opposite directions relative to the parent signal, they go in the opposite direction of each other.

is-sig-or-subsig(*subsig*, *sig*) ≜ (6.3.2.12)

```
1    subsig = sig ∨ is-proper-subsig(subsig, sig)
```

type: *Signal-identifier*$_1$ *Signal-identifier*$_1$ → *Bool*

Objective Test whether two signals are on the same or different refinement levels or not.

Parameters

subsig The “subsignal”.

sig The “parent signal”.

Result **true** if the two signals are the same or the former is on a finer refinement level of the latter, otherwise **false**.

Algorithm

Line 1 The condition holds if the signal *subsig* is either the same as *sig*, or *subsig* is a direct or indirect (proper) subsignal of *sig*.

is-proper-subsig(*subsig*, *sig*) $\triangleq$ (6.3.2.13)

1 $(\mathbf{let}\ \mathbf{mk}\text{-}\mathit{Identifier}_1(\mathit{qual}, \mathit{nm}) = \mathit{sig},$
2 $\quad \mathit{siglevel} = \mathit{qual} \frown \langle \mathbf{mk}\text{-}\mathit{Signal\text{-}qualifier}_1(\mathit{nm})\rangle\ \mathbf{in}$
3 $(\exists \mathit{qrest} \in \mathit{Path\text{-}item}_1{}^*)(\mathit{siglevel} \frown \mathit{qrest} = \mathbf{s}\text{-}\mathit{Qualifier}_1(\mathit{subsig})))$

type: $\mathit{Signal\text{-}identifier}_1\ \mathit{Signal\text{-}identifier}_1 \to \mathit{Bool}$

Objective Test whether one signal is on another refinement level than another signal.

Parameters

subsig The “subsignal”.

sig The “parent signal”.

Result **true** if the former signal is on a finer refinement level than another.

Algorithm

Line 1-2 Get the qualifier denoting the “parent signal” level.

Line 3 The signal *subsig* is on a finer refinement level than *sig* if the qualifier denoting the scope unit level of *sig* is a prefix of the qualifier contained in s*ubsig.*

parent-signal(*sig*) $\triangleq$ (6.3.2.14)

1 $(\mathbf{let}\ \mathbf{mk}\text{-}\mathit{Identifier}_1(\mathit{qual},) = \mathit{sig}\ \mathbf{in}$
2 $\mathbf{let}\ \mathit{qual}' = \langle \mathit{qual}[i] \mid 1 \le i < \mathbf{len}\ \mathit{qual}\rangle,$
3 $\quad \mathbf{mk}\text{-}\mathit{Signal\text{-}qualifier}_1(\mathit{nm}') = \mathit{qual}[\mathbf{len}\ \mathit{qual}]\ \mathbf{in}$
4 $\mathbf{mk}\text{-}\mathit{Identifier}_1(\mathit{qual}', \mathit{nm}'))$

type: $\mathit{Signal\text{-}identifier}_1 \to \mathit{Signal\text{-}identifier}_1$

Objective Get the parent signal of a signal.

Parameters

sig The signal.

Result The parent signal.

Algorithm

Line 1 Extract the qualifier of the signal.

Line 2-3 Get the qualifier and name of the parent signal.

Line 4 Construct the identifier of the parent signal.

6.4 Construction of Communication Paths

The functions in this section constructs the set of communication paths (*Reachability* sets) for all process instance sets and services in the SDL system which is going to be interpreted.

The way this construction is done is as follows:

1. For each internal channel/signal route path in a scope unit an outgoing and an ingoing partial *Reachability* set are constructed. Each member of the outgoing partial *Reachability* set is a partial *Reachability* containing an origin process or service, a sequence of signal route/channel paths leading to the given channel/signal route path, and the set of signals conveyed by this partial path. Analogously, each member of the ingoing partial *Reachability* set is a partial *Reachability* containing a destination process or service, a sequence of channel/signal route paths leading from the given channel/signal route path, and the set of signals conveyed by this partial path.
2. For each outgoing and ingoing partial *Reachability*, the outgoing partial path, the considered channel/signal route path and the ingoing partial path are concatenated, the intersection of the three corresponding signal sets is taken, and if this signal set is non-empty a (total) *Reachability* is constructed and inserted in the descriptor for the origin process or service.

For simplification of the *Reachability* construction, unidirectional channels and signal routes are treated as if they were bidirectional with an empty signal set in the reverse direction. Step 2 above ensures that this does not lead to extra *Reachabilities* in the final *Entity-dict.*

At system level, channels leading to or from the environment are treated like internal channels; however, in this case either the outgoing or ingoing partial *Reachability* set will not contain any “real” *Reachabilities* but instead be a singleton set containing the quotation (*Quot*) value ENVIRONMENT. At all lower scope unit levels the set of channels or signal routes leading to or from the scope unit boundary are not treated because they become part of the partial paths for internal channels/signal routes at higher scope unit levels.

For block internal signal route paths each of the two partial *Reachability* sets will contain “real” *Reachabilities* only if the corresponding process is decomposed into services. If the process is *not* decomposed into services and thus does not contain process internal service signal routes, the corresponding *Reachability* set will be a singleton set containing the identifier of the process.

In the comments attached to the functions below, the term *bridging* channel/signal route, or simply *bridge*, will be used. A bridging channel/signal route in a *Reachability* is the one which is defined at the highest scope unit level.

The entry function for construction of *Reachabilities is make-reachabilities.*

6.4.1 Reachability Construction

make-reachabilities(**mk**-*System-definition*$_1$(*snm, bset, cset, , ,*))(*dict*) $\triangleq$ (6.4.1.1)

(**let** *level* = ⟨**mk**-*System-qualifier*$_1$(*snm*)⟩ **in**
let *dict′* = *make-internal-reaches-chans*(*cset, bset, level*)(*dict*) **in**
let *dict″* = *make-internal-reaches-blocks*(*bset, level*)(*dict′*) **in**
dict″)

type: *System-definition*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the *Reachabilities* for an SDL system to be interpreted.

Parameters

snm,bset,cset The system name, block definitions and channel definitions in the system.

Result The *Entity-dict* where all *Reachabilities* have been inserted.

Algorithm

Line 1 Construct the system level qualifier.

Line 2 Construct the *Reachabilities* having the system level channels as bridges. Also channels leading from/to the system environment are treated here.

Line 3 Construct the internal *Reachabilities* of the system level blocks.

Line 4 Return the updated *Entity-dict.*

make-internal-reaches-blocks(*bset*, *level*)(*dict*) ≜ (6.4.1.2)

```
1   if bset = {} then
2     dict
3    else
4     (let block ∈ bset in
5      let dict′ = make-internal-reaches-block(block, level)(dict) in
6      make-internal-reaches-blocks(bset \ {block}, level)(dict′))
```

type: $Block\text{-}definition_1$**-set** $Qualifier_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the internal *Reachabilities* of a set of blocks.

Parameters

bset The set of block definitions.

level The qualifier of the enclosing system or substructure.

Result The *Entity-dict* where the block internal *Reachabilities* have been inserted.

Algorithm

Line 1-2 If the block set is empty the *Entity-dict* is not changed.

Line 4-5 Select a block and construct its internal *Reachabilities.*

Line 6 Construct the internal *Reachabilities* of the remaining blocks.

make-internal-reaches-block(**mk**-$Block\text{-}definition_1$(*bnm*, *pset*, , , *srset*, , , *osub*), *level*)(*dict*) ≜ (6.4.1.3)

```
1   (let level′ = level ⁀ ⟨mk-Block-qualifier1(bnm)⟩ in
2    if osub ≠ nil then
3      make-internal-reaches-sub(osub, level′)(dict)
4     else
5      (let srset′ = {sr ∈ srset | is-internal-sigroute(sr)} in
6       let dict′ = make-internal-reaches-sigroutes(srset′, pset, level′)(dict) in
7       let dict″ = make-internal-reaches-prcss(pset, level)(dict′) in
8       dict″))
```

type: $Block\text{-}definition_1$ $Qualifier_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the internal *Reachabilities* of a block.

Parameters

bnm,pset,srset,osub The block name, process definitions, signal routes and optional block substructure in the block.

level The qualifier of the enclosing system or substructure.

Result The *Entity-dict* where the block internal *Reachabilities* have been inserted.

Algorithm

Line 1	Construct the block level qualifier.
Line 2-3	If the block is substructured then the internal *Reachabilities* of the substructure is constructed.
Line 5	Select those signal routes which are internal to the block.
Line 6	Construct the *Reachabilities* having the block internal signal routes as bridges.
Line 7	Construct the internal *Reachabilities* of the block local processes.
Line 8	Return the updated *Entity-dict.*

make-internal-reaches-sub(**mk**-*Block-substructure-definition*$_1$(*bsnm*, *bset*, , *cset*, , ,), *level*)(*dict*) ≜ (6.4.1.4)

```
1   (let level′ = level ⁀ ⟨mk-Block-substructure-qualifier1(bsnm)⟩ in
2    let cset′ = {chan ∈ cset | is-internal-chan(chan)} in
3    let dict′ = make-internal-reaches-chans(cset′, bset, level′)(dict) in
4    let dict″ = make-internal-reaches-blocks(bset, level′)(dict′) in
5    dict″)
```

type: *Block-substructure-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the internal *Reachabilities* of a block substructure.

Parameters

bsnm,*bset*,*cset*	The block substructure name, subblock definitions and subchannels in the block substructure.
level	The qualifier of the enclosing block.

Result The *Entity-dict* where the block substructure internal *Reachabilities* have been inserted.

Algorithm

Line 1	Construct the substructure level qualifier.
Line 2	Select those subchannels which are internal to the substructure.
Line 3	Construct the *Reachabilities* having the substructure internal channels as bridges.
Line 4	Construct the *Reachabilities* of the substructure local blocks.
Line 5	Return the updated *Entity-dict.*

make-internal-reaches-prcss(*pset*, *level*)(*dict*) ≜ (6.4.1.5)

```
if pset = {} then
  dict
 else
  (let prcs ∈ pset in
   let dict′ = make-internal-reaches-prcs(prcs, level)(dict) in
   make-internal-reaches-prcss(pset \ {prcs}, level)(dict′))
```

type: *Process-definition*$_1$-**set** *Qualifier*$_1$ → *Entity dict* → *Entity dict*

Objective Construct the internal *Reachabilities* of a set of process definitions.

Parameters

pset	The set of process definitions.
level	The qualifier of the enclosing block.

Result The *Entity-dict* where the process internal *Reachabilities* have been inserted.

Algorithm

Line 1-2 If the set of process definitions is empty the *Entity-dict* is not changed.

Line 4-5 Select a process definition and construct its internal *Reachabilities.*

Line 6 Construct the internal *Reachabilities* of the remaining process definitions.

make-internal-reaches-prcs(*prcs, level*)(*dict*) ≜ (6.4.1.6)

```
1   (let mk-Process-definition₁(prnm, , , , , , , , , , grordec) = prcs in
2    (is-Process-graph₁(grordec)
3        → (let prid = mk-Identifier₁(level, prnm),
4               sigs = extract-inputsigs-prcs(prcs) in
5             update-endpd-self(prid, sigs)(dict)),
6     is-Service-decomposition₁(grordec)
7        → (let level′ = level ⁀ ⟨mk-Process-qualifier₁(prnm)⟩ in
8             make-internal-reaches-decomp(grordec, level′)(dict))))
```

type: *Process-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the internal *Reachabilities* of a process definition.

Parameters

prcs The process definition.

level The qualifier of the enclosing block.

Result The *Entity-dict* where the process internal *Reachabilities* have been inserted.

Algorithm

Line 1 Get the name and process graph/service decomposition of the process.

Line 2 Handle the case where the process is *not* decomposed into services.

Line 3 Construct the identifier of the process.

Line 4 Extract the input signal set of the process.

Line 5 Construct a *Reachability* from the process to itself and insert it in the *Entity-dict.*

Line 6 Handle the case where the process is decomposed into services.

Line 7 Construct the process level qualifier.

Line 8 Construct the internal *Reachabilities* of the service decomposition.

make-internal-reaches-decomp(**mk**-*Service-decomposition*$_1$(*servset, srset,*), *level*)(*dict*) ≜ (6.4.1.7)

```
(let srset′ = {sr ∈ srset | is-internal-sigroute(sr)} in
 let dict′ = make-internal-reaches-servsigroutes(srset′, level)(dict) in
 let dict″ = make-internal-reaches-servs(servset, level)(dict′) in
 dict″)
```

type: *Service-decomposition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the internal *Reachabilities* of a service decomposition.

Parameters

servset,srset — The service definitions and signal routes in the decomposition.

level — The qualifier of the enclosing process.

Result — The *Entity-dict* where the decomposition internal *Reachabilities* have been inserted.

Algorithm

Line 1 — Select those signal routes which are internal to the decomposition.

Line 2 — Construct the *Reachabilities* having the decomposition internal signal routes as bridges.

Line 3 — Construct the internal *Reachabilities* of the (decomposition local) service definitions.

Line 4 — Return the updated *Entity-dict.*

make-internal-reaches-servs(*servset*, *level*)(*dict*) ≜ (6.4.1.8)

```
1    if servset = {} then
2      dict
3     else
4      (let serv ∈  servset in
5       let dict′ = make-internal-reaches-serv(serv, level)(dict) in
6       make-internal-reaches-servs(servset \ {serv}, level)(dict′))
```

type: $\mathit{Service\text{-}definition}_1$**-set** $\mathit{Qualifier}_1 \rightarrow \mathit{Entity\text{-}dict} \rightarrow \mathit{Entity\text{-}dict}$

Objective — Construct the internal *Reachabilities* of a set of service definitions.

Parameters

servset — The set of service definitions.

level — The qualifier of the enclosing process definition.

Result — The *Entity-dict* where the service internal *Reachabilities* have been inserted.

Algorithm

Line 1-2 — If the service set is empty the *Entity-dict* is not changed.

Line 4-5 — Select a service and construct its internal *Reachabilities.*

Line 6 — Construct the internal *Reachabilities* of the remaining services.

make-internal-reaches-serv (*serv*, *level*)(*dict*) ≜ (6.4.1.9)

```
1    (let servid = mk-Identifier1(level, s-Service-name1(serv)),
2           sigs = extract-inputsigs-serv(serv) in
3     update-endpd-self(servid, sigs)(dict))
```

type: $\mathit{Service\text{-}definition}_1\ \mathit{Qualifier}_1 \rightarrow \mathit{Entity\text{-}dict} \rightarrow \mathit{Entity\text{-}dict}$

Objective — Construct the internal *Reachabilities* of a service definition.

Parameters

serv — The service definition.

level — The qualifier of the enclosing process definition.

Result The *Entity-dict* where the service internal *Reachabilities* have been inserted.

Algorithm

Line 1 Construct the identifier of the service.

Line 2 Extract the input signal set of the service.

Line 3 Construct a *Reachability* from the service to itself and insert it in the *Entity-dict.* The input signal set will also be inserted in the service descriptor.

make-internal-reaches-chans(*cset*, *bset*, *level*)(*dict*) ≜ (6.4.1.10)

```
1    if cset = {} then
2      dict
3     else
4      (let chan ∈ cset in
5       let dict′ = make-internal-reaches-chan(chan, bset, level)(dict) in
6       make-internal-reaches-chans(cset \ {chan}, bset, level)(dict′))
```

type: *Channel-definition*$_1$-**set** *Block-definition*$_1$-**set** *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the set of *Reachabilities* having a given set of channels as bridges.

Parameters

cset The set of channel definitions.

bset The set of blocks at the same scope unit level as the channels.

level The qualifier of the enclosing system or substructure.

Result The *Entity-dict* where the *Reachabilities* having the given channels as bridges have been inserted.

Algorithm

Line 1-2 If the channel set is empty the *Entity-dict* is not modified.

Line 4-5 Select a channel and construct the *Reachabilities* having this channel as bridge.

Line 6 Construct the *Reachabilities* having the remaining channels as bridges.

make-internal-reaches-chan(*chan*, *bset*, *level*)(*dict*) ≜ (6.4.1.11)

```
(let mk-Channel-definition1(chnm, nodelay, mk-Channel-path1(endp1, endp2, ), ) = chan in
 let chid = mk-Identifier1(level, chnm) in
 let foutreaches = inout-going-reaches(OUT, chid, endp1, bset, level),
     fpathelem = (chid, FORWARD, nodelay),
     fsigs = direction-signals-chan(chan, FORWARD),
     finreaches = inout-going-reaches(IN, chid, endp2, bset, level) in
 let routreaches = inout-going-reaches(OUT, chid, endp2, bset, level),
     rpathelem = (chid, REVERSE, nodelay),
     rsigs = direction-signals-chan(chan, REVERSE),
     rinreaches = inout-going-reaches(IN chid, endp1, bset, level) in
 let dict′ = update-endpd(foutreaches, fpathelem, fsigs, finreaches)(dict) in
 let dict″ = update-endpd(routreaches, rpathelem, rsigs, rinreaches)(dict′) in
 dict″)
```

type: *Channel-definition*$_1$ *Block-definition*$_1$-**set** *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the set of *Reachabilities* having a given channel as bridge.

Parameters

chan The channel definition.

bset The set of blocks at the same scope unit level as the channel.

level The qualifier of the enclosing system or substructure.

Result The *Entity-dict* where the *Reachabilities* having the given channel as bridge have been inserted.

Algorithm

Line 1 Obtain the name, optional **nodelay** attribute and origin and destination end point of the channel.

Line 2 Construct the identifier of the channel.

Line 3-6 Obtain the outgoing partial *Reachability* set leading to the origin end point of the channel (line 3), the *Path-element* denoting the forward channel path (line 4), the set of signals carried in the forward direction by the channel (line 5), and the ingoing partial *Reachability* set leading from the destination end point of the channel (line 6).

Line 7-10 Analogously to line 3-6, obtain the outgoing partial *Reachability* set leading to the destination end point of the channel (line 7), the *Path-element* denoting the reverse channel path (line 8), the set of signals carried in the reverse direction by the channel (empty if the channel is unidirectional) (line 9), and the ingoing partial *Reachability* set leading from the origin end point of the channel (line 10).

Line 11 Construct the total *Reachabilities* having the forward channel path as bridge.

Line 12 Analogously to line 11, construct the total *Reachabilities* having the reverse channel path as bridge.

Line 13 Return the updated *Entity-dict.*

make-internal-reaches-sigroutes(*srset, pset, level*)(*dict*) $\triangleq$ (6.4.1.12)

if *srset* = {} **then**
  dict
 else
  (**let** *sr* ∈ *srset* **in**
   let *dict′* = *make-internal-reaches-sigroute*(*sr, pset, level*)(*dict*) **in**
   make-internal-reaches-sigroutes(*srset* \ {*sr*}, *pset, level*)(*dict′*))

type: *Signal-route-definition*$_1$-**set** *Process-definition*$_1$-**set** *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the set of *Reachabilities* having a given set of signal routes as bridges. The function is analogous to *make-internal-reaches-chans.*

Parameters

srset The set of signal route definitions.

pset The set of process definitions at the same scope unit level as the signal routes.

level The qualifier of the enclosing block.

Result The *Entity-dict* where the *Reachabilities* having the given signal routes as bridges have been inserted.

Algorithm

Line 1-2 If the signal route set is empty the *Entity-dict* is not modified.

Line 4-5 Select a signal route and construct the *Reachabilities* having this signal route as bridge.

Line 6 Construct the *Reachabilities* having the remaining signal routes as bridges.

make-internal-reaches-sigroute(*sr*, *pset*, *level*)(*dict*) ≜ (6.4.1.13)

```
1   (let mk-Signal-route-definition_1(srnm, mk-Signal-route-path_1(endp1, endp2, ), ) = sr in
2    let srid = mk-Identifier_1(level, srnm) in
3    let foutreaches = inout-going-reaches′(OUT, srid, endp1, pset, level),
4        fpathelem = (srid, FORWARD, NODELAY),
5        fsigs = direction-signals-sigroute(sr, FORWARD),
6        finreaches = inout-going-reaches′(IN, srid, endp2, pset, level) in
7    let routreaches = inout-going-reaches′(OUT, srid, endp2, pset, level),
8        rpathelem = (srid, REVERSE, NODELAY),
9        rsigs = direction-signals-sigroute(sr, REVERSE),
10       rinreaches = inout-going-reaches′(IN, srid, endp1, pset, level) in
11   let dict′ = update-endpd(foutreaches, fpathelem, fsigs, finreaches)(dict) in
12   let dict″ = update-endpd(routreaches, rpathelem, rsigs, rinreaches)(dict′) in
13   dict″)
```

type: *Signal-route-definition*$_1$ *Process-definition*$_1$-**set** *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct the set of *Reachabilities* having a given signal route as bridge. The function is analogous to *make-internal-reaches-chan.*

Parameters

sr The signal route definition.

pset The set of process definitions at the same scope unit level as the signal route.

level The qualifier of the enclosing block.

Result The *Entity-dict* where the *Reachabilities* having the given signal route as bridge have been inserted.

Algorithm

Line 1 Obtain the name and origin and destination end point of the signal route.

Line 2 Construct the identifier of the signal route.

Line 3-6 Obtain the outgoing partial *Reachability* set leading to the origin end point of the signal route (line 3), the *Path-element* denoting the forward signal route path (line 4), the set of signals carried in the forward direction by the signal route (line 5), and the ingoing partial *Reachability* set leading from the destination end point of the signal route (line 6). The *Path-element* for the signal route path always contains NODELAY because a signal route never has a delay.

Line 7-10 Analogously to line 3-6, obtain the outgoing partial *Reachability* set leading to the destination end point of the signal route (line 7), the *Path-element* denoting the reverse signal route path (line 8), the set of signals carried in the reverse direction by the signal route (empty if the signal route is unidirectional) (line 9), and the ingoing partial *Reachability* set leading from the origin end point of the signal route (line 10).

Line 11 Construct the total *Reachabilities* having the forward signal route path as bridge.

Line 12 Analogously to line 11, construct the total *Reachabilities* having the reverse signal route path as bridge.

Line 13 Return the updated *Entity-dict.*

make-internal-reaches-servsigroutes(*srset*, *level*)(*dict*) ≜ (6.4.1.14)

```
1   if srset = { } then
2     dict
3    else
4     (let sr ∈ srset in
5      let dict′ = make-internal-reaches-servsigroute(sr, level)(dict) in
6      make-internal-reaches-servsigroutes(srset \ {sr}, level)(dict′))
```

type: $\textit{Signal-route-definition}_1$**-set** $\textit{Qualifier}_1 \to \textit{Entity-dict} \to \textit{Entity-dict}$

Objective Construct the set of *Reachabilities* having a given set of (service decomposition internal) signal routes as bridges. The function is analogous to *make-internal-reaches-chans* and *make-internal-reaches-sigroutes.*

Parameters

srset The set of signal route definitions.

level The qualifier of the enclosing process definition.

Result The *Entity-dict* where the *Reachabilities* having the given signal routes as bridges have been inserted.

Algorithm

Line 1-2 If the signal route set is empty the *Entity-dict* is not modified.

Line 4-5 Select a signal route and construct the *Reachabilities* having this signal route as bridge.

Line 6 Construct the *Reachabilities* having the remaining signal routes as bridges.

make-internal-reaches-servsigroute(*sr*, *level*)(*dict*) ≜ (6.4.1.15)

```
1   (let mk-Signal-route-definition1(srnm, mk-Signal-route-path1(endp1, endp2, ), ) = sr in
2    let srid = mk-Identifier1(level, srnm) in
3    let foutreaches = {endp1},
4        fpathelem = (srid, FORWARD, NODELAY),
5        fsigs = direction-signals-sigroute(sr, FORWARD),
6        finreaches = {endp2} in
7    let routreaches = {endp2} ,
8        rpathelem = (srid, REVERSE, NODELAY),
9        rsigs = direction-signals-sigroute(sr, REVERSE),
10       rinreaches = {endp1} in
11   let dict′ = update-endpd(foutreaches, fpathelem, fsigs, finreaches)(dict) in
12   let dict″ = update-endpd(routreaches, rpathelem, rsigs, rinreaches)(dict′) in
13   dict″)
```

type: $\textit{Signal-route-definition}_1\ \textit{Qualifier}_1 \to \textit{Entity-dict} \to \textit{Entity-dict}$

Objective Construct the set of *Reachabilities* having a given (service decomposition internal) signal route as bridge. The function is analogous to *make-internal-reaches-chan* and *make-internal-reaches-sigroute.*

Parameters

sr The signal route definition.

level The qualifier of the enclosing process definition.

Result The *Entity-dict* where the *Reachabilities*, having the given signal route as bridge have been inserted.

Algorithm

Line 1 Obtain the name and origin and destination end point of the signal route.

Line 2 Construct the identifier of the signal route.

Line 3-6 Obtain the outgoing partial *Reachability* set leading to the origin end point of the signal route (line 3), the *Path-element*, denoting the forward signal route path (line 4), the set of signals carried in the forward direction by the signal route (line 5), and the ingoing partial *Reachability* set leading from the destination end point of the signal route (line 6). As services do not contain signal routes both partial *Reachability* sets are singleton sets containing the respective end point (service) identifier. The *Path-element* for the signal route path always contains NODELAY because a signal route never has a delay.

Line 7-10 Analogously to line 3-6, obtain the outgoing partial *Reachability* set leading to the destination end point of the signal route (line 7), the *Path-element* denoting the reverse signal route path (line 8), the set of signals carried in the reverse direction by the signal route (empty if the signal route is unidirectional) (line 9), and the ingoing partial *Reachability* set leading from the origin end point of the signal route (line 10).

Line 11 Construct the total *Reachabilities* having the forward signal route path as bridge.

Line 12 Analogously to line 11, construct the total *Reachabilities* having the reverse signal route path as bridge.

Line 13 Return the updated *Entity-dict*.

6.4.2 Construction of Partial Reachabilities

inout-going-reaches(*inout, chid, endp, bset, level*) $\triangleq$ (6.4.2.1)

1 **if** *endp* = ENVIRONMENT **then**
2 {ENVIRONMENT}
3 **else**
4 (**let** *block* = *select-block*(*endp, bset*) **in**
5 *inout-going-reaches-block*(*inout, chid, block, level*))

type: (IN | OUT) *Channel-identifier*$_1$ (*Block-identifier*$_1$ | ENVIRONMENT) *Block-definition*$_1$-**set** *Qualifier*$_1$ $\rightarrow$ (ENVIRONMENT | *Reachability*)-**set**

Objective Obtain the in- or outgoing partial *Reachability* set (direction indicated by the first function argument) leading from/to a given channel end point.

Parameters

inout Indicates whether the in- or outgoing partial *Reachability* set is wanted.

chid The identifier of the bridging channel.

endp The channel end point (may be the **env** in case of a system level channel) at which the partial *Reachabilities* are wanted.

bset The set of blocks defined at the same scope unit level as the channel.

level The qualifier of the enclosing system or substructure.

Result The partial *Reachability* set, or a singleton set containing the *Quot* value ENVIRONMENT if the channel end point is **env**.

Algorithm

Line 1-2 If the channel end point is **env** the singleton *Reachability* set containing ENVIRONMENT is returned.

Line 4 Get the end point block definition from the block set.

Line 5 Extract from the block the in-/outgoing partial *Reachability* set.

inout-going-reaches′(*inout, srid, endp, pset, level*) $\triangleq$ (6.4.2.2)

(**let** *prcs* = *select-process*(*endp, pset*) **in**
inout-going-reaches-prcs(*inout, srid, prcs, level*))

type: (IN | OUT) *Signal-route-identifier*$_1$ *Process-identifier*$_1$*Process-definition*$_1$-**set** *Qualifier*$_1$ $\rightarrow$ (*Process-identifier*$_1$ | *Reachability*)-**set**

Objective Obtain the in- or outgoing partial *Reachability* set (direction indicated by the first function argument) leading from/to a given signal route end point. The function is analogous to *inout-going-reaches.*

Parameters

inout Indicates whether the in-or outgoing partial *Reachability* set is wanted.

srid The identifier of the bridging signal route.

endp The signal route end point at which the partial *Reachabilities* are wanted.

pset The set of process definitions at the same scope unit level as the signal route.

level The qualifier of the enclosing block.

Result The partial *Reachability* set, or a singleton set containing a process identifier if the denoted process instance set is not decomposed into services.

Algorithm

Line 1 Get the end point process definition from the set of process definitions.

Line 2 Extract the in-/outgoing partial *Reachability* set.

inout-going-reaches-block(*inout, chid, block, level*) ≜ (6.4.2.3)

1 (**let mk**-*Block-definition*$_1$(*bnm, pset, , connects, srset, , , osub*) = *block* **in**
2 **let** *level′* = *level* ⁀ ⟨**mk**-*Block-qualifier*$_1$(*bnm*)⟩ **in**
3 **if** *osub* ≠ **nil then**
4 *inout-going-reaches-sub*(*inout, chid, osub, level′*)
5 **else**
6 (**let mk**-*Channel-to-route-connection*$_1$(*chidset, sridset*) ∈ *connects*
7 **be s.t.** *chid* ∈ *chidset* **in**
8 **let** *srset′* = {*select-signalroute*(*srid, srset*) | *srid* ∈ *sridset*} **in**
9 **union** {*inout-going-reaches-sigroute*(*inout, sr, pset, level′*) | *sr* ∈ *srset′*}))

type: (IN | OUT) *Channel-identifier*$_1$ *Block-definition*$_1$ *Qualifier*$_1$ → *Reachability*-**set**

Objective Obtain from a block the in-/outgoing partial *Reachability* set leading from/to a given channel.

Parameters

inout Indicates whether the in- or outgoing *Reachabilities* are wanted.

chid The identifier of the channel.

block The block definition.

level The qualifier of the enclosing system or substructure.

Result The in-/outgoing partial *Reachabilities.*

Algorithm

Line 1-2 Decompose the block and construct the qualifier denoting its level.

Line 3-4 If the block is substructured the in-/outgoing partial *Reachabilities* are extracted from the substructure.

Line 6 Obtain the set of identifiers of signal routes connected to the channel.

Line 8 Obtain the set of signal routes connected to the channel.

Line 9 Construct all in-/outgoing partial *Reachabilities* leading from/to and including one of the signal routes.

inout-going-reaches-sub(*inout, chid, sub, level*) ≜ (6.4.2.4)

(**let mk**-*Block-substructure-definition*$_1$ (*bsnm, bset, connects, subchset, , ,*) = *sub* **in**
let *level′* = *level* ⁀ ⟨**mk**-*Block-substructure-qualifier*$_1$(*bsnm*)⟩ **in**
let mk-*Channel-connection*$_1$(*chidset, subchidset*) ∈ *connects* **be s.t.** *chid* ∈ *chidset* **in**
let *subchset′* = {*select-channel*(*subchid, subchset*) | *subchid* ∈ *subchidset*} **in**
union {*inout-going-reaches-chan*(*inout, subchan, bset, level′*) | *subchan* ∈ *subchset′*})

type: (IN | OUT) *Channel-identifier*$_1$ *Block-substructure-definition*$_1$ *Qualifier*$_1$
→ *Reachability*-**set**

Objective Obtain from a block substructure the in-/outgoing partial *Reachability* set leading from/to a given channel.

Parameters

inout Indicates whether the in- or outgoing *Reachabilities* are wanted.

chid The identifier of the channel.

sub The block substructure definition.

level The qualifier of the enclosing block.

Result The in-/outgoing partial *Reachabilities.*

Algorithm

Line 1-2 Decompose the block substructure and construct the qualifier denoting its level.

Line 3 Obtain the set of identifiers of subchannels connected to the channel.

Line 4 Obtain the set of subchannels connected to the channel.

Line 5 Construct all in-/outgoing partial *Reachabilities* leading from/to and including one of the subchannels.

inout-going-reaches-prcs(*inout, srid, prcs, level*) ≜ (6.4.2.5)

1 (**let mk**-*Process-definition*$_1$(*prnm, , , , , , , , , , grordec*) = *prcs* **in**
2 (**is**-*Process-graph*$_1$(*grordec*)
3 → {**mk**-*Identifier*$_1$(*level, prnm*)},
4 **is**-*Service-decomposition*$_1$(*grordec*)
5 → (**let** *level′* = *level* ⁀ ⟨**mk**-*Process-qualifier*$_1$(*prnm*)⟩ **in**
6 *inout-going-reaches-decomp*(*inout, srid, grordec, level′*))))

type: (IN | OUT) *Signal-route-identifier*$_1$ *Process-definition*$_1$-**set** *Qualifier*$_1$
→ (*Process-identifier*$_1$ | *Reachability*)-**set**

Objective Obtain from a process definition the in-/outgoing partial *Reachability* set leading from/to a given signal route.

Parameters

inout Indicates whether the in- or outgoing *Reachabilities* are wanted.

srid The identifier of the signal route.

prcs The process definition.

level The qualifier of the enclosing block.

Result The in-/outgoing partial *Reachabilities.*

Algorithm

Line 1 Decompose the process definition.

Line 2-3 If the process is not decomposed into service instances then the singleton *Reachability* set containing its identifier is returned.

Line 4-6 Otherwise the in-/outgoing *Reachabilities* are extracted from the service decomposition.

inout-going-reaches-decomp(*inout*, *srid*, *decomp*, *level*) $\triangleq$ (6.4.2.6)

1 (**let mk**-*Service-decomposition*$_1$(, *servsrset*, *connects*) = *decomp* **in**
2 **let mk**-*Signal-route-to-route-connection*$_1$(*sridset*, *servsridset*) ∈ *connects*
3 **be s.t.** *srid* ∈ *sridset* **in**
4 **let** *servsrset′* = {*select-signalroute*(*servsrid*, *servsrset*) | *servsrid* ∈ *servsridset*} **in**
5 {*inout-going-reaches-sigroute*(*inout*, *servsr*, *level*) | *servsr* ∈ *servsrset′*})

type: (IN | OUT) *Signal-route-identifier*$_1$ *Service-decomposition*$_1$ *Qualifier*$_1$
→ *Reachability*-**set**

Objective Obtain from a service definition the partial *Reachability* set leading from/to a given (block level) signal route.

Parameters

inout Indicates whether the in- or outgoing *Reachabilities* are wanted.

srid The identifier of the signal route.

decomp The service decomposition.

level The qualifier of the enclosing process.

Result The in-/outgoing partial *Reachabilities.*

Algorithm

Line 1 Decompose the service decomposition.

Line 2 Obtain the set of identifiers of service signal routes connected to the signal route.

Line 4 Obtain the set of service signal routes connected to the signal route.

Line 5 Construct all in-/outgoing partial *Reachabilities* leading from/to and including one of the service signal routes.

inout-going-reaches-chan(*inout*, *chan*, *bset*, *level*) $\triangleq$ (6.4.2.7)

1 (**let** *chid* = **mk**-*Identifier*$_1$(*level*, **s**-*Channel-name*$_1$(*chan*)),
2 *block* = *select-block*(*connected-block*(*chan*), *bset*) **in**
3 **let** *inoutreaches* = *inout-going-reaches-block*(*inout*, *chid*, *block*, *level*) **in**
4 {*append-chan-to-reach*(*inout*, *inoutreach*, *chan*, *level*) | *inoutreach* ∈ *inoutreaches*})

type: (IN | OUT) *Channel-definition*$_1$ *Block- definition*$_1$-**set** *Qualifier*$_1$ → *Reachability*-**set**

Objective Obtain the in-/outgoing partial *Reachability* set leading from/to and including a given non-local channel.

Parameters

inout Indicates whether the in- or outgoing partial *Reachabilities* are wanted.

chan The channel definition.

bset The set of blocks defined at the same scope unit level as the channel.

level The qualifier of the enclosing block substructure (system level non-local channels are treated like local channels).

Result The in-/outgoing partial *Reachabilities.*

Algorithm

Line 1 Construct the identifier of the channel.

Line 2 Get the block connected to the channel.

Line 3 Obtain the in-/outgoing partial *Reachabilities* leading to the channel.

Line 4 Append the channel to each of the partial *Reachabilities.*

append-chan-to-reach(*inout, inoutreach, chan, level*) ≜ (6.4.2.8)

1 (**let** *chansigs* = *inout-going-signals-chan*(*inout, chan*),
2 *chanpathelem* = *inout-going-path-elem-chan*(*inout, chan, level*) **in**
3 **let** (*reachendp, sigset, path*) = *inoutreach* **in**
4 (*reachendp, sigset* ∩ *chansigs,*
5 **cases** *inout*:
6 (IN → ⟨*chanpathelem*⟩ ⁀ *path,*
7 OUT → *path* ⁀ ⟨*chanpathelem*⟩)))

type: (IN | OUT) *Reachability Channel-definition*$_1$ *Qualifier*$_1$ → *Reachability*

Objective Append a non-local channel to an in-/outgoing partial *Reachability*.

Parameters

inout Indicates whether the partial *Reachability* is in- or outgoing.

inoutreach The partial *Reachability.*

chan The definition of the channel.

level The qualifier of the enclosing block substructure.

Result The partial *Reachability* where the channel has been appended.

Algorithm

Line 1 Extract the signals carried by the channel in the direction indicated by *inout.* Note that if the channel is unidirectional in the opposite direction the extracted signal set is empty.

Line 2 Construct the *Path-element* for the channel path which goes in the direction indicated by *inout.*

Line 3 Decompose the partial *Reachability* into its destination/origin end point, the set of signals which can be carried along the partial *Reachability*, and its sequence of *Path-elements.*

Line 4-7 The new partial *Reachability* is constructed as follows: Its destination/origin end point is that obtained before (line 4), the set of signal it can carry is the intersection of the signal set obtained before and the signal set from the channel (line 4), and its sequence of *Path-elements* is the sequence obtained before with the channel *Path-element* added at its “outer” end point (line 5-7).

inout-going-path-elem-chan(*inout, chan, level*) ≜ (6.4.2.9)

1 (**let mk**-*Channel-definition*$_1$(*chnm, nodelay*, ,) = *chan* **in**
2 (**mk**-*Identifier*$_1$(*level, chnm*),
3 *inout-going-path-direction-chan*(*inout, chan*),
4 *nodelay*))

type: (IN | OUT) *Channel-definition*$_1$ *Qualifier*$_1$ → *Path-element*

Objective Obtain the in- or outgoing *Path-element* for a non-local channel.

Parameters

inout Indicates whether the in- or outgoing *Path-element* is wanted.

chan The definition of the channel.

level The qualifier of the enclosing block substructure.

Result The *Path-element.*

Algorithm

Line 1 Get the name and **nodelay** attribute of the channel.

Line 2-4 Return the *Path-element* consisting of the identifier of the channel (line 2), the direction (forward or reverse) of the in-/outgoing channel direction indicated by *inout* (line 3), and the **nodelay** attribute (line 4).

inout-going-reaches-sigroute(*inout*, *sr*, *pset*, *level*) ≜ (6.4.2.10)

1 (**let** *srid* = **mk** *Identifier*$_1$(*level*, **s**-*Signal-route-name*$_1$(*sr*)),
2 *prcs* = *select-process*(*connected-process-or-service*(*sr*), *pset*) **in**
3 **let** *inoutreaches* = *inout-going-reaches-prcs*(*inout*, *srid*, *prcs*, *level*) **in**
4 {*append-sigroute-to-reach*(*inout*, *inoutreach*, *sr*, *level*) | *inoutreach* ∈ *inoutreaches*})

type: (IN | OUT) *Signal-route-definition*$_1$ *Process-definition*$_1$-**set** *Qualifier*$_1$ → *Reachability*-**set**

Objective Obtain the in-/outgoing partial *Reachability* set leading from/to and including a given non-local (block level) signal route. The function is analogous to *inout-going-reaches-chan.*

Parameters

inout Indicates whether the in- or outgoing partial *Reachabilities* are wanted.

sr The signal route definition.

pset The set of processes defined at the same scope unit level as the signal route.

level The qualifier of the enclosing block.

Result The in-/outgoing partial *Reachabilities.*

Algorithm

Line 1 Construct the identifier of the signal route.

Line 2 Get the process connected to the signal route.

Line 3 Obtain the in-/outgoing partial *Reachabilities* leading to the signal route.

Line 4 Append the signal route to each of the partial *Reachabilities.*

append-sigroute-to-reach(*inout*, *inoutreach*, *sr*, *level*) ≜ (6.4.2.11)

1 (**let** *srsigs* = *inout-going-signals-sigroute*(*inout*, *sr*),
2 *srpathelem* = *inout-going-path-elem-sigroute*(*inout*, *sr*, *level*) **in**
3 (**is**-*Identifier*$_1$(*inoutreach*)
4 → (*inoutreach*, *srsigs*, ⟨*srpathelem*⟩),
5 **is**-*Reachability*(*inoutreach*)
6 → (**let** (*reachendp*, *sigset*, *path* = *inoutreach* **in**
7 (*reachendp*, *sigset* ∩ *srsigs*,
8 **cases** *inout*:
9 (IN → ⟨*srpathelem*⟩ ⌢ *path*,
10 OUT → *path* ⌢ ⟨*srpathelem*⟩)))))

type: (IN | OUT) (*Process-Identifier*$_1$ | *Reachability*) *Signal-route-definition*$_1$ *Qualifier*$_1$ → *Reachability*

Objective Append a non-local (block level) signal route to an in-/outgoing partial *Reachability*. The function is analogous to *append-chan-to-reach*.

Parameters

inout Indicates whether the partial *Reachability* is in- or outgoing.

inoutreach A partial *Reachability* or a process identifier.

sr The definition of the signal route.

level The qualifier of the enclosing block.

Result The partial *Reachability* where the signal route has been appended.

Algorithm

Line 1 Extract the signals carried by the signal route in the direction indicated by *inout*. Note that if the signal route is unidirectional in the opposite direction the extracted signal set is empty.

Line 2 Construct the *Path-element* for the signal route path which goes in the direction indicated by *inout*.

Line 3-4 If the partial *Reachability* is a (process) identifier the resulting partial *Reachability* has this process as destination/origin (depending on *inout*) end point, the signals carried by the signal route path as signal set, and a *Path* consisting of the signal route path only.

Line 5 Handle the case where the partial *Reachability* is a "real" one.

Line 6 Decompose the partial *Reachability* into its destination/origin end point, the set of signals which can be carried along the partial *Reachability*, and its sequence of *Path-elements*.

Line 7-10 The new partial *Reachability* is constructed as follows: Its destination/origin end point is that obtained before (line 7), the set of signal it can carry is the intersection of the signal set obtained before and the signal set from the signal route (line 7), and its sequence of *Path-elements* is the sequence obtained before with the signal route *Path-element* added at its "outer" end point (line 8-10).

inout-going-reach-sigroute(*inout*, *sr*, *level*) $\triangleq$ (6.4.2.12)

```
1   (connected-process-or-service(sr),
2    inout-going-signals-sigroute(inout, sr),
3    ⟨inout-going-path-elem-sigroute(inout, sr, level)⟩)
```

type: (IN | OUT) *Signal-route-definition*$_1$ *Qualifier*$_1$ $\rightarrow$ *Reachability*

Objective Obtain the in-/outgoing partial *Reachability* consisting of a non-local (process level) signal route.

Parameters

inout Indicates whether the in- or outgoing partial *Reachability* is wanted.

sr The definition of the signal route.

level The qualifier of the enclosing process.

Result The partial *Reachability*.

Algorithm

Line 1-3 Construct and return the partial *Reachability* as follows: Its destination/origin (depending on *inout*) end point is the service connected to the signal route (line 1), its signal set is the set of signals carried by the signal route in the given direction (line 2), and its *Path* consists of the in-/outgoing path of the signal route (line 3).

inout-going-path-sigroute(*inout*, *sr*, *level*) $\triangleq$ (6.4.2.13)

1 (**mk**-$Identifier_1$(*level*, **s**-$Signal\text{-}route\text{-}name_1$(*sr*)),
2 *inout-going-path-direction-sigroute*(*inout*, *sr*),
3 NODELAY)

type: (IN | OUT) $Signal\text{-}route\text{-}definition_1$ $Qualifier_1$ → *Path-element*

Objective Obtain the in- or outgoing *Path-element* for a non-local signal route. The function is analogous to *inout-going-path-elem-chan.*

Parameters

inout Indicates whether the in- or outgoing *Path-element* is wanted.

chan The definition of the signal route.

level The qualifier of the enclosing block or process.

Result The *Path-element.*

Algorithm

Line 1-3 Return the *Path-element* consisting of the identifier of the signal route (line 1), the direction (forward or reverse) of the in-/outgoing signal route direction indicated by *inout* (line 2), and the quotation (*Quot*) value NODELAY because a signal route never has a delay (line 3).

6.4.3 Extraction of Input Signal Sets

extract-inputsigs-prcs(**mk**-*Process-definition*$_1$(, , , *prcdset*, , , , , , , , *grordec*)) $\triangleq$ (6.4.3.1)

1 **union** {*extract-inputsigs-prcd*(*prcd*) | *prcd* $\in$ *prcdset*} $\cup$
2 *extract-inputsigs-grordec*(*grordec*)

type: *Process-definition*$_1$ $\to$ *Signal-identifier*$_1$-**set**

Objective Obtain the input signal set of a process.

Parameters

prcdset,*grordec* The set of procedures and the process graph/service decomposition in the process.

Result The set of signals which the process is able to receive.

Algorithm

Line 1-2 The set of signals which the process can receive is the union of the sets of signals which each contained procedure can receive and the set of signals which can be received by the process graph/services. Note that if the process is decomposed into services no procedures are defined at process level.

extract-inputsigs-grordec(*grordec*) $\triangleq$ (6.4.3.2)

(**is**-*Process-graph*$_1$(*grordec*)
  $\to$ *extract-inputsigs-graph*(*grordec*),
 is-*Service-decomposition*$_1$(*grordec*)
  $\to$ *extract-inputsigs-decomp*(*grordec*))

type: (*Process-graph*$_1$ | *Service-decomposition*$_1$) $\to$ *Signal-identifier*$_1$-**set**

Objective Obtain the set of signals which can be received (directly) by a process graph or service decomposition.

Parameters

grordec The process graph/service decomposition.

Result The set of signals which can be received.

extract-inputsigs-decomp(**mk**-*Service-decomposition*$_1$(*servset*, ,)) $\triangleq$ (6.4.3.3)

union{*extract-inputsigs-serv*(*serv*) | *serv* $\in$ *servset*}

type: *Service-decomposition*$_1$ $\to$ *Signal-identifier*$_1$-**set**

Objective Obtain the set of signals which can be received by a service decomposition.

Parameters

servset The service definitions contained in the decomposition.

Result The set of signals which can be received.

Algorithm

Line 1 The set of signals which the decomposition can receive is the union of the input signals sets for each service.

extract-inputsigs-serv(**mk**-*Service-definition*$_1$(, *prcdset*, , , , , , *graph*)) $\triangleq$ (6.4.3.4)

1 **union** {*extract-inputsigs-prcd*(*prcd*) | *prcd* ∈ *prcdset*} ∪
2 *extract-inputsigs-graph*(*graph*)

type: *Service-definition*$_1$ → *Signal-identifier*$_1$-**set**

Objective Obtain the input signal set of a service.

Parameters

prcdset,*graph* The set of procedures and the service graph in the service.

Result The set of signals which the service is able to receive.

Algorithm

Line 1-2 The set of signals which the service can receive is the union of the sets of signals which each contained procedure can receive and the set of signals which can be received by the service graph.

extract-inputsigs-prcd(**mk**-*Procedure-definition*$_1$(, , *prcdset*, , , , *graph*)) $\triangleq$ (6.4.3.5)

1 **union** {*extract-inputsigs-prcd*(*prcd*) | *prcd* ∈ *prcdset*} ∪
2 *extract-inputsigs-graph*(*graph*)

type: *Procedure-definition*$_1$ → *Signal-identifier*$_1$-**set**

Objective Obtain the set of signals which can be received by a procedure.

Parameters

prcdset,*graph* The set of procedures and the procedure graph in the procedure.

Result The set of signals which can be received by the procedure.

Algorithm

Line 1-2 The set of signals which the procedure can receive is the union of the sets of signals which each contained procedure can receive and the set of signals which can be received by the procedure graph.

extract-inputsigs-graph(*graph*) $\triangleq$ (6.4.3.6)

(**let** (, *statenodes*) = *decomp-graph*(*graph*) **in**
let *savenodes* = {*svnd* | **mk**-*State-node*$_1$(, *svnd*, ,) ∈ *statenodes*},
inputnodes = **union** {*inpnds* | **mk**-*State-node*$_1$(, , *inpnds*,) ∈ *statenodes*} **in**
union{*sigset* | **mk**-*Save-signalset*$_1$(*sigset*) ∈ *savenodes*} ∪
{*sigid* | **mk**-*Input-node*$_1$(*sigid*, ,) ∈ *inputnodes*})

type: (*Process-graph*$_1$ | *Service-graph*$_1$ | *Procedure-graph*$_1$) → *Signal-identifier*$_1$-**set**

Objective Obtain the set of signals which can be received by a process, service or procedure graph.

Parameters

graph The process/service/procedure graph.

Result The set of signals which can be received by the graph.

Algorithm

Line 1 Extract all state nodes from the graph.

Line 2-3 Extract all save nodes and input nodes from the state nodes.

Line 4-5 Extract all input signals from the save nodes and input nodes and return this signal set. Note that if the graph contains more than one state node, all input signals could be obtained from just one of the state nodes. However, the expression in line 4-5 also works when the graph contains no state nodes.

6.4.4 Update of Descriptors with Reachabilities

The following auxiliary domain is used in this section.

Reachability-or-endp = *Reachability-endp* | *Reachability*

This domain covers the possible kinds of members in partial *Reachability* sets which as mentioned earlier can either contain “real” *Reachabilities* or be singleton sets containing a *Reachability-endp*oint.

update-endpd(*outreaches*, *pathelem*, *sigset*, *inreaches*)(*dict*) $\triangleq$ (6.4.4.1)

1 (**let** *totalreaches* = {*total-reach*(*outreach*, *pathelem*, *sigset*, *inreach*) |
2 *outreach* ∈ *outreaches* ∧ *inreach* ∈ *inreaches*} **in**
3 *update-endpd'*(*totalreaches*)(*dict*))

type: *Reachability-or-endp*-**set** *Path-element* *Signal-identifier*$_1$-**set** *Reachability-or-endp*-**set**
→ *Entity-dict* → *Entity-dict*

Objective Construct total *Reachabilities* from an outgoing partial *Reachability* set, a bridging *Path-element*, the set of signals carried by this bridge, and an ingoing partial *Reachability* set, and insert the total *Reachabilities* in the *Entity-dict.*

Parameters

outreaches The outgoing *Reachability* set.
It is either a “real” partial *Reachability* set or a singleton set containing a *Reachability-end*point.

pathelem The bridging *Path-element.*

sigset The signals carried by the bridge.

inreaches The incoming *Reachability* set.
It is either a “real” partial *Reachability* set or a singleton set containing a *Reachability-endp*oint.

Result The *Entity-dict* where the total *Reachabilities* have been inserted.

Algorithm

Line 1-2 Construct a set of (origin end point, total *Reachability*) pairs. The set contains one element for each outgoing and each ingoing partial *Reachability*.

Line 3 Use this set to insert total *Reachabilities* in the *Entity-dict.*

total-reach(*outreach*, *pathelem*, *sigset*, *inreach*) ≙ (6.4.4.2)

```
 1   (is-Reachability-endp(outreach) ∧ is-Reachability-endp(inreach)
 2      → (let orgp = outreach
 3              destp = inreach in
 4           (orgp, (destp, sigset, ⟨pathelem⟩))),
 5    is-Reachability-endp(outreach)
 6      → (let orgp = outreach,
 7              (destp, insigs, inpath) = inreach in
 8           (orgp, (destp, sigset ∩ insigs, ⟨pathelem⟩ ⁀ inpath))),
 9    is-Reachability-endp(inreach)
10      → (let (orgp, outsigs, outpath) = outreach,
11              destp = inreach in
12           (orgp, (destp, outsigs ∩ sigset, outpath ⁀ ⟨pathelem⟩))),
13    T → (let (orgp, outsigs, outpath) = outreach,
14              (destp, insigs, inpath) = inreach in
15           (orgp, (destp, outsigs ∩ sigset ∩ insigs, outpath ⁀ ⟨pathelem⟩ ⁀ inpath))))
```

type: *Reachability-or-endp Path-element Signal-identifier*$_1$-**set** *Reachability-or-endp*
→ *Reachability-endp Reachability*

Objective Construct a total *Reachability* from an outgoing partial *Reachability*, a bridging *Path-element*, the signals carried by the bridge, and an ingoing partial *Reachability*.

Parameters

outreach The outgoing partial *Reachability*.

pathelem The bridging *Path-element.*

sigset The signals carried by the bridge.

inreach The ingoing partial *Reachability*.

Result A pair consisting of an origin end point and a total *Reachability*.

Algorithm

Line 1-4 If both partial *Reachabilities* are end points the origin of the total *Reachability* is the "outgoing" end point, the destination is the "ingoing" end point, the *Path* consists of the bridge, and the signal set is that of the bridge.

Line 5-8 If the outgoing partial *Reachability* is an end point and the ingoing partial *Reachability* is a "real" one, the origin of the total *Reachability* is the "outgoing" end point, the destination is that of the ingoing partial *Reachability*, the *Path* is the bridge appended in front of the ingoing partial *Path*, and the signals carried are those carried by both the bridge and the ingoing partial *Path.*

Line 9-12 This case is analogous to the case covered in line 5-8. Here the outgoing partial *Reachability* is a "real" one and the ingoing partial *Reachability* is an end point.

Line 13-15 If both partial *Reachabilities* are "real" *Reachabilities* the origin is that of the outgoing partial one, the destination is that of the ingoing partial one, the *Path* is the bridge connecting the two partial *Paths*, and the signals carried are those carried by the bridge and both partial *Paths.*

update-endpd'(*totalreaches*)(*dict*) ≜ (6.4.4.3)

```
1   if totalreaches = {} then
2     dict
3    else
4     (let totalreach ∈  totalreaches in
5      let (orgp, reach) = totalreach in
6      let dict' = add-reachability(orgp, reach)(dict) in
7      update-endpd'(totalreaches \ {totalreach})(dict'))
```

type: (*Reachability-endp Reachability*)-**set** → *Entity-dict* → *Entity-dict*

Objective Insert a set of total *Reachabilities* in the *Entity-dict.*

Parameters

totalreaches A set of (origin end point, total *Reachability*) pairs.

Result The *Entity-dict* where the set of total *Reachabilities* has been inserted.

Algorithm

Line 1-2 If the set of total *Reachabilities* is empty the *Entity-dict* is unchanged.

Line 4-5 Select a pair from the set and decompose it.

Line 6 Insert the selected total *Reachability* in the *Entity-dict.*

Line 7 Insert the remaining total *Reachabilities* in the *Entity-dict.*

update-endpd-self(*endp, sigset*)(*dict*) ≜ (6.4.4.4)

```
1   (let reach = (endp, sigset, ⟨⟩) in
2    let dict' = add-reachability(endp, reach)(dict) in
3    let dict'' = insert-input-signals(endp, sigset)(dict') in
4    dict'')
```

type: (*Process-identifier*$_1$ | *Service-identifier*$_1$) *Signal-identifier*$_1$-**set** → *Entity-dict* → *Entity-dict*

Objective Construct a total "self" *Reachability* for a process or service and insert it in the *Entity-dict.*

Parameters

endp The identifier of the process or service.

sigset The complete input signal set of the process or service.

Result The *Entity-dict* where the "self" *Reachability* has been inserted. If appropriate (i.e. if the entity is a service) the input signals are also inserted in the *Entity-dict.*

Algorithm

Line 1 The destination of the "self" *Reachability* is the process/service, the *Path* is empty, and it carries the input signals of the process/service.

Line 2 Insert the *Reachability* in the *Entity-dict.*

Line 3 Insert the input signals in the *Entity-dict* if appropriate.

Line 4 Return the updated *Entity-dict.*

add-reachability(*orgp*, *reach*)(*dict*) ≜ (6.4.4.5)

```
1  (let (, sigset,) = reach in
2   if sigset = {} then
3     dict
4    else
5     (orgp = ENVIRONMENT
6        → (let oldreaches = dict(ENVIRONMENT) in
7            dict + [ENVIRONMENT ↦ oldreaches ∪ {reach}]),
8      (orgp, PROCESS) ∈ dom dict
9        → (let mk-ProcessDD(parmdl, init, maxi, ograph, oldreaches) = dict((orgp, PROCESS)) in
10           dict + [(orgp, PROCESS) ↦ mk-ProcessDD(parmdl, init, maxi, ograph, oldreaches ∪ {reach})]),
11     (orgp, SERVICE) ∈ dom dict
12       → (let mk-ServiceDD(graph, insigs, oldreaches) = dict((orgp, SERVICE)) in
13           dict + [(orgp, SERVICE) ↦ mk-ServiceDD(graph, insigs, oldreaches ∪ {reach})])))
```

type: *Reachability-endp Reachability* → *Entity-dict* → *Entity-dict*

Objective Add a total *Reachability* to the *Entity-dict.*

Parameters

orgp The origin of the *Reachability*.

reach The *Reachability*.

Result The *Entity-dict* where the *Reachability* has been inserted (unless it carries no signals).

Algorithm

Line 1-3 If the *Reachability* is empty then is not inserted in the *Entity-dict.*

Line 5-7 If the origin of the *Reachability* is the system environment, the *Reachability* is included in the ENVIRONMENT entry.

Line 8-10 If the origin of the *Reachability* is a process, the *Reachability* is added to its descriptor in the *Entity-dict.*

Line 11-13 If the origin of the *Reachability* is a service, the *Reachability* is added to its descriptor in the *Entity-dict.*

insert-input-signals(*endp*, *sigset*)(*dict*) ≜ (6.4.4.6)

```
((endp, PROCESS) ∈ dom dict
   → dict,
 (endp, SERVICE) ∈ dom dict
   → (let mk-ServiceDD(graph, , reaches) = dict((endp, SERVICE)) in
       dict + [(endp, SERVICE) ↦ mk-ServiceDD(graph, sigset, reaches)]))
```

type: (*Process-identifier*$_1$ | *Service-identifier*$_1$) *Reachability* → *Entity-dict* → *Entity-dict*

Objective Insert the set of input signals for a process or service in the *Entity-dict.* (Actually the *Entity-dict* is only changed for services but it makes the definition of other functions easier.)

Parameters

endp The identifier of the process or service.

sigset The set of input signals.

Result The *Entity-dict* where the input signals have been inserted.

Algorithm

Line 1-2 If the entity is a process, the *Entity-dict* is not changed because process descriptors do not contain an input signal field.

Line 3-5 If the entity is a service, the input signal field of its descriptor is updated with the input signal set.

6.5 Simple Information Extraction from Channels/Signal Routes

This section defines some simple auxiliary functions for information extraction from channels and signal routes, such as whether a channel/signal route is internal to its enclosing scope unit, which signals are carried in a given direction (in or out) by a non-internal channel/signal route, which block/process/service is connected to a non-internal channel/signal route.

6.5.1 Information from All Channels/Signal Routes

is-internal-chan(**mk**-*Channel-definition*$_1$)(, , *forwpath*,)) $\triangleq$ (6.5.1.1)

1 (**let mk**-*Channel-path*$_1$(*endp*1, *endp*2,) = *forwpath* **in**
2 *endp*1 $\neq$ ENVIRONMENT $\wedge$ *endp*2 $\neq$ ENVIRONMENT)

type: *Channel-definition*$_1$ $\rightarrow$ *Bool*

Objective Test whether a channel is internal to its enclosing scope unit.

Parameters

forwpath The forward channel path in the definition of the channel.

Result **true** if the channel is internal, **false** if the channel leads from or to the boundary of its enclosing scope unit.

Algorithm

Line 1 Get the origin and destination end point of the channel.

Line 2 The channel is internal if none of its end points is the **env** of its enclosing scope unit.

is-internal-sigroute(**mk**-*Signal-route-definition*$_1$)(, *forwpath*,)) $\triangleq$ (6.5.1.2)

1 (**let mk**-*Signal-route-path*$_1$(*endp*1, *endp*2) = *forwpath* **in**
2 *endp*1 $\neq$ ENVIRONMENT $\wedge$ *endp*2 $\neq$ ENVIRONMENT)

type: *Signal-route-definition*$_1$ $\rightarrow$ *Bool*

Objective Test whether a signal route is internal to its enclosing scope unit.

Parameters

forwpath The forward signal route path in the definition of the signal route.

Result **true** if the signal route is internal, **false** if the signal route leads from or to the boundary of its enclosing scope unit.

Algorithm

Line 1 Get the origin and destination end point of the signal route.

Line 2 The signal route is internal if none of its end points is the **env** of its enclosing scope unit.

direction-signals-chan(**mk**-*Channel-definition*$_1$(, , *forwpath*, *orevpath*), *pathdir*) $\triangleq$ (6.5.1.3)

```
1   cases pathdir:
2    (FORWARD
3       → (let mk-Channel-path1(, , forwsigs) = forwpath in
4           forwsigs),
5     REVERSE
6       → if orevpath = nil then
7            {}
8           else
9            (let mk-Channel-path1(, , revsigs) = orevpath in
10             revsigs))
```

type: *Channel-definition*$_1$ *Path-direction* → *Signal-identifier*$_1$**-set**

Objective Extract from a channel the signals carried in a given direction (forward or reverse).

Parameters

forwpath,*orevpath* The forward and optional reverse channel path in the definition of the channel.

pathdir The direction (forward or reverse) of which the signals are wanted.

Result The set of signals carried in the given direction.

Algorithm

line 2-4 If the forward signals are wanted then extract the signals from the forward channel path.

Line 5-10 If the reverse signals are wanted there are two possibilities: Either the channel is unidirectional so no signals are carried in the reverse direction (line 6), or the channel is bidirectional and then the signals are extracted from the reverse channel path (line 9-10).

direction-signals-sigroute(**mk**-*Signal-route-definition*$_1$(, *forwpath*, *orevpath*), *pathdir*) $\triangleq$ (6.5.1.4)

```
cases pathdir:
 (FORWARD
    → (let mk-Signal-route-path1(, , forwsigs) = forwpath in
        forwsigs),
  REVERSE
    → if orevpath = nil then
         {}
        else
         (let mk-Signal-route-path1(, , revsigs) = orevpath in
          revsigs))
```

type: *Signal-route-definition*$_1$ *Path-direction* → *Signal-identifier*$_1$**-set**

Objective Extract from a signal route the signals carried in a given direction (forward or reverse).

Parameters

forwpath,*orevpath* The forward and optional reverse signal route path in the definition of the signal route.

pathdir The direction (forward or reverse) of which the signals are wanted.

Result The set of signals carried in the given direction.

Algorithm

Line 2-4	If the forward signals are wanted then extract the signals from the forward signal route path.
Line 5-10	If the reverse signals are wanted there are two possibilities: Either the signal route is unidirectional so no signals are carried in the reverse direction (line 6), or the signal route is bidirectional and then the signals are extracted from the reverse signal route path (line 9-10).

6.5.2 Information from Non-Internal Channels/Signal Routes

```
inout-going-signals-chan(inout, mk-Channel-definition1(, , forwpath, orevpath)) ≜          (6.5.2.1)

1       (let mk-Channel-path1(endp1, endp2, forwsigs) = forwpath in
2         cases inout:
3          (IN
4               → (endp1 = ENVIRONMENT
5                       → forwsigs,
6                    orevpath = nil
7                       → {},
8                    ⊤ → (let mk-Channel-path1(, , revsigs) = orevpath in
9                          revsigs)),
10          OUT
11              → (endp2 = ENVIRONMENT
12                      → forwsigs,
13                   orevpath = nil
14                      → {},
15                   ⊤ → (let mk-Channel- path1(, , revsigs) = orevpath in
16                         revsigs))))

type:   (IN | OUT) Channel-definition1 → Signal-identifier1-set
```

Objective Extract from a non-internal channel the signals carried in a given direction (in or out).

Parameters

inout Indicates whether the in- or outgoing signals are wanted.

forwpath, orevpath The forward and optional reverse channel path in the definition of the channel.

Result The set of signals carried in the given direction.

Algorithm

Line 1 Get the two channel end points and the forward signals.

Line 3-9 Handle the case where the ingoing signals are wanted. If the origin end point of the channel is the scope unit boundary the ingoing signals are the forward signals of the channel (line 4-5); else if the channel is unidirectional (in the outgoing direction) the ingoing signal set is empty (line 6); else the ingoing signals are the reverse signals of the channel (line 8-9).

Line 10-16 Handle the case where the outgoing signals are wanted. The case is handled analogously to the one in line 3-9.

inout-going-signals-sigroute(*inout*-**mk**-*Signal-route-definition*$_1$ (, *forwpath*, *orevpath*)) ≜ (6.5.2.2)

```
1     (let mk-Signal-route-path1(endp1, endp2, forwsigs) = forwpath in
2       cases inout:
3        (IN
4             → (endp1 = ENVIRONMENT
5                     → forwsigs,
6                  orevpath = nil
7                     → {},
8                  ⊤ → (let mk-Signal-route-path1(, , revsigs) = orevpath in
9                          revsigs)),
10        OUT
11            → (endp2 = ENVIRONMENT
12                    → forwsigs,
13                 orevpath = nil
14                    → {},
15                 ⊤ → (let mk-Signal-route-path1(, , revsigs) = orevpath in
16                         revsigs))))
```

type: (IN | OUT) *Signal-route-definition*$_1$ → *Signal-identifier*$_1$-**set**

Objective Extract from a non-internal signal route the signals carried in a given direction (in or out).

Parameters

inout Indicates whether the in- or outgoing signals are wanted.

forwpath,*orevpath* The forward and optional reverse signal route path in the definition of the signal route.

Result The set of signals carried in the given direction.

Algorithm

Line 1 Get the two signal route end points and the forward signals.

Line 3-9 Handle the case where the ingoing signals are wanted. If the origin end point of the signal route is the scope unit boundary the ingoing signals are the forward signals of the signal route (line 4-5); else if the signal route is unidirectional (in the outgoing direction) the ingoing signal set is empty (line 6); else the ingoing signals are the reverse signals of the signal route (line 8-9).

Line 10-16 Handle the case where the outgoing signals are wanted. The case is handled analogously to the one in line 3-9.

inout-going-path-direction-chan(*inout*, **mk**-*Channel-definition*$_1$(, , *forwpath*,)) ≜ (6.5.2.3)

```
1     (let mk-Channel-path1(endp1, endp2,) = forwpath in
2       cases inout:
3        (IN      → if endp1 = ENVIRONMENT then FORWARD else REVERSE,
4         OUT     → if endp2 = ENVIRONMENT then FORWARD else REVERSE))
```

type: (IN | OUT) *Channe-definition*$_1$ → *Path-direction*

Objective Get for a non-internal channel the direction (forward or reverse) of a given direction (in or out).

Parameters

inout Indicates whether the in- or outgoing direction is wanted.

forwpath The forward channel path in the definition of the channel.

Result A forward/reverse channel direction indication.

Algorithm

Line 1 Get the two end points of the channel.

Line 3 Handle the case where the ingoing direction is wanted. If the channel origin is the scope unit boundary, the ingoing direction is forward, otherwise it is reverse.

Line 4 Handle the case where the outgoing direction is wanted. The case is analogous to that of line 3.

inout-going-path-direction-sigroute(*inout*, **mk**-*Signal-route-definition*$_1$(, *forwpath*,)) ≙ (6.5.2.4)

```
1    (let mk-Signal-route-path1(endp1, endp2,) = forwpath in
2      cases inout:
3       (IN     → if endp1 = ENVIRONMENT then FORWARD else REVERSE,
4        OUT    → if endp2 = ENVIRONMENT then FORWARD else REVERSE))
```

type: (IN | OUT) *Signal-route-definition*$_1$ → *Path-direction*

Objective Get for a non-internal signal route the direction (forward or reverse) of a given direction (in or out).

Parameters

inout Indicates whether the in- or outgoing direction is wanted.

forwpath The forward signal route path in the definition of the signal route.

Result A forward/reverse signal route direction indication.

Algorithm

Line 1 Get the two end points of the signal route.

Line 3 Handle the case where the ingoing direction is wanted. If the signal route origin is the scope unit boundary, the ingoing direction is forward, otherwise it is reverse.

Line 4 Handle the case where the outgoing direction is wanted. The case is analogous to that of line 3.

connected-block(**mk**-*Channel-definition*$_1$(, , *forwpath*,)) ≙ (6.5.2.5)

```
1    (let mk-Channel-path1(endp1, endp2,) = forwpath in
2     if endp2 = ENVIRONMENT then endp1 else endp2)
```

type: *Channel-definition*$_1$ → *Block-definition*$_1$

Objective Get for a non-internal channel the identifier of the block to which it is connected.

Parameters

forwpath The forward channel path in the definition of the channel.

Result The block identifier.

Algorithm

Line 1 Get the two end points of the channel.

Line 2 If the destination end point of the channel is the scope unit boundary, the connected block is the origin end point, else it is the destination end point.

connected-process-or-service(**mk**-*Signal-route-definition*$_1$(, *forwpath*,)) ≜ (6.5.2.6)

1 (**let mk**-*Signal-route-path*$_1$(*endp*1, *endp*2,) = *forwpath* **in**
2 **if** *endp*2 = ENVIRONMENT **then** *endp*1 **else** *endp*2)

type: *Signal-route-definition*$_1$ → (*Process-identifier*$_1$ | *Service-identifier*$_1$)

Objective Get for a non-internal signal route the identifier of the process or service to which it is connected.

Parameters

forwpath The forward signal route path in the definition of the signal route.

Result The block identifier.

Algorithm

Line 1 Get the two end points of the signal route.

Line 2 If the destination end point of the signal route is the scope unit boundary, the connected process/service is the origin end point, else it is the destination end point.

7 General-Purpose Auxiliary Functions

This section defines some simple general-purpose functions for handling of SDL abstract syntax (AS_1) domains.

7.1 Simple Identifier Handling

enclosing-scopeunit(**mk**-*Identifier*$_1$(*qual*,)) $\triangleq$ (7.1.1)

```
1   convert-to-identifier(qual)
```

type: *Identifier*$_1$ → *Identifier*$_1$

Objective Get the identifier of the enclosing scope unit of an entity.

Parameters

qual The qualifier in the identifier of the entity.

Result The identifier of the enclosing scope unit.

Algorithm

Line 1 Convert the qualifier to an identifier denoting the same entity.

enclosing-block(**mk**-*Identifier*$_1$(*qual*,)) $\triangleq$ (7.1.2)

```
1   convert-to-identifier(bloc-scopeunit (qual))
```

type: *Identifier*$_1$ → *Identifier*$_1$

Objective Get the identifier of the enclosing block of an entity.

Parameters

qual The qualifier in the identifier of the entity.

Result The identifier of the enclosing block.

Algorithm

Line 1 Find the qualifier denoting the enclosing block and convert it to an identifier denoting the same entity.

block-scopeunit(*qual*) $\triangleq$ (7.1.3)

```
1   (let pathitem = qual[len qual] in
2    if is-Block-qualifier1(pathitem) then
3       convert-to-identifier(qual)
4      else
5       (let restqual = ⟨qual[i] | 1 ≤ i < len qual⟩ in
6        block-scopeunit(restqual)))
```

type: *Qualifier*$_1$ → *Identifier*$_1$

Objective Get the identifier of the block which encloses (or is) a given entity.

Parameters

qual The qualifier denoting the entity.

Result The identifier of the enclosing block (or the entity itself if it is a block).

Algorithm

Line 1 Get the rightmost path item in the qualifier.

Line 2-3 If the path item denotes a block the whole qualifier is converted to an identifier.

Line 5-6 Remove the rightmost path item from the qualifier and call the function recursively on the rest of the qualifier.

process-or-service-scopeunit(*qual*) $\triangleq$ (7.1.4)

1 (**let** *pathitem* = *qual*[**len** *qual*] **in**
2 **if is**-$\mathit{Process\text{-}qualifier}_1$(*pathitem*) $\vee$ **is**-$\mathit{Service\text{-}qualifier}_1$(*pathitem*) **then**
3 *convert-to-identifier*(*qual*)
4 **else**
5 (**let** *restqual* = $\langle qual[i] \mid 1 \le i <$ **len** $qual\rangle$ **in**
6 *process-or-service-scopeunit*(*restqual*)))

type: $\mathit{Qualifier}_1 \to \mathit{Identifier}_1$

Objective Get the identifier of the process or service which encloses (or is) a given entity.

Parameters

qual The qualifier denoting the entity.

Result The identifier of the enclosing process or service (or the entity itself if it is a process or service).

Algorithm

Line 1 Get the rightmost path item in the qualifier.

Line 2-3 If the path item denotes a process or service the whole qualifier is converted to an identifier.

Line 5-6 Remove the rightmost path item from the qualifier and call the function recursively on the rest of the qualifier.

convert-to-identifier(*qual*) $\triangleq$ (7.1.5)

1 (**let** *qual'* = $\langle qual[i] \mid 1 \le i <$ **len** $qual\rangle$,
2 *nm'* = **cases** *qual*[**len** *qual*]:
3 (**mk**-$\mathit{Block\text{-}qualifier}_1$(*nm*) $\to$ *nm*,
4 **mk**-$\mathit{Block\text{-}substructure\text{-}qualifier}_1$(*nm*) $\to$ *nm*,
5 **mk**-$\mathit{Process\text{-}qualifier}_1$(*nm*) $\to$ *nm*,
6 **mk**-$\mathit{Service\text{-}qualifier}_1$(*nm*) $\to$ *nm*,
7 **mk**-$\mathit{Procedure\text{-}qualifier}_1$(*nm*) $\to$ *nm*,
8 **mk**-$\mathit{Signal\text{-}qualifier}_1$(*nm*) $\to$ *nm*,
9 **mk**-$\mathit{Sort\text{-}qualifier}_1$(*nm*) $\to$ *nm*) **in**
10 **mk**-$\mathit{Identifier}_1$(*qual', nm'*))

type: $\mathit{Qualifier}_1 \to \mathit{Identifier}_1$

Objective Convert a qualifier to an identifier denoting the same scope unit.

Parameters

qual The qualifier.

Result The corresponding identifier.

Algorithm

Line 1 Obtain the qualifier denoting the enclosing scope unit.

Line 2-9 Extract the scope unit name from the rightmost path item.

Line 10 Construct the identifier.

7.2 Selection of Definitions from Definition Sets

select-block(*blid*, *bset*) $\triangleq$ (7.2.1)

1 (**let** *block* ∈ *bset* **be s.t. s**-*Block-name*$_1$(*block*) = **s**-*Name*$_1$(*blid*) **in**
2 *block*)

type: *Block-identifier*$_1$ *Block-definition*$_1$-**set** → *Block-definition*$_1$

Objective Get from a set of block definitions the one denoted by a given identifier.

Parameters

blid The block identifier.

bset The set of block definitions.

Result The block definition.

Algorithm

Line 1 The block definition wanted is that with the same name as the name part of the identifier.

Line 2 Return the block definition.

select-process(*prid*, *pset*) $\triangleq$ (7.2.2)

1 (**let** *prcs* ∈ *pset* **be s.t. s**-*Process-name*$_1$(*prcs*) = **s**-*Name*$_1$(*prid*) **in**
2 *prcs*)

type: *Process-identifier*$_1$ *Process-definition*$_1$-**set** → *Process-definition*$_1$

Objective Get from a set of process definitions the one denoted by a given identifier.

Parameters

prid The process identifier.

pset The set of process definitions.

Result The process definition.

Algorithm

Line 1 The process definition wanted is that with the same name as the name part of the identifier.

Line 2 Return the process definition.

select-channel(*chid*, *cset*) $\triangleq$ (7.2.3)

1 (**let** *chan* ∈ *cset* **be s.t. s**-*Channel-name*$_1$(*chan*) = **s**-*Name*$_1$(*chid*) **in**
2 *chan*)

type: *Channel-identifier*$_1$ *Channel-definition*$_1$-**set** → *Channel-definition*$_1$

Objective Get from a set of channel definitions the one denoted by a given identifier.

Parameters

chid The channel identifier.

cset The set of channel definitions.

Result The channel definition.

Algorithm

Line 1 The channel definition wanted is that with the same name as the name part of the identifier.

Line 2 Return the channel definition.

select-signalroute(*srid, srset*) $\triangleq$ (7.2.4)

1 (**let** $sr \in$ *srset* **be s.t. s**-$Signal\text{-}route\text{-}name_1(sr)$ = **s**-$Name_1(srid)$ **in**
2 *sr*)

type: $Signal\text{-}route\text{-}identifier_1$ $Signal\text{-}route\text{-}definition_1$-**set** $\rightarrow$ $Signal\text{-}route\text{-}definition_1$

Objective Get from a set of signal route definitions the one denoted by a given identifier.

Parameters

srid The signal route identifier.

srset The set of signal route definitions.

Result The signal route definition.

Algorithm

Line 1 The signal route definition wanted is that with the same name as the name part of the identifier.

Line 2 Return the signal route definition.

7.3 Simple Decomposition of Behaviour Graphs

decomp-graph(*graph*) $\triangleq$ (7.3.1)

cases *graph*:
(**mk**-*Process-graph*$_1$(*strt*, *stnds*) → (*strt*, *stnds*),
mk-*Service-graph*$_1$(*strt*, *stnds*) → (*strt*, *stnds*),
mk-*Procedure-graph*$_1$(*strt*, *stnds*) → (*strt*, *stnds*))

type: (*Process-graph*$_1$ | *Service-graph*$_1$ | *Procedure-graph*$_1$) →
(*Process-start-node*$_1$ | *Service-start-node*$_1$ | *Procedure-start-node*$_1$) *State-node*$_1$-**set**

Objective Decompose a process/service/procedure graph into its start node and state node set.

Parameters

graph The behaviour graph.

Result A pair consisting of the start node and the state node set.

decomp-start-node(*start*) $\triangleq$ (7.3.2)

cases *start*:
(**mk**-*Process-start-node*$_1$(*trans*) → *trans*,
mk-*Service-start-node*$_1$(*trans*) → *trans*,
mk-*Procedure-start-node*$_1$(*trans*) → *trans*)

type: (*Process-start-node*$_1$ | *Service-start-node*$_1$ | *Procedure-start-node*$_1$) → *Transition*$_1$

Objective Extract from a process/service/procedure start node its contained transition.

Parameters

start The start node.

Result The contained transition.

Domain Index

Function Index

Processor Index

Variable Index

Error Messages

§**2.7.5:**	Answers in decision actions are not mutually exclusive 96
§**2.7.5:**	No matching answer 76
§**3.2.1:**	Block or subblock is not in consistent subset 124
§**3.2.1:**	Leaf block contains no processes 124
§**3.3:**	Illegal refinement of channel 128, 129
§**5.2.1:**	Generation or reduction of equivalence classes of the enclosing scope unit 101
§**5.3.1.7:**	Expression, term or value is equivalent to the error term 103
§**5.3.1.7:**	Literal is equivalent to the error term 101
§**5.3.1.9:**	Value is not within the range of the syntype 69, 73, 74, 75, 80, 82
§**5.4.2.1:**	Attempt to evaluate error expression 78
§**5.4.2.2:**	The viewed value is undefined 81
§**5.4.2.2:**	Value of accessed variable is undefined 81
§**5.4.3:**	Ground expression in assignment statement is out of range 96
§**5.4.4.4:**	No revealed variable access can be made 37
§**5.4.4.6:**	Attempt to evaluate an anyvalue expression for an empty sort or syntype 83