INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

BLUE BOOK

VOLUME X – FASCICLE X.5

ANNEX F.3 TO RECOMMENDATION Z.100: SDL FORMAL DEFINITION

DYNAMIC SEMANTICS

IXTH PLENARY ASSEMBLY
MELBOURNE, 14-25 NOVEMBER 1988

Geneva 1989

INTERNATIONAL TELECOMMUNICATION UNION

CCITT

THE INTERNATIONAL
TELEGRAPH AND TELEPHONE
CONSULTATIVE COMMITTEE

BLUE BOOK

VOLUME X – FASCICLE X.5

ANNEX F.3 TO RECOMMENDATION Z.100: SDL FORMAL DEFINITION

DYNAMIC SEMANTICS

IXTH PLENARY ASSEMBLY
MELBOURNE, 14-25 NOVEMBER 1988

Geneva 1989

ISBN 92-61-03791-7

CONTENTS OF THE CCITT BOOK
APPLICABLE AFTER THE NINTH PLENARY ASSEMBLY (1988)

BLUE BOOK

FASCICLE III.6 – Line transmission of non-telephone signals. Transmission of sound-programme and television signals. Series H and J Recommendations (Study Group XV).

FASCICLE III.7 – Integrated Services Digital Network (ISDN) – General structure and service capabilities. Recommendations I.110-I.257 (Study Group XVIII).

FASCICLE III.8 – Integrated Services Digital Network (ISDN) – Overall network aspects and functions, ISDN user-network interfaces. Recommendations I.310-I.470 (Study Group XVIII).

FASCICLE III.9 – Integrated Services Digital Network (ISDN) – Internetwork interfaces and maintenance principles. Recommendations I.500-I.605 (Study Group XVIII).

Volume IV

FASCICLE IV.1 – General maintenance principles: maintenance of international transmission systems and telephone circuits. Recommendations M.10-M.782 (Study Group IV).

FASCICLE IV.2 – Maintenance of international telegraph, phototelegraph and leased circuits. Maintenance of the international public telephone network. Maintenance of maritime satellite and data transmission systems. Recommendations M.800-M.1375 (Study Group IV).

FASCICLE IV.3 – Maintenance of international sound-programme and television transmission circuits. Series N Recommendations (Study Group IV).

FASCICLE IV.4 – Specifications for measuring equipment. Series O Recommendations (Study Group IV).

Volume V – Telephone transmission quality. Series P Recommendations (Study Group XII).

Volume VI

FASCICLE VI.1 – General Recommendations on telephone switching and signalling. Functions and information flows for services in the ISDN. Supplements. Recommendations Q.1-Q.118 *bis* (Study Group XI).

FASCICLE VI.2 – Specifications of Signalling Systems Nos. 4 and 5. Recommendations Q.120-Q.180 (Study Group XI).

FASCICLE VI.3 – Specifications of Signalling System No. 6. Recommendations Q.251-Q.300 (Study Group XI).

FASCICLE VI.4 – Specifications of Signalling Systems R1 and R2. Recommendations Q.310-Q.490 (Study Group XI).

FASCICLE VI.5 – Digital local, transit, combined and international exchanges in integrated digital networks and mixed analogue-digital networks. Supplements. Recommendations Q.500-Q.554 (Study Group XI).

FASCICLE VI.6 – Interworking of signalling systems. Recommendations Q.601-Q.699 (Study Group XI).

FASCICLE VI.7 – Specifications of Signalling System No. 7. Recommendations Q.700-Q.716 (Study Group XI).

FASCICLE VI.8 – Specifications of Signalling System No. 7. Recommendations Q.721-Q.766 (Study Group XI).

FASCICLE VI.9 – Specifications of Signalling System No. 7. Recommendations Q.771-Q.795 (Study Group XI).

FASCICLE VI.10 – Digital subscriber signalling system No. 1 (DSS 1), data link layer. Recommendations Q.920-Q.921 (Study Group XI).

FASCICLE VI.11 – Digital subscriber signalling system No. 1 (DSS 1), network layer, user-network management. Recommendations Q.930-Q.940 (Study Group XI).

FASCICLE VI.12 – Public land mobile network. Interworking with ISDN and PSTN. Recommendations Q.1000-Q.1032 (Study Group XI).

FASCICLE VI.13 – Public land mobile network. Mobile application part and interfaces. Recommendations Q.1051-Q.1063 (Study Group XI).

FASCICLE VI.14 – Interworking with satellite mobile systems. Recommendations Q.1100-Q.1152 (Study Group XI).

Volume VII

FASCICLE VII.1 – Telegraph transmission. Series R Recommendations. Telegraph services terminal equipment. Series S Recommendations (Study Group IX).

FASCICLE VII.2 – Telegraph switching. Series U Recommendations (Study Group IX).

FASCICLE VII.3 – Terminal equipment and protocols for telematic services. Recommendations T.0-T.63 (Study Group VIII).

FASCICLE VII.4 – Conformance testing procedures for the Teletex Recommendations. Recommendation T.64 (Study Group VIII).

FASCICLE VII.5 – Terminal equipment and protocols for telematic services. Recommendations T.65-T.101, T.150-T.390 (Study Group VIII).

FASCICLE VII.6 – Terminal equipment and protocols for telematic services. Recommendations T.400-T.418 (Study Group VIII).

FASCICLE VII.7 – Terminal equipment and protocols for telematic services. Recommendations T.431-T.564 (Study Group VIII).

Volume VIII

FASCICLE VIII.1 – Data communication over the telephone network. Series V Recommendations (Study Group XVII).

FASCICLE VIII.2 – Data communication networks: services and facilities, interfaces. Recommendations X.1-X.32 (Study Group VII).

FASCICLE VIII.3 – Data communication networks: transmission, signalling and switching, network aspects, maintenance and administrative arrangements. Recommendations X.40-X.181 (Study Group VII).

FASCICLE VIII.4 – Data communication networks: Open Systems Interconnection (OSI) – Model and notation, service definition. Recommendations X.200-X.219 (Study Group VII).

FASCICLE VIII.5 – Data communication networks: Open Systems Interconnection (OSI) – Protocol specifications, conformance testing. Recommendations X.220-X.290 (Study Group VII).

FASCICLE VIII.6 – Data communication networks: interworking between networks, mobile data transmission systems, internetwork management. Recommendations X.300-X.370 (Study Group VII).

FASCICLE VIII.7 – Data communication networks: message handling systems. Recommendations X.400-X.420 (Study Group VII).

FASCICLE VIII.8 – Data communication networks: directory. Recommendations X.500-X.521 (Study Group VII).

Volume IX – Protection against interference. Series K Recommendations (Study Group V). Construction, installation and protection of cable and other elements of outside plant. Series L Recommendations (Study Group VI).

Volume X

CONTENTS OF FASCICLE X.5 OF THE BLUE BOOK

Annex F.3 to Recommendation Z.100

REMARK

Due to the specialized nature of the SDL semantics, this Fascicle is published in English only.

REMARQUE

Etant donné la nature très spéciale de la sémantique du LDS, ce fascicule est publié uniquement en anglais.

OBSERVACIÓN

Debido a la naturaleza especializada de la semántica del LED, este fascículo sólo se publica en inglés.

PRELIMINARY NOTES

1 The Questions entrusted to each Study Group for the Study Period 1989-1992 can be found in Contribution No. 1 to that Study Group.

2 In this Fascicle, the expression "Administration" is used for shortness to indicate both a telecommunication Administration and a recognized private operating agency.

PAGE INTENTIONALLY LEFT BLANK

PAGE LAISSEE EN BLANC INTENTIONNELLEMENT

Contents

FASCICLE X.5

Annex F.3 to Recommendation Z.100

SDL FORMAL DEFINITION DYNAMIC SEMANTICS

Introduction

This part of The Formal Definition defines the dynamic properties of SDL. For a description of the over-all 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 as a number of concurrent processes. The communication between these is synchronous, CSP-like communication. The lines in the picture indicate communication by means of CSP-output. The *system*-process creates instances of the other processes: one instance of the *view*- and *timer*-process, one instance of the *path*-process for each distinct path an SDL-output may be transported by, and one instance of the pair *sdl-process*, *input-port* for each actual SDL-process instance. Totally, six different meta process types are used in the model:

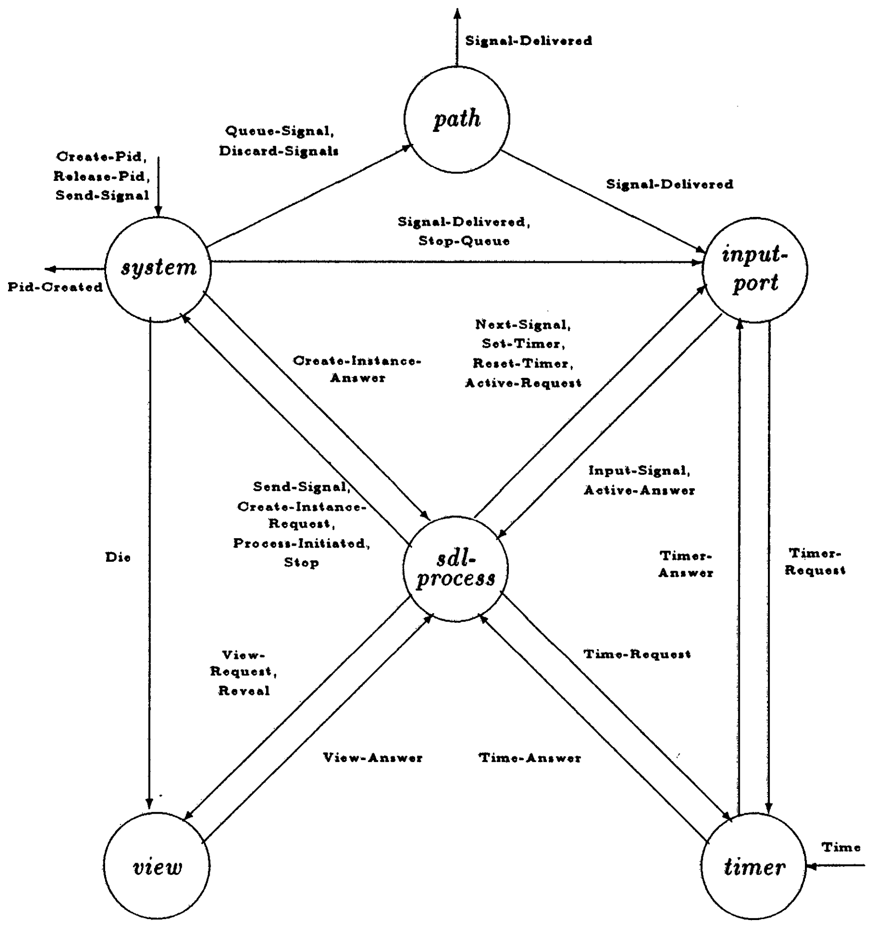

Figure 1: Structure of Interpretation Model

The processes are:

1. *system*

   Which handles the signal routing and the creation of *sdl-process*es.

2. *path*

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

3. *timer*

   Which keeps track of the current time and handles time-out. 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.

4. *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*.

5. *sdl-process*

   Which interprets the behaviour of an SDL-process.

6. *input-port*

   Which handles the queueing of signals in an SDL-process. For each instance of *sdl-process* there exists exactly one *input-port*. Signals are always received by an *sdl-process* in its *input-port*.

1 Domains for the Process communication

1.1 *sdl-process* $\leftrightarrow$ *system*

1	*Process-Initiated*	::	*Port*
2	*Port*	=	Π(*input-port*)

When an *sdl-process* has been created it answers *Process-Initiated*, when it is ready to interpret its process graph. The data carried is the CSP-instance of the input-port started by the process instance.

3	*Create-Instance-Request*	::	*Process-identifier*$_1$ *Value-List*
4	*Create-Instance-Answer*	::	[*Offspring-Value*]
5	*Offspring-Value*	=	*Pid-Value*
6	*Pid-Value*	=	*Value*
7	*Value*	=	*Ground-term*$_1$

When a process interprets the create request node, it will output the *Create-Instance-Request* to *system*. The data carried are the process identifier of the process to be started, and the list of actual parameters. *system* will respond by outputting *Create-Instance-Answer* back, carrying the Pid-Value of the started process. If no process could be started, **nil** is returned.

8	*Send-Signal*	::	*Signal-identifier*$_1$ *Value-List* [*Pid-Value*] *Direct-via*$_1$

When an instance of *sdl-process* interprets an output node, it will output *Send-Signal.* This transfers the identifier of the SDL signal being sent, the list of optional values attached to the signal, the optional destination process instance, and the optional via set of channel identifiers or signal route identifiers.

9	*Stop*	::	()

When an instance of *sdl-process* interprets the stop node, it will send *Stop* to *system* which keeps track of whether instances are alive or dead.

1.2 *sdl-process* $\leftrightarrow$ *input-port*

1	*Next-Signal*	::	*Signal-identifier*$_1$-**set**
2	*Input-Signal*	::	*Signal-identifier*$_1$ *Value-List Sender-Value*

The *sdl-process* outputs a *Next-Signal* to its *input-port* and the *input-port* responds (when non-empty queue) with the output of *Input-Signal.* The *Signal-identifier*$_1$-**set** denotes the signals which should remain in the queue (Save-set).

3	*Set-Timer*	::	*Timer-identifier*$_1$ *Timeout-value Arglist Equivalent-test*
4	*Reset-Timer*	::	*Timer-identifier*$_1$ *Arglist Equivalent-test*
5	*Timeout-value*	=	*Value*
6	*Arglist*	=	*Value**
7	*Equivalent-test*	::	*Ground-term*$_1$ *Ground-term*$_1$ $\rightarrow$ *Bool*

When a *input-port* inputs *Set-Timer* from the *sdl-process*, interpreting a set timer action, it starts a timer with expiration time denoted by *Timeout-value.* A timer has also a value list attached which, together with the *Timer-identifier*$_1$, identifies the timer instance. The *input-port* tests whether two requests (i.e. *Set-Timer, Reset-Timer or Active-Request*) refers to the same timer instance by comparing their *Timer-identifiers* and applying the *Equivalent-test* function on the elements in the two associated *Arglists*. The *Equivalent-test* function takes two *Ground-term*$_1$s as arguments and returns (indicated by "$\rightarrow$") a *Bool.*

8	*Active-Request*	::	*Timer-identifier*$_1$ *Arglist Equivalent-test*
9	*Active-Answer*	::	*Bool*

The *sdl-process* sends an *Active-Request* to the *input-port*, to determine whether or not, the timer, identified by *Timer-identifier*$_1$, is active. *Arglist* and *Equivalent-test* are explained above.

1.3 *sdl-process* $\leftrightarrow$ *view*

1	*Reveal*	:: *Variable-identifier*$_1$ (*Value* | UNDEFINED) *Pid-Value*

When an *sdl-process* updates a revealed variable, it will output *Reveal*. *Reveal* carries the identifier of the revealed variable, the new value of the variable, and the *Pid-Value* of "self".

2	*View-Request*	:: *Variable-identifier*$_1$ *Pid-Value*
3	*View-Answer*	:: (*Value* | UNDEFINED)

When an *sdl-process* views a variable it will output *View-Request*. *View-Request* carries the identifier of the variable to be viewed, and the *Pid-Value* of the instance, which reveals it. *view* responds by outputting *View-Answer*, which carries the requested value.

1.4 *sdl-process, input-port* $\leftrightarrow$ *timer*

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

When an *sdl-process* evaluates the NOW expression, it will send *Time-Request*. *timer* responds by sending *Time-Answer*, which carries the value of the current time.

The *input-port* continuously test on the expiration time of its timers. For that purpose it needs the actual time from the *timer*. This communication is the same as between *sdl-process* and *timer*.

1.5 *system* $\leftrightarrow$ *environment*

1	*Create-Pid*	:: *Port*
2	*Pid-Created*	:: *Pid-Value*
3	*Release-Pid*	:: *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. When a process instance is created in the environment, its CSP-name *input-port* is sent to system carried by *Create-Pid*. The system responds by outputting the associated SDL *Pid-value* back to environment carried by *Pid-Created*. When a process instance in the environment ceases to exist, the system will receive *Release-Pid* with the SDL *Pid-Value* of the stopped process from environment. The main purpose of the scheme is to justify the administration within the system of Pid-Values in the environment.

1.6 *system* $\leftrightarrow$ *view*

1	*Die*	:: *Pid-Value*

When an SDL process has stopped, *view* inputs *Die* such that the instance entry can be deleted from its internal map of possibly revealed variables.

1.7 *system* $\leftrightarrow$ *path*

1	*Queue-Signal*	::	*Signal-identifier*$_1$ *Value-List Pid-Value Port*

A Signal is transferred by the *system* by outputting a *Queue-Signal* to the instance of *path* corresponding to the selected route from sender to receiver. *Queue-Signal* transfers the identifier of the signal, the values carried by the signal, the Pid-Value denoting the sender, and the CSP-instance value of the receiving *input-port.*

2	*Discard-Signals*	::	*Port*

When an sdl-process stops, the *system* demands all *paths* to remove signals directed towards the *input-port* of the stopping *sdl-process.* This is done by outputting *Discard-Signals.*

1.8 *system, path* $\leftrightarrow$ *input-port*

1	*Signal-Delivered*	::	*Signal-identifier*$_1$ *Value-List Sender-Value*
2	*Value-List*	=	[*Value*]*
3	*Sender-Value*	=	*Pid-Value*

The *path* sends the signal to *input-port* when it has been released. The *system* sends the signal directly to *input-port,* if the sender and receiver is within same block.

1.9 *system* $\leftrightarrow$ *input-port*

1	*Stop-Queue*	::	()

When an *sdl-process* instance stops, *system* outputs *Stop-Queue* to make its *input-port* stop.

1.10 *timer* $\leftrightarrow$ *tick*

1	*Time*	::	()

The tick-process is not formally modelled. It is a process which sends "ticks" with regular intervals to the system. Thus it forms the basis of the timer-process. The ticks should be regarded as part of the input stream, which the SDL-system transforms into an output-stream.

2 Domains for the Entity Information

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

1 *Entity-dict* = (*Identifier*$_1$ SIGNAL) $\overrightarrow{m}$ *SignalDD* ∪
(*Identifier*$_1$ PROCEDURE) $\overrightarrow{m}$ *ProcedureDD* ∪
(*Identifier*$_1$ TYPE) $\overrightarrow{m}$ *TypeDD* ∪
(*Identifier*$_1$ SORT) $\overrightarrow{m}$ (*SyntypeDD* | *SortDD*) ∪
(*Identifier*$_1$ PROCESS) $\overrightarrow{m}$ *ProcessDD* ∪
(*Identifier*$_1$ VALUE) $\overrightarrow{m}$ (*VarDD* | *OperatorDD*) ∪
ENVIRONMENT $\overrightarrow{m}$ *Reachabilities* ∪
EXPIREDF $\overrightarrow{m}$ *Is-expired* ∪
PIDSORT $\overrightarrow{m}$ *Sort-identifier*$_1$ ∪
NULLVALUE $\overrightarrow{m}$ *Literal-operator-identifier*$_1$ ∪
TRUEVALUE $\overrightarrow{m}$ *Literal-operator-identifier*$_1$ ∪
FALSEVALUE $\overrightarrow{m}$ *Literal-operator-identifier*$_1$ ∪
SCOPEUNIT $\overrightarrow{m}$ *Qualifier*$_1$ ∪
PORT $\overrightarrow{m}$ Π(*input-port*) ∪
SELF $\overrightarrow{m}$ Π(*input-port*) ∪
PARENT $\overrightarrow{m}$ Π(*input-port*)

Entity-dict consist of a map from pairs of Identifier$_1$s (*Identifier*$_1$s) and their associated entity class into descriptors. An entity class is either SIGNAL, PROCEDURE, TYPE, SORT, PROCESS or VALUE.

In addition, it contains information of how signals from/to the environment of the system can be routed. *ENVIRONMENT* is explained below.

A descriptor is either a descriptor of a signal, a procedure, a type, a syntype, a process, a sort, a variable, a literal or operator. 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. Those objects are accessed via some *Quot* values:

ENVIRONMENT — When applied on *Entity-dict* the result is the *Reachabilities* leading to/originating from the environment.

EXPIREDF — When applied on *Entity-dict* the result is a function used by the *timer* processor.

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 the AS_1 identifier of the PiD literal null.

TRUEVALUE — When applied on *Entity-dict* the result is the AS_1 identifier of the boolean literal true.

FALSEVALUE — When applied on *Entity-dict* the result is the AS_1 identifier of the boolean literal false.

SCOPEUNIT — When applied on *Entity-dict* the result is the qualifier denoting the current scopeunit.

PORT — When applied on *Entity-dict* the result is the Π value of input port of an sdl process.

SELF — When applied on *Entity-dict* the result is the Π value of the sdl process using the *Entity-dict*.

PARENT When applied on *Entity-dict* the result is the Π value of the parent of the sdl process using the *Entity-dict*.

2.1 The Signal Descriptor

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

SignalDD is a descriptor of a signal. It contains the list of sort or syntype identifiers attached to the signal.

2.2 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*$_1$.

2.3 The Type Descriptor

1	*TypeDD*	:: *Sortmap Equations*$_1$
2	*Sortmap*	= *Sort-identifier*$_1$ $\overrightarrow{m}$ *Term-class*-**set**
3	*Term-class*	= (*Ground-term*$_1$ | *Error-term*$_1$)-**set**

TypeDD is a descriptor of a data type definition. It contains a map (*Sortmap*) of all *Sort-identifier*$_1$s visible in the scopeunit enclosing the data type definition into the set of equivalent classes existing for the sort. An equivalent class (*Term-class*) is a set of ground terms possible joined with the error term.

It also contains the equations (*Equations*$_1$) from which the equivalent classes are derived.

2.4 The Sort Descriptor

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

SortDD and *SyntypeDD* are descriptors of newtypes and syntypes respectively. A newtype descriptor contains the identifier of the enclosing data type definition as all the properties of newtypes are hold in that descriptor.

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

2.5 The Process Descriptor

1	*ProcessDD*	::	*ParameterDD** *Initial Maximum Process-graph*$_1$ *Reachabilities*
2	*Reachabilities*	=	*Reachability*-**set**
3	*ParameterDD*	=	*Variable-identifier*$_1$
4	*Initial*	=	*Intg*
5	*Maximum*	=	*Intg*
6	*Reachability*	=	(*Process-identifier*$_1$ | ENVIRONMENT) *Signal-identifier*$_1$-**set** *Path*
7	*Path*	=	*Path-identifier**
8	*Path-identifier*	=	*Identifier*$_1$

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 *Reachability* defines a *Process-identifier*$_1$ 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 signalroute and channel identifiers (*Path-identifiers*). *Path* is empty in the cases where *Process-identifier*$_1$ is both the sender and the receiver. A formal parameter descriptor is the *Variable-identifier*$_1$ of the parameter.

2.6 The Variable Descriptor

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

VarDD is a descriptor of a variable. It contains the variable identifier, the sort or syntype identifier, the REVEALED attribute and optionally a reference to a storage. There is no descriptor for view variables because the *View-definition*$_1$s contains no (important) information. Each time a procedure is invoked, *Entity-Dict* is overwritten with the descriptors representing the formal parameters and local declarations. For IN/OUT parameters the descriptors contain the associated actual parameters and a reference to the version of the storage where the version of the *Variable-identifier*$_1$ is found, i.e. because SDL allows recursive procedures, there may exist several versions of the same *Variable-identifier*$_1$, one for each recursive call and therefore also several versions of the storage.

2.7 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 The Underlying System

3.1 System Processor

This processor is the entry point of interpretation for an SDL-description. 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.

system **processor** (*as*$_1$*tree*, *subset*, *auxinf*) $\triangleq$ (3.1.1)

```
1  (let (timeinf, terminf, expiredf, delayf) = auxinf in
2   dcl instancemap := [] type Π(sdl-process) m→
3       ((ENVIRONMENT | Process-identifier₁) Pid-Value);
4   dcl queuemap := [] type Pid-Value-set m→Port;
5   dcl pidno := [] type Process-identifier₁ m→N₀;
6   dcl pathmap := [] type Channel-identifier₁* m→Π(path);
7   dcl pidset := {} type Pid-Value-set;
8   trap exit with error in
9   (let entitydict = extract-dict(as₁tree, subset, expiredf, terminf) in
10   start view();
11   start timer(timeinf);
12   start-initial-processes(entitydict);
13   pathd(delayf)(entitydict);
14   handle-inputs(entitydict)))
```

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

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 1):
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 *timer* processor.
terminf	A closure containing the AS$_1$ identifier of the Pid sort, the Pid null literal and the Boolean literals 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.

Algorithm

Line 2 Let instancemap denote a map from csp-processor instances to a composite domain of *Process-identifier*$_1$ or ENVIRONMENT and *Pid-Value*. This map is instantiated as empty. It is primarily used for routing of signals and for creation of new instances.

Line 4 Let queuemap denote a map from equivalence classes of *Pid-Values* to the *input-port* of the *sdl-processes*. This map is instantiated as empty. Queuemap is used for same purposes as instancemap.

Line 5 Let pidno denote a map for checking that the maximum number of instances of a process-definition is not exceeded. The map is instantiated as empty.

Line 6	Let pathmap denote a map from delaying paths to csp-instances of the *path*-processor. A delaying path is a list of channels traversed by a signal instance, when an output-node is interpreted. It is necessary to distinguish possible delaying paths, since sequence is only guaranteed, when following the same sequence of channels.
Line 7	Let pidset denote the initially empty set of *Pid-Values*.
Line 11	Start *timer* with actual parameters for the handling of NOW (further explained in *timer*).
Line 12	Start *sdl-processes*.
Line 13	Start one processor instance for each communication path in the system.
Line 14	Handle all further communication.

start-initial-processes(*entitydict*) $\triangleq$ **(3.1.2)**

```
1 (let pset = {id | (id, PROCESS) ∈ dom entitydict} in
2  for all p ∈ pset do
3  (let mk-ProcessDD(, no, , , ) = entitydict((p, PROCESS)) in
4   (for i = 1 to no do
5    handle-create-instance-request(p, nil, nil)(entitydict))))
```

type : *Entity-dict* $\Rightarrow$

Objective Start sdl-processes.

Algorithm

Line 3	Let *no* denote the number of instances to be started of a process.
Line 4	Start the requested number of instances.

pathd(*delayf*)(*entitydict*) $\triangleq$ **(3.1.3)**

```
(let rs = entitydict(ENVIRONMENT) in
 for all reach ∈ rs do
 (let (, , p) = reach in
  let p' = ⟨p[i] | 1 ≤ i ≤ len p − 1⟩ in
  (if p' ∉ dom pathmap
     then (def cspp : start path(delayf);
              pathmap := c pathmap + [p' ↦ cspp])
     else I));
 for all (pd, PROCESS) ∈ dom entitydict do
 (let mk-ProcessDD(, , , , rs) = entitydict((pd, PROCESS)) in
  for all reach ∈ rs do
  (let (d, , p) = reach in
   let p' = ⟨p[i] | 2 ≤ i ≤ (d = ENVIRONMENT
                                  → len p,
                              T → len p − 1)⟩ in
   if len p' > 0 ∧ p ∉ dom pathmap then
     (def cspp : start path(delayf);
      pathmap := c pathmap + [p' ↦ cspp])
    else
     I)))
```

type : $(() \Rightarrow$ *Bool*$) \rightarrow$ *Entity-dict* $\Rightarrow$

Objective Start a *path* processor instance for each pair of *Process-identifier*$_1$ and path in the system. Updates a map from paths to csp-instances.

Parameters

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

Algorithm

Line 1	Let *rs* denote the *reachability-set* of (processes in) the environment. This information is based on channels leading into the system from the environment, and extracted from *entitydict*.
Line 2 - 8	Start a processor instance for each reachability in the set. *pcsp* denotes the csp-instance of the started processor.
Line 4	Let *p'* denote the path causing delay (i.e. excluding the last item, which is a signal route).
Line 7	Update the pathmap accordingly.
Line 9-18	Repeat this scheme for originating processes within the system.
Line 13	Define the delaying part of the path as starting from second element, since first element is a signal route, and ending with second last element if within the system, since then the last element is a signal route.
Line 16	Only start a processor, if the remaining path is non-empty (causes delay).

handle-inputs(*entitydict*) $\triangleq$ (3.1.4)

```
1  (cycle {input mk-Send-Signal(si, vl, r, p) from se
2            ⇒  handle-send-signal(si, vl, r, p, se)(entitydict),
3         input mk-Create-Instance-Request(prid, vl) from se
4            ⇒  handle-create-instance-request(prid, vl, se)(entitydict),
5         input mk-Stop() from se
6            ⇒  handle-stop(se),
7         input mk-Create-Pid(port) from se
8            ⇒  handle-create-from-environment(port, se)(entitydict),
9         input mk-Release-Pid(p) from se
10           ⇒  handle-stops-in-environment(p, se)})
```

type : *Entity-dict* $\Rightarrow$

Objective Handle all communication of *system* after initializations.

Algorithm

Line 1 — Start a loop forever. In each pass 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 handle-function.

handle-stops-in-environment(*p*, *se*) $\triangleq$ (3.1.5)

```
(def class s.t. p ∈ class ∧ class ∈ dom c queuemap;
 def q : c queuemap(class);
 instancemap := c instancemap \ {se};
 queuemap := c queuemap \ {class};
 discard-signals-to-port(q))
```

type : *Pid-Value* $\Pi \Rightarrow$

Objective Handle stop of "processes" in the environment by updating maps within the system.

Parameters

p Pid-Value of the "process" to stop.

se Csp-instance of the "SENDER".

Algorithm

Line 3-4 Remove the "process" and its "input port" from the maps of living instances.

Line 5 Handle the removal of signals to the stopping process in the environment waiting on communication paths.

$$handle\text{-}create\text{-}from\text{-}environment(port, se)(entitydict) \triangleq \tag{3.1.6}$$

```
1  (def (pid, pidclass) : getpid(entitydict);
2   instancemap := c instancemap + [se ↦ (ENVIRONMENT, pid)];
3   queuemap := c queuemap + [pidclass ↦ port];
4   output mk-Pid-Created(pid) to se)
```

type : $\Pi\ \Pi \rightarrow Entity\text{-}dict \Rightarrow$

Objective Handle the creation of 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

port Csp-instance of the input-port of "the sender". The environment is assumed to contain an input-port, since this is the way asynchronous communication is implemented.

se The csp-instance of "the sender".

Algorithm

Line 2-3 Update the maps of living instances with the "process" communicated by the environment.

Line 4 Return the *Pid-Value* to the environment.

handle-send-signal(*si*, *vl*, *r*, *p*, *se*)(*entitydict*) $\triangleq$ **(3.1.7)**

```
1   (def (sid, sp) : c instancemap(se);
2    (let re = if is-Identifier₁(sid)
3               then s-Reachabilities(entitydict((sid, PROCESS)))
4               else entitydict(ENVIRONMENT) in
5     let re' = {(, s', ) ∈ re | si ∈ s'} in
6     let re'' = if p = {} then re' else {(, ,p') ∈ re' | p ∩ elems p' ≠ {}} in
7     def rp : (r ≠ nil
8                 → {(rident, r) ∈ rng c instancemap | (rident, , ) ∈ re''},
9               T → {(rident, ) ∈ rng c instancemap | (rident, , ) ∈ re''});
10    (card (rp) = 0
11       → exit("§2.7.4: No receiver found"),
12     card (rp) > 1
13       → exit("§2.7.4: Multiple receivers found"),
14     T → (let {(rident, ri)} = rp in
15          let (rident', , path) ∈ re'' be s.t. rident' = rident in
16          (def class s.t. ri ∈ class ∧ class ∈ dom queuemap;
17           def rcsp : c queuemap(class);
18           (let reduced-path = delaying-path(path, sid, rident) in
19            if reduced-path = ⟨⟩
20              then output mk-Signal-Delivered(si, vl, sp) to rcsp
21              else (def path' : c pathmap(reduced-path);
22                       output mk-Queue-Signal(si, vl, sp, rcsp) to path')))))))
```

type: *Signal-identifier*$_1$ *Value-List* [*Pid-Value*] *Direct-via*$_1$
Π(*sdl-process*) $\rightarrow$ *Entity-dict* $\Rightarrow$

Objective Routing of signals.

Parameters

si	Signal being sent.
vl	Optional list of values carried by the signal.
r	Optional *Pid-Value* denoting the receiver, from the TO-clause.
p	Optional set of paths, from the VIA-clause.
se	Csp-instance of the sending *sdl-process*.

Algorithm

Line 1	Let *sid* and *sp* denote the *Process-identifier*$_1$ and *Pid-Value* of the sender.
Line 2	Test whether the signal is sent from the environment (line 4) or from a process within the system (line 3). In both cases *re* denotes the *Reachability*-set of the sender. The remaining function consecutively restricts the reachability of the sender (until line 9).
Line 5	Restrict to those reachabilities which may convey the actual signal, *si*.
Line 6-6	Restrict based on the paths given in the VIA-clause, *p*.
Line 6	No restriction if the VIA-clause was absent.
Line 8	Check paths from the environment.
Line 6	Restrict the reachability-set to those members which mentions a member of *p* from the VIA-clause in their path.
Line 7	Let *rp* denote the set of potential receivers. The members of *rp* are pairs (*Process-identifier*$_1$, *Pid-Value*).
Line 8	Handle the case, where a TO-clause was given. The pair must then denote a living instance where the *rident* is in a reachability.
Line 9	Handle the case without a TO-clause. In this case the *Pid-Value* member of the pair is left unspecified.

Line 10-14	Test the number of receivers found.
Line 11	Define the error of no (reachable and living) receiver.
Line 13	Define the error of more than one receiver (indeterminism of the OUTPUT-node).
Line 14	Indicate success: *rp* contains one and only one member.
Line 15	Choose a *path* leading to the unique receiver. This choice is non-deterministic, if sub-channels leading towards the same process may carry the same signal.
Line 16	Let *rcsp* denote the csp-instance of the input-port of the receiving *sdl-process*.
Line 18	Let *reduced-path* denote the part of the *Path*, which causes delay (the channels).
Line 19	If the signal passes on no channels (within same block), then the signal is output to the *input-port* processor of the receiver.
Line 21	Let *path'* denote the csp-instance of the corresponding *path* processor.
Line 22	Output the signal to the selected *path* processor.

delaying-path(*path*, *sid*, *rid*) $\triangleq$ **(3.1.8)**

1 $(\mathbf{len}\, path \leq 1$
2 $\quad \rightarrow \langle\rangle,$
3 $sid = \mathsf{ENVIRONMENT}$
4 $\quad \rightarrow \langle path[i] \mid 1 \leq i \leq \mathbf{len}\, path - 1\rangle,$
5 $rid = \mathsf{ENVIRONMENT}$
6 $\quad \rightarrow \mathbf{tl}\, path,$
7 $\mathsf{T} \rightarrow \langle path[i] \mid 2 \leq i \leq \mathbf{len}\, path - 1\rangle)$

type : $Path\ (\mathsf{ENVIRONMENT} \mid Process\text{-}identifier_1)$
$(\mathsf{ENVIRONMENT} \mid Process\text{-}identifier_1) \rightarrow Path$

Objective Reduce the communication path to the *delaying* path.

Parameters

path	A complete path from sender to receiver
sid	Identity of sender
rid	Identity of receiver

Result The delaying path.

Algorithm

Line 1	If the path is empty or consist of a single signal route identifier, return it unmodified.
Line 3	If the signal originates from the environment then remove the signal route ending the *Path*
Line 5	If the destination is the environment then remove the signal route identifier starting the *Path.*
Line 7	If the signal is sent from one block to another block, then remove the starting and the ending signal route identifier

handle-create-instance-request(*prid*, *vl*, *se*)(*entitydict*) $\triangleq$ **(3.1.9)**

```
 1  (if prid ∉ dom c pidno
 2     then pidno := c pidno + [prid ↦ 0]
 3     else I;
 4   (let mk-ProcessDD(il, , maximum, , ) = entitydict((prid, PROCESS)) in
 5    let vl′ = if vl = nil then ⟨nil | 1 ≤ i ≤ len il⟩ else vl in
 6    def parent : if se = nil then nil else s-Pid-Value(c instancemap(se));
 7    def exceed : (maximum = c pidno(prid));
 8    def (newpid, pidclass) : getpid(entitydict);
 9    if ¬exceed then
10      (def csppid : start sdl-process(parent, newpid, vl′, prid)(entitydict);
11       (input mk-Process-Initiated(qcsppid) from csppid
12          ⇒  (instancemap := c instancemap + [csppid ↦ (prid, newpid)];
13               queuemap := c queuemap + [pidclass ↦ qcsppid];
14               pidno := c pidno + [prid ↦ c pidno(prid) + 1])))
15     else
16      I;
17    if se ≠ nil
18      then output mk-Create-Instance-Answer(if exceed then nil else newpid) to se
19      else I))
```

type : $Process\text{-}identifier_1$ [*Value-List*] [*Π*(*sdl-process*)] → *Entity-dict* ⇒

Objective Handle creation of *sdl-processes*.

Parameters

prid	$Process\text{-}identifier_1$ of the process to be started.
vl	Optional list of actual parameters (= nil if called during system initialization).
se	Optional parent (= nil if called during system initialization).

Algorithm

Line 1	Initiate the map for a *prid* with 0 instances to be 0.
Line 6	Let *parent* denote the parent value to be carried to the new instance.
Line 8	Create a unique *Pid-Value*.
Line 7	Perform the test of exceeding maximum number of instances of a process definition.
Line 10	Start the *sdl-process*-instance itself.
Line 11	Wait for initialization acknowledge from the started *sdl-process*
Line 12	Add the *sdl-process* to the map of *sdl-processes*.
Line 13	Add its *input-port* to the map of *input-ports*.
Line 14	Update the current number of instances for the process definition.
Line 18	Send answer back to "SENDER" with *Pid-Value* of the new process if the create is caused by a $Create\text{-}node_1$. If the maximum number was exceeded, nil is returned.

getpid(*entitydict*) $\triangleq$ (3.1.10)

```
1  (let pidsortid = entitydict(PIDSORT) in
2   let mk-SortDD(tid) = entitydict((pidsortid, SORT)) in
3   let mk-TypeDD(sortmap, ) = entitydict((tid, TYPE)) in
4   let classes = sortmap(pidsortid) in
5   let nullterm = entitydict(NULLVALUE) in
6   def class s.t. class ∈ classes ∧ (nullterm ∉ class) ∧ ¬(∃term ∈ class)(term ∈ c pidset);
7   let pid ∈ class in
8   pidset := c pidset ∪ class;
9   return (pid, class))
```

type : *Entity-dict* $\Rightarrow$ *Pid-Value Pid-Value*-**set**

Objective Extract a *Pid-Value* not used yet. The Unique! operator defined for the Pid sort in Z.100 ensures that there exist 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 terms (values) is found in *entitydict*.

Note, that the model assumes, that the *system*-processor also maintains unique *Pid-Values* for the environment. Otherwise it is hard to imagine how *Pid-Values* may be used to address processes in the environment.

Result An unused *Pid-Value* and the equivalence class, it belongs to.

Algorithm

Line 1 Extract the $Identifier_1$ of the PID sort from *entitydict*.

Line 2 Extract the type identifier defined on the system level.

Line 3 Extract the sortmap containing the equivalent classes of the sort defined on the system level.

Line 4 Extract the equivalence classes of the Pid sort. Note that for the Pid sort, every equivalence class contains exactly one ground term.

Line 5 Extract the AS_1 representation of the NULL term.

Line 6 Take an equivalence class not represented in pidset and different from the NULL term.

Line 7 Take a ground term in this equivalence class.

Line 8 Add this new value to the set of *Pid-Values*.

handle-stop(*se*) $\triangleq$ (3.1.11)

```
1  (def (prid, p) : c instancemap(se);
2   def class s.t. p ∈ class ∧ class ∈ dom queuemap;
3   def q : c queuemap(class);
4   instancemap := c instancemap \ {se};
5   queuemap := c queuemap \ {class};
6   pidno := c pidno + [prid ↦ c pidno(prid) − 1];
7   discard-signals-to-port(q);
8   output mk-Stop-Queue() to q;
9   output mk-Die(p) to view)
```

type : Π(*sdl-process*) $\Rightarrow$

Objective Handle $STOP\text{-}Node_1$s.

Parameters

se Csp-instance of *sdl-process* to be stopped.

Algorithm

Line 4	Subtract the process from the map of living instances.
Line 5	Subtract the corresponding *input-port* from its map.
Line 6	Update the current number of instances of *prid* by subtracting the stopped process.
Line 7	Discard signals waiting on paths to the stopped process.
Line 8	Request the *input-port* to stop.
Line 9	Request the *view* to update map of revealed variables.

discard-signals-to-port$(q) \triangleq$ **(3.1.12)**

```
(for all r ∈ rng c pathmap do
 output mk-Discard-Signals(q) to r)
```

type : *Port* ⇒

Objective Output to all *path*-instances telling them to delete signal instances waiting to be transmitted to one *input-port*.

Parameters

q	Csp-instance of the *input-port*.

3.2 View Processor

view **processor** () $\triangleq$ **(3.2.1)**

```
1  (dcl viewmap := [] type (Pid-Value Variable-identifier₁) ⇸ (Value | UNDEFINED);
2   trap exit with error in
3   (cycle {input mk-Reveal(id, value, pid) from sdl-process
4              ⇒  viewmap := c viewmap + [(pid, id) ↦ value],
5          input mk-View-Request(id, revealpid) from viewpid
6              ⇒  (let entry = (revealpid, id) in
7                   if entry ∈ dom c viewmap
8                     then output mk-View-Answer(c viewmap(entry)) to viewpid
9                     else exit("§5.5.4.4: Revealing process is not alive")),
10         input mk-Die(pid) from system
11             ⇒  (for all (pid, id) ∈ dom c viewmap do
12                  viewmap := c viewmap \ {(pid, id)})}))
```

type : () ⇒

Objective Interpret the concept of VIEW and REVEAL.

Algorithm

Line 1	Let viewmap denote a map from a pair of *Pid-Value* and *Variable-identifier*$_1$ to a revealed *Value*.
Line 3	Handle the *Reveal* input.
Line 4	Update the map with the new entry.
Line 5	Handle a VIEW from *sdl-process*.
Line 8	Return the value to *sdl-process*.
Line 9	Define the error of a variable not being revealed.
Line 10	Handle the notice of a stopped *sdl-process*.
Line 11	Subtract all revealed variables of the stopped process from the map.

3.3 Path Processor

path **processor** (*delayf*) $\triangleq$ **(3.3.1)**

```
 1  (dcl pqueue := ⟨⟩ type (Signal-identifier₁ Value-List Pid-Value Port)*;
 2   cycle {input mk-Queue-Signal(si, vl, sp, rcsp) from system
 3          ⇒  (pqueue := c pqueue ⌢ ⟨(si, vl, sp, rcsp)⟩),
 4         input mk-Discard-Signals(q) from system
 5          ⇒  (pqueue := ⟨c pqueue[i] | 1 ≤ i ≤ len c pqueue ∧
 6                                        (def (, , , r) : c pqueue[i];
 7                                         return r ≠ q)⟩),
 8         (delayf() ∧ c pqueue ≠ ⟨⟩)
 9         (output mk-Signal-Delivered(s-Signal-identifier₁(hd c pqueue),
10                                     s-Value-List(hd c pqueue),
11                                     s-Pid-Value(hd c pqueue)) to s-Port(hd c pqueue))
12          ⇒ (pqueue := tl c pqueue)})
```

type : $(() \Rightarrow \mathit{Bool}) \Rightarrow$

Objective Interpret the potential delay in a path. An instance exists for each value of path in *Reachability*-set in *ProcessDD*.

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 Handle the removal of signals directed to a specific *input-port*, *q*. Is used when the *sdl-process* of the *input-port* stops.

Line 5 Let the new pqueue equal the old one except for items directed to *q*.

Line 8 This clause models the non-deterministic delay on the path. The **output** is guarded by a predicate: it may only take place if *delayf* yields true and pqueue is non-empty. The concrete syntax is:

```
(<predicate>)(<communication event>)
   =>  <statement>
```

The indeterminism is expressed in terms of the imperative function *delayf*, the definition of which is outside the scope of this formal definition. If the predicate holds, the signal is output to the instance of the *input-port*.

Line 12 Remove the output signal from the queue.

3.4 Input-Port Processor

This processor implements the unbounded buffers of *sdl-processes* and timers. The unbounded buffer is reflected in the processor as the variable queue.

input-port **processor** (*ppid*, *expiredf*) $\triangleq$ (3.4.1)

```
1  (dcl queue := ⟨⟩ type (Signal-identifier1 Value-List Pid-Value)*;
2   dcl waiting := false type Bool;
3   dcl pendingset := {} type Signal-identifier1-set;
4   dcl timers := [] type (Timer-identifier1 Arglist) →m (Π(sdl-process) Value Equivalent-test);
5   cycle {input mk-Signal-Delivered(sid, vl, se) from p
6              ⇒  handle-queue-insert(sid, vl, se, p),
7          input mk-Next-Signal(saveset) from p
8              ⇒  handle-queue-extract(saveset, ⟨⟩, c queue, p),
9          input mk-Stop-Queue() from p
10             ⇒  stop,
11         input mk-Set-Timer(tid, v, al, et) from p
12             ⇒  handle-set-timer(tid, v, al, et, p),
13         input mk-Reset-Timer(tid, al, et) from p
14             ⇒  handle-reset-timer(tid, al, et),
15         input mk-Active-Request(tid, al, et) from p
16             ⇒  handle-active-request(tid, al, et, p),
17         (output mk-Time-Request() to timer;
18          handle-time-request(ppid, expiredf))})
```

type : *Pid-Value Is-expired* ⇒

Objective Interpret the input-port of *sdl-process*. An instance exists for each instance of *sdl-process*.

Parameters

pcsp The PId-value of the served *sdl-process*.

expiredf Function delivering True if a given timer has expired.

Algorithm

Line 1 Let queue denote the unbounded buffer of the *sdl-process*. Each entry contains a *Signal-identifier*$_1$, a possibly empty list of *Values* and a *Pid-Value* denoting "SENDER".

Line 2 Let waiting denote whether the *sdl-process* is waiting 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 were members of saveset parameter. In case of a pending request, pendingset (see below) holds the saveset of the pending request.

Line 3 Let pendingset denote the saveset associated with a pending request from *sdl-process* as indicated by waiting.

Line 4 Let timers denote a map for active timers. The Π*(sdl-process)* of the map denotes the *sdl-process* which sets the timer. The *Value* of the map denotes the expiration time, and is nil after expiration (that is: when the time is on the queue). The *Equivalence-test* is used to compare values of *Arglists* (e.g. to compare an element from the timers map with a signal in the queue in the function *handle-queue-extract*). *Value* holds the expiration time. A timer is subtracted from the map, when it is returned to *sdl-process* as a signal.

Line 5 Is the entry of the main cycle of *input-port*.

Line 7 — Note: this input cannot always be answered immediately. The reason for introducing the variables waiting and pendingset 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 empty queue.

Line 17 — Include one output in this scheme. It is the repeated request for the actual time from the *timer*.

handle-queue-insert(*sid*, *vl*, *se*, *pcsp*) $\triangleq$ (3.4.2)

```
1  (queue := c queue ⌢ ⟨(sid, vl, se)⟩;
2   if ¬waiting then
3     handle-queue-extract(c pendingset, ⟨⟩, c queue, pcsp)
4     else
5     I)
```

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

Objective — Insert a signal in the queue.

Parameters

sid — Signal to be inserted.

vl — Its optional list of values.

se — Sender.

pcsp — The CSP-instance of the served *sdl-process*.

Algorithm

Line 1 — Concatenate the signal to queue.

Line 2-3 — Test if a *Next-Signal* is pending, and if so, extract an element from the queue. This may lead to an *Input-Signal* to *sdl-process*.

handle-queue-extract(*saveset*, *qf*, *qa*, *pcsp*) $\triangleq$ **(3.4.3)**

```
 1  (if qa ≠ ⟨⟩ then
 2    (let s = hd qa in
 3     let (sid, vl, se) = s in
 4     (if sid ∈ saveset
 5        then handle-queue-extract(saveset, qf ⌢ ⟨s⟩, tl qa, pcsp)
 6        else (output mk-Input-Signal(sid, vl, se) to pcsp;
 7              queue := qf ⌢ tl qa;
 8              waiting := false;
 9              if (sid, ) ∈ dom c timers then
10                 if (∃a, et)((sid, a) ∈ dom c timers ∧
11                             (, , et) = c timers(sid, a) ∧
12                             same-argument-values(a, vl, et)) then
13                   (def (a, et) s.t. (sid, a) ∈ dom c timers ∧
14                        (, , et) = c timers(sid, a) ∧ same-argument-values(a, vl, et);
15                    timers := c timers \ {(sid, a)})
16                  else
17                   I
18                else
19                 I)))
20     else
21    (pendingset := saveset;
22     waiting := true))
```

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

Objective Extract one element from the queue and send it to *sdl-process* if *sdl-process* is ready to receive input.

Parameters

saveset	Set of signals not to be extracted from queue in this situation.
qf	Part of the queue already examined.
qa	Part of the queue which yet needs examination.
pcsp	The CSP-instance of the served *sdl-process*.

Algorithm

Line 1	Stop the extraction without success if *qa* is empty.
Line 2	Otherwise take the first element of *qa* and
Line 3	decompose the queue into *Signal-identifier*$_1$, a list of *Values*, and "SENDER".
Line 5	If the signal is in *saveset*, then concatenate it to the queue which has been examined, and repeat the search on the remaining part of *qa*.
Line 6	Output *Next-Signal* to *sdl-process* if *si* is not in *saveset*. Update the queue by concatenation of *qf* and remaining part of *qa*. Set the flag for no pending requests, and finally in
Line 9	update the timers map, if the signal extracted was a timer.
Line 13	The timer to be removed should have same identifier as the signal, *si* and a comparison by *et* should conclude, that the argument list from the timer is equivalent to the one from the timers map.
Line 21	In case of no success, set the mark for pending request with the actual *saveset*.

$same\text{-}argument\text{-}values(a, vl, et) \triangleq$ **(3.4.4)**

1 $(\mathbf{len}\ a = \mathbf{len}\ vl\ \wedge$
2 $(\forall i \in \mathbf{ind}\ a)(et(a[i], vl[i])))$

type : *Arglist Arglist Equivalent-test* $\rightarrow$ *Bool*

Objective Test whether two lists of $Term_1$s are equivalent, as defined by *et*.

Parameters

a	One list to check.
vl	The other one.
et	Equivalent-test function.

Algorithm

Line 1	The length of the two lists should be the same.
Line 2	For each index the test should success.

$handle\text{-}set\text{-}timer(tid, v, al, et, p) \triangleq$ **(3.4.5)**

1 $(handle\text{-}reset\text{-}timer(tid, al, et);$
2 $\mathrm{timers} := \mathbf{c}\ \mathrm{timers} + [(tid, al) \mapsto (p, v, et)])$

type : $Timer\text{-}identifier_1$ *Value Arglist Equivalent-test* $\Pi(sdl\text{-}process) \Rightarrow$

Objective Set a timer, by updating the timers map.

Parameters

tid	Identifier of the timer.
v	Expiration time.
al	Argument list of the timer.
et	Corresponding equivalence-test function, which may be applied to each member in the argument list.
p	The *sdl-process* which set the timer.

Algorithm

Line 1	Reset a possibly existing timer with same identifier and argument-list.
Line 2	Update the timers map.

$handle\text{-}reset\text{-}timer(tid, al, et) \triangleq$ **(3.4.6)**

$(\mathbf{for\ all}\ (t, a) \in \mathbf{dom}\,\mathbf{c}\ \mathrm{timers}\ \mathbf{do}$
$\quad(\mathbf{if}\ (\exists a)((tid, a) \in \mathbf{dom}\,\mathbf{c}\ \mathrm{timers}\ \wedge$
$\qquad\qquad same\text{-}argument\text{-}values(a, al, et))\ \mathbf{then}$
$\qquad(\mathbf{def}\ (, e,) : \mathbf{c}\ \mathrm{timers}(tid, al);$
$\qquad\ (\mathrm{timers} := \mathbf{c}\ \mathrm{timers} \setminus \{(t, a)\};$
$\qquad\quad \mathbf{if}\ \mathbf{c}\ e = \mathbf{nil}$
$\qquad\qquad \mathbf{then}\ (handle\text{-}remove\text{-}timer\text{-}from\text{-}queue(tid, al, et, \langle\rangle, \mathbf{c}\ \mathrm{queue}))$
$\qquad\qquad \mathbf{else}\ \mathbf{I}))$
$\qquad \mathbf{else}$
$\qquad\ \mathbf{I}))$

type : $Timer\text{-}identifier_1$ *Arglist Equivalent-test* $\Rightarrow$

Objective Reset a timer by updating the timers map and the queue.

Parameters

tid	Identifier of the timer.
al	Argument list of the timer.
et	Corresponding equivalence-test function, which may be applied to each member in the argument list.

Algorithm

Line 2	Select the appropriate timer as having an argument list such that (*tid*, *a*) is in the domain for the timers map, and such that *a* matches *al* by applying the equivalence-test, *et*.
Line 5	Subtract (*tid*, *a*) from the timers map.
Line 7	Remove *tid* from the queue, if it has been put there (marked in the *Value* field of the range for timers map).

$\mathit{handle\text{-}remove\text{-}timer\text{-}from\text{-}queue}(sid, al, et, qf, qa) \triangleq$ (3.4.7)

1 $(\mathbf{let}\ (si, vl,) = \mathbf{hd}\ qa\ \mathbf{in}$
2 $\quad \mathbf{if}\ si = sid \wedge \mathit{same\text{-}argument\text{-}values}(vl, al, et)$
3 $\qquad \mathbf{then}\ \text{queue} := qf \frown \mathbf{tl}\ qa$
4 $\qquad \mathbf{else}\ \mathit{handle\text{-}remove\text{-}timer\text{-}from\text{-}queue}(sid, al, et, qf \frown \langle \mathbf{hd}\ qa \rangle, \mathbf{tl}\ qa))$

type : $\mathit{Signal\text{-}identifier}_1\ \mathit{Arglist}\ \mathit{Equivalent\text{-}test}$
$(\mathit{Signal\text{-}identifier}_1\ [\mathit{Value}^*]\ \mathit{Pid\text{-}Value})^*$
$(\mathit{Signal\text{-}identifier}_1\ [\mathit{Value}^*]\ \mathit{Pid\text{-}Value})^* \Rightarrow$

Objective Remove one element from the queue.

Parameters

sid	Signal to be removed.
al	Argument list of the timer.
et	Corresponding equivalence-test function , which may be applied to each member in the argument list.
qf	Part of the queue examined.
qa	Part of the queue to be examined yet.

Algorithm

Line 1	Let *si* denote the $\mathit{Signal\text{-}identifier}_1$ of the first element of *qa*.
Line 3	If *si* is the signal to be removed then update queue to be *qf* concatenated with remaining part of *qa*, otherwise in
Line 4	continue the search on the remaining part of *qa*, note that *sid* should always be present in the queue, on the outermost call of the function, so a test terminating the recursion is not needed!

$\mathit{handle\text{-}active\text{-}request}(tid, al, et, pcsp) \triangleq$ (3.4.8)

$(\mathbf{def}\ \mathit{stat} : (\exists a)((tid, a) \in \mathbf{dom}\,\mathbf{c}\ \text{timers} \wedge$
$\qquad\qquad \mathit{same\text{-}argument\text{-}values}(al, a, et));$
$\quad \mathbf{output}\ \mathbf{mk}\text{-}\mathit{Active\text{-}Answer}(stat)\ \mathbf{to}\ pcsp)$

type : $\mathit{Timer\text{-}identifier}_1\ \mathit{Arglist}\ \mathit{Equivalent\text{-}test}\ \Pi(\mathit{sdl\text{-}process}) \Rightarrow$

Objective Supply the answer to ACTIVE based on the timers map.

Parameters

tid	Identifier of the timer.
al	Argument list of the timer.
et	Corresponding equivalence-test function , which may be applied to each member of the argument list.
pcsp	The CSP-instance of the *sdl-process* being served.

Algorithm

Line 1	Let *stat* denote true if the specified timer is in the domain of the timers map, otherwise false.
Line 3	Use this value as parameter in the output to *sdl-process*.

handle-time-request(*ppid*, *expiredf*) $\triangleq$ (3.4.9)

```
1  (input mk-Time-Answer(t) from timer
2      ⇒  (for all (tid, al) ∈ dom c timers do
3           (def (p, expt, et) : c timers(tid, al);
4            if expt ≠ nil ∧ expiredf(expt, t) then
5               (timers := c timers + [(tid, al) ↦ (p, nil, et)];
6                handle-queue-insert(tid, al, ppid, p))
7             else
8               I)))
```

type : *Value Is-expired* ⇒

Objective Handle the comparison with the actual time for all timers being set.

Parameters

ppid	The PId-value of the *sdl-process* being served.
expiredf	Function (constructed in Annex F.2) delivering True if a given timer has expired.

Algorithm

Line 1	Receive the actual time from *timer* in *t*.
Line 2	Start the examination for timers set.
Line 4	Examine whether it is already on the queue, and whether it is expired.
Line 5	In that case change the timers map to contain “on the queue”.
Line 6	Insert the timer on the queue, with “SENDER” equal to “SELF” for the served *sdl-process*.

3.5 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*) $\triangleq$ (3.5.1)

```
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            ⇒  output mk-Time-Answer(c time-now) to p})
```

type : *Time-information* $\Rightarrow$

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 Return NOW.

3.6 Informal Tick Processor

tick **processor** () $\triangleq$ (3.6.1)

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

type : () $\Rightarrow$

4 The SDL-Process

This section describes how the META-IV processor *sdl-process* interpret an instance of an SDL-process. The definition of the SDL-process is from the entity-dict. All inter-process and other communication is managed by the underlying system.

Each SDL-process instance have a local storage which type is given by:

1 $Stg \quad = \quad Identifier_1 \ \overrightarrow{m} (\ Value \mid \mathsf{UNDEFINED})$

4.1 The sdl-process

The META-IV processor *sdl-process* is created by the processor *system* and by its actual parameters given knowledge of its surroundings, itself and the SDL-process it must interpret. An *sdl-process* instance cease to exist when the SDL-process has been interpreted.

sdl-process **processor** (*parentp*, *selfp*, *actparml*, *process-id*)(*dict*) $\triangleq$ **(4.1.1)**

```
 1  (let mk-Identifier₁(qual, nm) = process-id in
 2   let nullterm = dict(NULLVALUE) in
 3   def dict₁ : dict + [SCOPEUNIT ↦ qual ⌢ ⟨mk-Process-qualifier₁(nm)⟩] +
 4       [PORT ↦ start input-port(selfp, dict(EXPIREDF))] +
 5       [SELF ↦ selfp] +
 6       [PARENT ↦ (parentp = nil
 7                    → nullterm,
 8                   T → parentp)];
 9   dcl sender := nullterm type Pid-Value;
10   dcl offspring := nullterm type Pid-Value;
11   dcl stg := [] type Stg;
12   (trap exit() with error in
13    (trap exit(STOP) with output mk-Stop() to system in
14     (let mk-ProcessDD(formparml, , , graph, ) = dict₁((process-id, PROCESS)) in
15      (def dict₂ : dict₁ + [(id, VALUE) ↦ mk-VarDD(id, sort, rev, stg) |
16                    (id, VALUE) ∈ dom dict₁ ∧ is-VarDD(dict₁((id, VALUE))) ∧
17                    s-Qualifier₁(id) = dict₁(SCOPEUNIT) ∧
18                    mk-VarDD(, sort, rev, ) = dict₁((id, VALUE))];
19       init-process-decls(dict₂);
20       init-process-parms(formparml, actparml)(dict₂);
21       output mk-Process-Initiated(dict₂(PORT)) to system;
22       int-process-graph(graph)(dict₂))))))
```

type : $[Pid\text{-}Value]\ Pid\text{-}Value\ Value\text{-}List\ Process\text{-}identifier_1 \rightarrow Entity\text{-}dict \Rightarrow$

Objective Interprets an sdl-process.

Parameters

parentp	The SDL PID-value of the process that created this one.
selfp	The SDL PID-value of this process.
actparml	The list of actual parameters.
process-id	The SDL-identifier of this process.

Algorithm

Line 2	Extract the AS_1 version of the PID value NULL.

Line 3 - 6	Augment the dict so that:
Line 3	SCOPEUNIT denotes the current scope. SCOPEUNIT is updated whenever the interpretation of the SDL-process enters a new scope unit,
Line 4	PORT denotes the CSP-name of the input port, The actual parameters for the creation of the *input-port* processor is the SDL PID-value of this process and a function to test whether or not a timer is expired. The input port is the process that handles the signals send to the process and manipulation of timers.
Line 5	SELF denotes the SDL PID-value of the process itself,
Line 6	PARENT denotes the SDL PID-value of the parent process, if any. There is no parent process if the process is created during system initialization.
Line 9 - 10	Declare the variables sender and offspring, both initialized to *null-term.*
Line 11	Declare a variable, stg, which are to be the local storage of this sdl-process and initiate it to be empty.
Line 12	Trap any exit with error.
Line 13	Traps **exit**(STOP) by sending a stop signal to the system and terminate.
Line 15 - 18	*dict* is changed so that it for each local variable contain a reference to the local storage. Both declared variable and formal parameters are considered as local variables.
Line 19	Augment the storage according to the declarations of the sdl-process.
Line 20	Augment the storage according to the contents of the actual parameters.
Line 21	The process is now initiated and the *system* processor is given knowledge about this and the CSP name of the *input-port* processor by outputting the signal *Process-Initiated* to *system.*
Line 22	Interpret the sdl-process.

init-process-decls(*dict*) $\triangleq$ (4.1.2)

```
1  (for all (id, VALUE) ∈ dom dict do
2   if is-VarDD(dict((id, VALUE))) ∧ s-Qualifier₁(id) = dict(SCOPEUNIT)
3      then update-stg(id, nil)(dict)
4      else I)
```

type : *Entity-dict* $\Rightarrow$

Objective Update the storage with the variable declarations associated to the sdl-process being interpreted.

Algorithm

Line 1	For all those identifiers with VALUE attribute and
Line 2	which are variable descriptors and declared in this process,
Line 3	the storage is initiated with **nil**. The optional initiation belonging to a variable declaration is in AS_1 transformed into assignments in a task prefixing the start transition.

init-process-parms(formparml, actparml)(dict) $\triangleq$ **(4.1.3)**

1 (**for all** $i \in$ **ind** *formparml* **do**

2 *update-stg(formparml*$[i]$*, actparml*$[i]$*)(dict)*)

type : *ParameterDD** *Value-List* $\rightarrow$ *Entity-dict* $\Rightarrow$

Objective Updates the local process storage with formal process parameters and the optionally applied actual parameters.

Parameters

formparml	The list of formal parameters.
actparml	The list of actual parameters.

Algorithm

Line 1 - 2	The local storage is updated with the variable denoted by each formal parameter and the value of its associated actual parameter. The range check is postponed to *update-stg.*

4.2 Interpretation of a process-graph

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

int-process-graph(graph)(dict) $\triangleq$ **(4.2.1)**

1 (**let mk-***Process-graph*$_1$(**mk-***Process-start-node*$_1$(*trans*), *stateset*) = *graph* **in**

2 (**tixe** [*statenm* $\mapsto$ *int-state-node(statenode)(dict)* |

3 *statenode* $\in$ *stateset* $\wedge$ **s-***State-name*$_1$(*statenode*) = *statenm*] **in**

4 (**let mk-***Transition*$_1$(*nodel, termordec*) = *trans* **in**

5 *int-transition(nodel, termordec)(dict)*))))

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

Objective Interprets the SDL process graph

Parameters

graph	The process graph.

Algorithm

Line 1	Partition of the graph into a start transition and a set of states.
Line 2	Traps all **exit**(state-name) 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$_1$) is encountered within the scope of the tixe-construct, that is either in the range of the tixe map (int-state-node) or in int-transition, the interpretation of the process is continued interpreting the state-node with the name statenm$_1$.
Line 4	Interpretation of the start-transition.

int-state-node(**mk-***State-node*$_1$(, **mk-***Save-signalset*$_1$(*saveset*), *inputset*))(*dict*) $\triangleq$ (4.2.2)

```
1  (output mk-Next-Signal(saveset) to dict(PORT);
2   input mk-Input-Signal(sid, actparml, sender') from dict(PORT)
3     ⇒ (sender := sender';
4         (let {inpnode} = {inp ∈ inputset | s-Signal-identifier1(inp) = sid} in
5          let mk-Input-node1(, formparml, trans) = inpnode in
6          for all i ∈ ind formparml do
7          (if formparml[i] ≠ nil
8             then update-stg(formparml, actparml[i])(dict)
9             else I);
10         (let mk-Transition1(nodel, termordec) = trans in
11          int-transition(nodel, termordec)(dict)))))
```

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

Objective Request a new signal from the input-port, receives it and interprets the corresponding transition.

Parameters

state-node Composed of a *saveset* which is the signal to be saved by the input-port and an *inputset* which is a set of signals and associated 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.

Line 2 Receive a signal composed of a signal-identifier, an actual parameter list and the SDL Pid-value of the sender.

Line 3 The process variable sender is updated with the sender value of the just received signal.

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

Line 5 Decompose the selected input into the formal parameter list of the signal and the associated transition.

Line 6 - 9 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 11 Interpret the selected transition.

int-transition(*nodel*, *termordec*)(*dict*) $\triangleq$ (4.2.3)

```
(if nodel = ⟨⟩
   then cases termordec:
          (mk-Nextstate-node1(nm)
              → exit(nm),
           mk-Stop-node1()
              → exit(STOP),
           mk-Return-node1()
              → exit(RETURN),
           mk-Decision-node1(, , )
              → int-decision-node(termordec)(dict))
   else (int-graph-node(hd nodel)(dict);
         int-transition(tl nodel, termordec)(dict)))
```

type : *Graph-node*$_1$* (*Terminator*$_1$ | *Decision-node*$_1$) $\rightarrow$ *Entity-dict* $\Rightarrow$

Objective Interprets a transition.

Parameters

nodel	The list of graph nodes not yet interpreted.
termordec	A terminator node or a decision node.

Algorithm

Line 2	If the node list is empty then *termordec* is interpreted.
Line 3	A nextstate node is interpreted by exit with the name of the next state.
Line 5	A stop node by exit with STOP.
Line 7	A return node by exit with RETURN.
Line 9	A decision node by calling the *int-decision-node* function.
Line 10	If the node list is not empty then the first node is interpreted by the function *int-graph-node.*
Line 11	The remaining part of the transition is interpreted by recursion.

$$int\text{-}decision\text{-}node(\mathbf{mk}\text{-}Decision\text{-}node_1(quest, answset, elseansw))(dict) \triangleq \qquad (4.2.4)$$

```
 1  (let answset' = matching-answer(quest, answset)(dict) in
 2   (if answset' ≠ {}
 3      then (let {mk-Decision-answer1(, trans)} = answset' in
 4              let mk-Transition1(nodel, termordec) = trans in
 5              int-transition(nodel, termordec)(dict))
 6      else (elseansw ≠ nil
 7                → (let mk-Else-answer1(trans) = elseansw in
 8                   let mk-Transition1(nodel, termordec) = trans in
 9                   int-transition(nodel, termordec)(dict)),
10            T → exit("§2.7.5: No matching answer"))))
```

type : $Decision\text{-}node_1 \rightarrow Entity\text{-}dict \Rightarrow$

Objective Interpret a decision-node by on the basis of the question to select an answer from the answer set and interprets the associated transition or if there is not an matching answer in the answer set to interpret the else transition. An error occurs if there is no answer matching the question and no else answer.

Parameters

quest	The question of the decision.
answset	The set of answers and associated transitions.
elseansw	The optionally else transition.

Algorithm

Line 1	Extract the set of matching answers by calling *matching-answer.*
Line 2 - 5	If the extracted set of answers is not empty the it contain only one answer (it is tested in the static semantic that the answers do not overlap) and the transition associated with the selected answer is interpreted.
Line 6 - 9	If no matching answers is found then the transition associated with else answer is interpreted.
Line 10	If no matching answers is found and no else answer is supplied, an error occurs.

matching-answer(*quest*, *answset*)(*dict*) $\triangleq$ **(4.2.5)**

```
1  (let gterm = dict(TRUEVALUE) in
2   {mk-Decision-answer_1(valsetortext, ) ∈ answset |
3    (is-Range-condition_1(valsetortext) ∧ is-Expression_1(quest)
4       → (let mk-Range-condition_1(orid, cset) = valsetortext in
5           let operator_1 = make-valuetest-operator(quest, orid, cset) in
6           is-equivalent((trap exit() with false in
7                          eval-expression(operator_1)(dict)), gterm, dict(SCOPEUNIT))(dict)),
8      T → text-equality(quest, valsetortext))})
```

type : *Decision-question*$_1$ *Decision-answer*$_1$**-set** $\rightarrow$ *Entity-dict* $\rightarrow$ *Decision-answer*$_1$**-set**

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

Parameters

quest	The question of the decision.
answset	The set of answers and associated transitions.

Result The matching answer and its associated transition.

Algorithm

Line 1	Extract the AS_1 version of the AS_0 literal TRUE.
Line 2 -8	Construct the set of answers from *answset* selected by the predicates in Line 3 - 8.
Line 3 - 6	If neither the question nor the answer is informal then a value test operator is made to test whether the question match the answer or not.
Line 8	If the question or the answer is informal the equality is tested by text-equality.

make-valuetest-operator(*exp*1, *orid*, *cset*) $\triangleq$ **(4.2.6)**

```
(let v ∈ cset in
 let op = cases v:
            (mk-Closed-range_1(aop, c1, c2)
               → (let mk-Open-range_1(relop, co1) = c1 in
                   let t1 = mk-Operator-application_1(relop, ⟨co1, exp1⟩),
                       t2 = make-valuetest-operator(exp1, orid, {c2}) in
                   mk-Operator-application_1(aop, ⟨t1, t2⟩)),
             mk-Open-range_1(relop, c1)
               → mk-Operator-application_1(relop, ⟨exp1, c1⟩)) in
 if card cset = 1 then
   op
  else
   (let op' = make-valuetest-operator(exp1, orid, cset − {v}) in
    mk-Operator-application_1(orid, ⟨op, op'⟩)))
```

type : *Expression*$_1$ *Operator-identifier*$_1$ *Condition*$_1$**-set** $\rightarrow$ *Expression*$_1$

Objective Make an operator that are able to test whether the expression exp_1 fulfill the condition composed by the identified operator and *orid* and the range condition set *cset.*

Parameters

exp_1	The expression to be tested.

orid — Identifies the operator which are used to compose the condition in *cset*.

cset — Set of range condition which exp_1 should fulfill.

Result — An operator capable of testing whether the value of exp_1 match the condition composed by the operator identified by *orid* and *cset*.

Algorithm

Line 1 — Select a range condition, v, in *cset*.

Line 3 — If v is a *Closed-range*$_1$ it is decomposed into a and-operator, *aop*, and two *Open-range*$_1$ conditions, c_1 and c_2.

Line 5 — Let t_1 denote the value test operator for c_1.

Line 6 — Let t_2 denote the value test operator for c_2.

Line 7 — Compose an *Operator-application*$_1$ that applies t_1 and t_2 on *aop*, and let *op* denote this application i.e. construct the operator `"AND"("<="(co1,exp1),t2)`.

Line 8 — Compose an *Operator-application*$_1$ that applies exp_1 and c_1 on *relop* and call it v i.e. construct the operator `"<="(exp1,c1)`.

Line 11 — If v is the last element in *cset* *op* is returned.

Line 13 — Make a value test operator for the rest of the *cset* and call it *op'*.

Line 14 — Compose an *Operator-application*$_1$ where *orid* is applied the two operator applications *op* and *op'*.

text-equality(exp-text, valueset-text) $\triangleq$ (4.2.7)

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

type : $(\textit{Informal-text}_1 \mid \textit{Expression}_1)\ (\textit{Informal-text}_1 \mid \textit{Range-condition}_1) \rightarrow \textit{Bool}$

int-graph-node(graphnode)(dict) $\triangleq$ (4.2.8)

```
( cases graphnode:
  (mk-Task-node1(silt)                → int-task-node(silt)(dict),
   mk-Output-node1(,,,)               → int-output-node(graphnode)(dict),
   mk-Create-request-node1(,)         → int-create-node(graphnode)(dict),
   mk-Call-node1(,)                   → int-call-node(graphnode)(dict),
   mk-Set-node1(,,)                   → int-set-node(graphnode)(dict),
   mk-Reset-node1(,)                  → int-reset-node(graphnode)(dict)))
```

type : $\textit{Graph-node}_1 \rightarrow \textit{Entity-dict} \Rightarrow$

Objective — Interprets a graph node.

Parameters

graphnode — The graphnode to be interpreted.

int-task-node(silt)(dict) $\triangleq$ (4.2.9)

```
( cases silt:
  (mk-Assignment-statement1(,) → int-assign-stmt(silt)(dict),
   mk-Informal-text1()         → int-informal-text(silt)))
```

type : $(\textit{Assignment-statement}_1 \mid \textit{Informal-text}_1) \rightarrow \textit{Entity-dict} \Rightarrow$

Objective Interpret a task-node.

Parameters

silt	An assignment statement or informal text.

Algorithm

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

int-set-node(**mk**-*Set-node*$_1$(*texp*, *tid*, *exprl*))(*dict*) $\triangleq$ (4.2.10)

```
1  (def val : eval-expression(texp)(dict);
2   def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
3   let mk-SignalDD(sortl) = dict((tid, SIGNAL)) in
4   def vall' : ⟨reduce-term(sortl[i], vall[i], dict(SCOPEUNIT))(dict) | 1 ≤ i ≤ len vall⟩;
5   def f(t1, t2) : is-equivalent(t1, t2, dict(SCOPEUNIT))(dict);
6   if (∀i ∈ ind vall)(range-check(sortl[i], vall'[i])(dict))
7     then output mk-Set-Timer(tid, val, vall', f) to dict(PORT)
8     else exit("§5.4.1.9: Value is not within the range of the Syntype"))
```

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

Objective Interprets the set node by checking the actual parameters of the timer-signal for range-errors and then output the set timer signal to the input-port.

Parameters

texp	The timer expression whose value denote the time to which the timer should be set.
tid	The identifier of the timer to be set.
exprl	The actual parameters for the timer.

Algorithm

Line 1	Evaluate the timer expression and the list of actual parameters.
Line 4	See *reduce-term*.
Line 5	Make the *isequivalent* function to be used in the *inputport* processor to test whether this timer already is set with the same actual parameters.
Line 6	Test if the values of the actual parameters for the timer are within the range of their associated sorts.

int-reset-node(**mk**-*Reset-node*$_1$(*tid*, *exprl*))(*dict*) $\triangleq$ (4.2.11)

```
1  (def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
2   let mk-SignalDD(sortl) = dict((tid, SIGNAL)) in
3   def vall' : ⟨reduce-term(sortl[i], vall[i], dict(SCOPEUNIT))(dict) | 1 ≤ i ≤ len vall⟩;
4   def f(t1, t2) : is-equivalent(t1, t2, dict(SCOPEUNIT))(dict);
5   if (∀i ∈ ind vall')(range-check(sortl[i], vall'[i])(dict))
6     then output mk-Reset-Timer(tid, vall', f) to dict(PORT)
7     else exit("§5.4.1.9: Value is not within the range of the Syntype"))
```

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

Objective Interpret the reset-node by checking whether the actual parameters of the timer signal are within the range of their sorts, and if so, output the *Reset-Timer* signal to the *input-port* processor.

Parameters

tid	The identifier of the timer to be reset.
exprl	The actual parameters for the timer.

Algorithm

Line 1	Evaluate the timer the list of actual parameters.
Line 3	See *Reduce-term.*
Line 4	Make the is-equivalent function to be used in the *input-port* processor to test whether this timer already is set with the same actual parameters.
Line 5	Test if the values of the actual parameters for the timer are within the range of their associated sorts.

int-assign-stmt(**mk**-*Assignment-statement*$_1$(*vid*, *exp*))(*dict*) $\triangleq$ **(4.2.12)**

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

type : *Assignment-statement*$_1$ $\rightarrow$ *Entity-dict* $\Rightarrow$

Objective The variable is assigned to the value of the expression.

Parameters

vid	The 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*$_1$()) $\triangleq$ **(4.2.13)**

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

type : *Informal-text*$_1$ $\Rightarrow$

int-output-node(**mk**-*Output-node*$_1$(*sid*$_1$, *exprl*, *dest*, *via*))(*dict*) $\triangleq$ **(4.2.14)**

1 (**def** *vall* : $\langle$*eval-expression*(*exprl*[*i*])(*dict*) $\mid 1 \leq i \leq$ **len** *exprl*$\rangle$;
2 **def** *pidval* : *eval-expression*(*dest*)(*dict*);
3 **let mk**-*SignalDD*(*sortl*) = *dict*((*sid*$_1$, SIGNAL)) **in**
4 **def** *vall'* : $\langle$*reduce-term*(*sortl*[*i*], *vall*[*i*], *dict*(SCOPEUNIT))(*dict*) $\mid 1 \leq i \leq$ **len** *vall*$\rangle$;
5 **if** ($\forall i \in$ **ind** *vall'*)(*range-check*(*sortl*[*i*], *vall'*[*i*])(*dict*))
6 **then output mk**-*Send-Signal*(*sid*$_1$, *vall'*, *pidval*, *via*) **to** *system*
7 **else exit**("§5.4.1.9: Value is not within the range of the Syntype"))

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

Objective Interpret an output node by checking if the actual parameters are within the range of their sorts, if so, Send-Signal is output to the system processor.

Parameters

sid_1	The identifier of the signal to be send.
exprl	The actual parameters for the signal.
dest	A PID expression denoting the process to which the signal should be send.
via	A set of path identifiers denoting the path the signal should follow.

Algorithm

Line 1	Evaluate the list of actual parameters and the pid-value.
Line 4	See *reduce-term*.
Line 5	Test if the values of the actual parameters for the signal are within the range of their associated sorts.

int-create-node(**mk**-*Create-request-node*$_1$(*pid*, *exprl*))(*dict*) $\triangleq$ (4.2.15)

```
1   (def vall : ⟨eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl⟩;
2    let mk-ProcessDD(formparms, , , , ) = dict((pid, PROCESS)) in
3    let sortl = ⟨s-Sort-reference-identifier1(dict((formparms[i], VALUE))) | 1 ≤ i ≤ len formparms⟩ in
4    def vall' : ⟨reduce-term(sortl[i], vall[i], dict(SCOPEUNIT))(dict) | 1 ≤ i ≤ len vall⟩;
5    output mk-Create-Instance-Request(pid, vall') to system;
6    input mk-Create-Instance-Answer(offspring') from system
7        ⇒  if offspring' = nil then
8              (let nullterm = dict(NULLVALUE) in
9               offspring := nullterm)
10            else
11             offspring := offspring')
```

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

Objective Interprets the create node.

Parameters

pid	The identifier of the process to be created.
exprl	The list of actual parameters.

Algorithm

Line 1	Evaluate the value of each actual parameter.
Line 2	Establish the list of Sort-reference-identifiers of the formal parameters.
Line 4	See *reduce-term*.
Line 5	Output an Create-Instance-Request to the system processor.
Line 6	Input either the PID value of the created process or, if the process could not be created, nil.
Line 9	If the process could not be created the offspring assigned to the *nullterm*. A new process cannot be created if there already exists the maximal number of instances of that process.
Line 11	If the process could be created then offspring is assigned to the PID value received from the system processor.

int-call-node(**mk-***Call-node*$_1$(*prd-id*, *exprl*))(*dict*) $\triangleq$ **(4.2.16)**

1 (**dcl** newstg := [] **type** *Stg*;
2 **let mk-***ProcedureDD*(*formparms*, *graph*) = *dict*((*prd-id*, PROCEDURE)) **in**
3 **let mk-***Identifier*$_1$(*qual*, *nm*) = *prd-id* **in**
4 **let** *newlevel* = *qual* $\frown$ ⟨**mk-***Procedure-qualifier*$_1$(*nm*)⟩ **in**
5 **let** *decl-parm-set* = {(**mk-***Identifier*$_1$(*l*,), VALUE) $\in$ **dom** *dict* | *l* = *newlevel*} **in**
6 **let** *dict*$_1$ = *establ-dyn-dict*(*formparms*, *exprl*, newstg, *decl-parm-set*)(*dict*) **in**
7 **let** *dict*$_2$ = *dict*$_1$ + [SCOPEUNIT $\mapsto$ *newlevel*] **in**
8 (**trap exit**(RETURN) **with I in**
9 *int-procedure-graph*(*graph*)(*dict*$_2$)))

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

Objective Interpret a procedure call node.

Parameters

prd-id	The identifier of the procedure to be called.
exprl	The actual parameters for the procedure call.

Algorithm

Line 1	Declare a new empty storage variable to be used as the storage local to the procedure.
Line 2	Establish the list of procedure formal parameters and the procedure graph.
Line 3 - 4	Calculate the scope level of the procedure.
Line 5	Extract the set of variables defined or used as formal parameters on this level.
Line 6	Make a new dict and update the new storage according to the new definitions and the formal and actual parameters by calling *establ-dyn-dict*.
Line 7 - 9	Enter the new level and trap **exit**(RETURN) from the interpretation of the procedure graph by doing nothing.

int-procedure-graph(*graph*)(*dict*) $\triangleq$ **(4.2.17)**

1 (**let mk-***Procedure-graph*$_1$(**mk-***Procedure-start-node*$_1$(*trans*), *stateset*) = *graph* **in**
2 **tixe** [*statenm* $\mapsto$ *int-state-node*(*statenode*)(*dict*) |
3 *statenode* $\in$ *stateset* $\wedge$ **s-***State-name*$_1$(*statenode*) = *statenm*] **in**
4 **let mk-***Transition*$_1$(*nodel*, *termordec*) = *trans* **in**
5 *int-transition*(*nodel*, *termordec*)(*dict*))

type : *Procedure-graph*$_1$ $\rightarrow$ *Entity-dict* $\Rightarrow$

Objective Interpret a procedure graph.

Parameters

graph	The procedure graph.

Algorithm

Line 1	Partition of the graph into a start transition and a set of states.
Line 2	Trap all **exit**(*statenm*) from int-state-node and int-transition by interpreting the associated state-node (An explanation of the tixe construct is given in the annotations of int-process-graph).
Line 4	Interpretation of the start-transition.

4.3 Auxiliary functions

The following defines the auxiliary functions used in the previous sections. The auxiliary functions evaluate expressions, perform range check, manages the local storage and the dynamic part of the entity dict (see section 2.6).

eval-expression(exp)(dict) $\triangleq$ (4.3.1)

```
1  (if exp = nil then
2     nil
3    else
4     cases exp:
5      (mk-Identifier_1(, )
6          → eval-variable-identifier(exp)(dict),
7       mk-Ground-expression_1()
8          → eval-ground-expression(exp)(dict),
9       mk-Operator-application_1(, )
10         → eval-operator-application(exp)(dict),
11      mk-Conditional-expression_1(e1, e2, e3)
12         → eval-conditional-expression(e1, e2, e3)(dict),
13      mk-View-expression_1(, )
14         → eval-view-expression(exp)(dict),
15      mk-Timer-active-expression_1(, )
16         → eval-active-expression(exp)(dict),
17      mk-Now-expression_1()
18         → eval-now-expression(),
19      mk-Self-expression_1()
20         → mk-Ground-term_1(dict(SELF)),
21      mk-Parent-expression_1()
22         → mk-Ground-term_1(dict(PARENT)),
23      mk-Offspring-expression_1()
24         → mk-Ground-term_1(c offspring),
25      mk-Sender-expression_1()
26         → mk-Ground-term_1(c sender)))
```

type : $[Expression_1] \rightarrow Entity\text{-}dict \Rightarrow [Value]$

Objective Evaluate an AS_1 expression.

Parameters

The AS_1 expression.

Result The value of the expression.

Algorithm

Line 1 If the expression equals nil the result is the nil value.

Line 19 If the expression is a Self, Parent, Offspring or Sender-expression the ground-term of the contents of self, parent, offspring or sender is returned.

eval-variable-identifier(*id*)(*dict*) $\triangleq$ **(4.3.2)**

```
1  (let mk-VarDD(vid, , , stg) = dict((id, VALUE)) in
2   if c stg(vid) = UNDEFINED
3     then exit("§5.5.2.2: Value of accessed variable is undefined")
4     else c stg(vid))
```

type : $Identifier_1 \rightarrow Entity\text{-}dict \Rightarrow Value$

Objective Evaluate a variable identifier.

Parameters

id The variable identifier.

Result The contents, if any, of that variable.

Algorithm

Line 1 Gets the referenced identifier.

Line 3 If the contents of storage for the referenced identifier is undefined an error occurs.

Line 4 The contents of storage for the referenced identifier is returned.

eval-ground-expression(**mk-**$Ground\text{-}expression_1$(*ex*))(*dict*) $\triangleq$ **(4.3.3)**

```
1  (let mk-Ground-term1(e) = ex in
2   if is-Identifier1(e) then
3     ex
4    else
5     if is-Conditional-term1(e) then
6       (let mk-Conditional-term1(bex, ex1, ex2) = e in
7        eval-conditional-expression(bex, ex1, ex2)(dict))
8      else
9       (let (opid, arglist) = e in
10       let mk-OperatorDD(sortlist, sort) = dict((opid, VALUE)) in
11       if (∀i ∈ ind arglist)(range-check(sortlist[i], arglist[i])(dict)) then
12         (let arglist' = ⟨eval-expression(arglist[i])(dict) | 1 ≤ i ≤ len arglist⟩ in
13          let t = mk-Ground-term1((opid, arglist')) in
14          if range-check(sort, t)(dict)
15            then t
16            else exit("§5.4.1.9: Value is not within the range of the Syntype"))
17        else
18         exit("§5.4.1.9: Value is not within the range of the Syntype")))
```

type : $Ground\text{-}expression_1 \rightarrow Entity\text{-}dict \rightarrow Value$

Objective Evaluate a ground expression.

Parameters

ex A ground term.

Result The value of the ground expression, given as the operator identifier and the evaluated argument list.

Algorithm

Line 2 If the ground term consist of an identifier the ground term is returned.

Line 6	If the ground term is a conditional term then it is decomposed and evaluated.
Line 9	If the ground term neither is an identifier nor a conditional expression it must be an operator application.
Line 11	The argument list of the operator is tested for range errors according to its associated sortlist. If no range error is detected the operator identifier and the evaluated sort list is composed into a ground term. Otherwise range error occurs.
Line 14	The ground term is tested for range error according to the sort of the operator. If no range error is detected the ground term is returned.

eval-operator-application(**mk-***Operator-application*$_1$(*opid*, *expl*))(*dict*) $\triangleq$ **(4.3.4)**

```
1  (def vall : ⟨eval-expression(expl[i])(dict) | 1 ≤ i ≤ len expl⟩;
2   let term = mk-Ground-term1((opid, vall)) in
3   let mk-OperatorDD(sortl, result) = dict((opid, VALUE)) in
4   if (∀i ∈ ind sortl)(range-check(sortl[i], vall[i])(dict)) ∧
5     range-check(result, term)(dict)
6     then term
7     else exit("§5.4.1.9: Value is not within the range of the Syntype"))
```

type : *Operator-application*$_1$ $\rightarrow$ *Entity-dict* $\Rightarrow$ *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, that is a ground term of the operator identifier and the evaluated argument list.

Algorithm

Line 1	Evaluate the list of arguments.
Line 2	Make a ground term of the operator identifier and the evaluated argument list.
Line 3	Lookup the operators denotation in *Entity-dict*.
Line 4	Test whether the evaluated arguments are within the range of their associated sorts.
Line 5	Test whether the term made in *line 2* is within the range of the sort of the result.
Line 6	If no range error is found the term made in *line 2* is returned.

eval-view-expression(**mk-***View-expression*$_1$(*id*$_1$, *exp*))(*dict*) $\triangleq$ **(4.3.5)**

```
(def pid : eval-expression(exp)(dict);
 def pid' : reduce-term(dict(PIDSORT), pid, dict(SCOPEUNIT))(dict);
 output mk-View-Request(id1, pid') to view;
 (input mk-View-Answer(val) from view
     ⇒  if val = UNDEFINED
           then exit("§5.5.2.2: The viewed value is undefined")
           else val))
```

type : *View-expression*$_1$ $\rightarrow$ *Entity-dict* $\Rightarrow$ *Value*

Objective Evaluate a VIEW expression

eval-conditional-expression(exp_1, exp_2, exp_3)(*dict*) $\triangleq$ **(4.3.6)**

```
1  (let trueterm = dict(TRUEVALUE) in
2   let falseterm = dict(FALSEVALUE) in
3   if is-equivalent(eval-expression(exp1)(dict), trueterm, dict(SCOPEUNIT))(dict) then
4     eval-expression(exp2)(dict)
5    else
6     if is-equivalent(eval-expression(exp1)(dict), falseterm, dict(SCOPEUNIT))(dict) then
7       eval-expression(exp3)(dict)
8      else
9       exit("§5.5.2.3: Condition must evaluate to TRUE or FALSE"))
```

type : $Expression_1$ $Expression_1$ $Expression_1$ $\rightarrow$ *Entity-dict* $\stackrel{\sim}{\Rightarrow}$ *Value*

Objective Evaluate a conditional expression.

Parameters

exp_1	The condition expression.
exp_2	The consequence expression.
exp_3	The alternative expression.

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

Algorithm

Line 1	Extract the AS_1 term for **TRUE**.
Line 2	Extract the AS_1 term for **FALSE**.
Line 3	If the trueterm is equal to the condition then
Line 4	Evaluate the consequence expression else
Line 6	If the falseterm is equal to the condition then
Line 7	Evaluate the alternative expression else
Line 9	it is an error (only possible if the Boolean sort has been enriched by additional values).

eval-active-expression(**mk-**$Timer\text{-}active\text{-}expression_1$(*timer*, *exprl*))(*dict*) $\triangleq$ **(4.3.7)**

```
1  (let mk-Identifier1(qual, ) = timer in
2   let mk-SignalDD(sortl) = dict((timer, SIGNAL)) in
3   let trueterm = dict(TRUEVALUE),
4       falseterm = dict(FALSEVALUE) in
5   def vall : <eval-expression(exprl[i])(dict) | 1 ≤ i ≤ len exprl>;
6   def vall' : <reduce-term(sortl[i], vall[i], dict(SCOPEUNIT))(dict) | 1 ≤ i ≤ len vall>;
7   let f(t1, t2) = is-equivalent(t1, t2, qual)(dict) in
8   output mk-Active-Request(timer, vall', f) to dict(PORT);
9   (input mk-Active-Answer(b) from dict(PORT)
10      ⇒  if b then mk-Ground-term1(trueterm) else mk-Ground-term1(falseterm)))
```

type : $Timer\text{-}active\text{-}expression_1$ $\rightarrow$ *Entity-dict* $\Rightarrow$ *Value*

Objective Test whether the depicted timer is active.

Parameters

timer	The identifier of the timer.
exprl	The parameters of the timer.

Result The AS_1 value of TRUE if the depicted timer is active else the AS_1 value of FALSE.

Algorithm

Line 1	Establish the sort list of the timer.
Line 3-4	Extract the AS_1 value of TRUE and FALSE.
Line 5	Evaluate the timer parameters.
Line 6	See *reduce-term*
Line 7	Define a function to test for equivalence between *vall'* and the parameters of the potential active timer.
Line 8	Send an *Active-Request* to the input port.
Line 9	Receive an *Active-Answer* from the input port with a parameter *b*, denoting the "activeness" of the timer.
Line 10	Return the AS_1 version of TRUE or FALSE depending on *b*.

eval-now-expression() $\triangleq$ (4.3.8)

```
1  (output mk-Time-Request() to timer;
2   (input mk-Time-Answer(val) from timer
3       ⇒  val))
```

type : () ⇒ *Value*

Objective Evaluate the now expression.

Result A value denoting now, see the *Timer* processor.

Algorithm

Line 1	Send a *Time-Request* to the timer processor.
Line 2	Receive *Time-Answer* with the value of now.

establ-dyn-dict(formparml, exprl, stg, decl-parm-set)(dict) $\triangleq$ (4.3.9)

```
(if decl-parm-set ≠ {} then
   (let (id, VALUE) ∈ decl-parm-set in
    def dict1 : (mk-InoutparmDD(id) ∈ elems formparml
                    → (let i ∈ ind formparml be s.t. formparml[i] = mk-InoutparmDD(id) in
                        let mk-VarDD(vid, sid, rev, stg') = dict((exprl[i], VALUE)) in
                        [(id, VALUE) ↦ mk-VarDD(vid, sid, rev, stg')]),
                 mk-InparmDD(id) ∈ elems formparml
                    → (let i ∈ ind formparml be s.t. formparml[i] = mk-InparmDD(id) in
                        let mk-VarDD(vid, sid, rev, ) = dict((id, VALUE)) in
                        let dict' = [(id, VALUE) ↦ mk-VarDD(vid, sid, rev, stg)] in
                        update-stg(vid, eval-expression(exprl[i])(dict))(dict + dict');
                        dict'),
                 T → (let mk-VarDD(vid, sid, rev, ) = dict((id, VALUE)) in
                        let dict' = [(id, VALUE) ↦ mk-VarDD(vid, sid, rev, stg)] in
                        update-stg(vid, nil)(dict + dict');
                        dict'));
    return establ-dyn-dict(formparml, exprl, stg, decl-parm-set \ {(id, VALUE)})(dict) + dict1)
  else
   dict)
```

type : *FormparmDD** $Expression_1$* **ref** *Stg* ($Identifier_1$ VALUE)-**set** → *Entity-dict* ⇒ *Entity-dict*

Objective Perform the necessary changes in the dynamic part of entity-dict when a procedure call is interpreted. Futhermore the storage is updated according to the variables defined in the procedure and the formal IN parameters.

Parameters

formparml	The list of formal parameters.
actparml	The list of actual parameters.
stg	A reference to the new storage.
dcl-parm-set	The set of *dict* entries for which the *dict* and the storage should be updated.

Result The updated *Entity-dict*.

Algorithm

Line 1	The recursion stops if the set of dict entries is empty.
Line 2	Take one of the dict entries.
Line 3	If it corresponds to one of the formal IN/OUT parameters then:
Line 4 - 5	Lookup the associated actual parameter, and its descriptor in *dict*.
Line 6	Let the variable of the formal parameter be described by the variable descriptor of the actual parameter.
Line 7	If it corresponds to one of the formal IN parameters then:
Line 8	Lookup the index of the formal parameter.
Line 9 -10	Change the variable descriptors of the formal parameter to reference the new storage.
Line 11	Update the storage for the variable with the value of the actual parameter using the new descriptor.
Line 13	If it neither corresponds to a formal IN nor to a formal IN/OUT parameter then:
Line 13 - 16	It is treated like the formal IN parameter, but the storage is updated with **nil** (UNDEFINED).
Line 17	Call *establ-dyn-dict* for the rest of the *dcl-parm-set* and overwrite the result with the just constructed entry.
Line 19	Stops the recursion when there are no more definitions or parameters to examine and return the old dict.

update-stg(*id*, *val*)(*dict*) $\triangleq$ (4.3.10)

```
(let mk-VarDD(vid, sid, revealed, stg') = dict((id, VALUE)) in
 def val' : if val = nil then
               UNDEFINED
              else
               reduce-term(sid, val, dict(SCOPEUNIT))(dict);
 if range-check(sid, val')(dict)
   then (stg' := c stg' + [vid ↦ val'];
         if revealed = REVEALED then
           (output mk-Reveal(vid, val', dict(SELF)) to view)
          else
           I)
   else exit("§5.4.1.9: Value is not within the range of the Syntype"))
```

type : $Identifier_1$ *Value* $\rightarrow$ *Entity-dict* $\Rightarrow$

Objective Updates the storage for the applied variable identifier with the applied value and reveal the variable if it is declared revealed.

Parameters

id	The variable identifier for which the storage should be updated.
val	The value with which the storage should be updated.

Algorithm

Line 1	Lookup the description of the variable identifier.
Line 2	If *val* is different from **nil** it must be changed to match the sort identifier of the variable (See *reduce-term*), if the value is equal to **nil** the storage will be updated with UNDEFINED.
Line 7 - 8	The referenced storage is overwritten with the new variable - value pair.
Line 9	For revealed variables Reveal is send to the view processor with the variable identifier, the "reduced" value and the PID value of this process.
Line 12	If the "reduced" value is not within the range of the variable sort, a range error occurs.

range-check(*sort-id*, *value*)(*dict*) $\triangleq$ (4.3.11)

```
1  if value ∈ {nil, UNDEFINED} then
2    true
3   else
4    ( cases dict((sort-id, SORT)):
5      (mk-SyntypeDD(, mk-Range-condition₁(orid, condset))
6        → (let operator₁ = make-valuetest-operator(value, orid, condset) in
7           let value' = eval-ground-expression(operator₁)(dict) in
8           let trueterm = dict(TRUEVALUE) in
9           is-equivalent(eval-expression(value')(dict), trueterm, dict(SCOPEUNIT))(dict)),
10      T → true))
```

type : $\mathit{Sort\text{-}reference\text{-}identifier}_1\ [\mathit{Value}] \rightarrow \mathit{Entity\text{-}dict} \rightarrow \mathit{Bool}$

Objective Test whether a value is within the range of its sort.

Parameters

sort-id	The sort identifier.
value	The value to be tested.

Result True if the value is in range else false.

Algorithm

Line 1	**nil** or UNDEFINED is within the range of all sorts.
Line 4	Lookup the description of the sort.
Line 6	If the sort is a syntype then use the *orid*, the *condset* and the *value* to make an SDL-operator ($\mathit{operator}_1$) to perform the range check. See *make-valuetest-operator*.
Line 8	Extract the AS_1 version of true.

5 Construction of *Entity-dict* and Handling of Abstract Data Types

This section contains the functions which build the *entitydict* (see the domain definition of *Entity-dict*). *entitydict* is used by the *sdl-process* as well as by the *System* process which also creates the object by applying the entry function *extract-dict*.

The section is divided into four subsections :

1. The creation of simple self-contained descriptors such as descriptors for syntypes, variables, signals etc. Also the descriptors for processes (i.e. *ProcessDDs*) are created, but with an empty *Reachability* set.

   Descriptors are created for entities regardless of whether 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-definition*$_1$s (*TypeDD*). For each scopeunit, this descriptor is created before the descriptors for the sorts (*SortDD*) 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.)

The selection of the consistent subset is made after descriptors for all the entities are constructed, but before *Reachabilities* are constructed by removing the descriptors of the processes which are not in the consistent subset. The construction of the *entitydict* 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:

- The check on selection of a consistent refinement subset requires construction of *Reachabilities*.

  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.

- Consistency checks on equivalent classes and on mutual exclusion of decision answers cannot easily by expressed in terms of AS$_1$, i.e. these (static) checks are placed in the Dynamic Semantics because construction of the equivalent classes is required.

$$\mathit{extract\text{-}dict}(\mathit{as_1tree}, \mathit{blockset}, \mathit{expiredf}, \mathit{terminf}) \triangleq \tag{5.1}$$

$$
\begin{array}{l}
(\textbf{let } (\mathit{as_1pid}, \mathit{as_1null}, \mathit{as_1true}, \mathit{as_1false}) = \mathit{terminf} \textbf{ in} \\
\;\textbf{let } d = [\begin{array}[t]{ll}
\mathsf{EXPIREDF} & \mapsto \mathit{expiredf}, \\
\mathsf{PIDSORT} & \mapsto \mathit{as_1pid}, \\
\mathsf{NULLVALUE} & \mapsto \mathit{as_1null}, \\
\mathsf{TRUEVALUE} & \mapsto \mathit{as_1true}, \\
\mathsf{FALSEVALUE} & \mapsto \mathit{as_1false}] \textbf{ in}
\end{array} \\
\;(\textbf{let mk-}\mathit{System\text{-}definition_1}(\mathit{nm}, \mathit{bset}, \mathit{cset}, \mathit{sigset}, \mathit{tp}, \mathit{synset}) = \mathit{as_1tree} \textbf{ in} \\
\;\;\textbf{let } \mathit{level} = \langle \textbf{mk-}\mathit{System\text{-}qualifier_1}(\mathit{nm}) \rangle \textbf{ in} \\
\;\;\textbf{let } \mathit{leafprocesses} = \mathit{select\text{-}consistent\text{-}subset}(\mathit{bset}, \mathit{blockset}, \mathit{level}) \textbf{ in} \\
\;\;\textbf{let } \mathit{dict}' = \mathit{extract\text{-}sortdict}(\mathit{tp}, \mathit{level})(d) \textbf{ in} \\
\;\;\textbf{let } \mathit{dict}'' = \textbf{merge}\, \{\mathit{make\text{-}entity}(\mathit{entity}, \mathit{level})(\mathit{dict}') \mid \mathit{entity} \in (\mathit{sigset} \cup \mathit{synset} \cup \mathit{bset})\} \textbf{ in} \\
\;\;\textbf{let } d' = \mathit{dict}'' + [(\mathit{id}, q) \mapsto d((\mathit{id}, q)) \mid (\mathit{id}, q) \in \textbf{dom}\, d \wedge (q = \mathsf{PROCESS} \supset \mathit{id} \in \mathit{leafprocesses})] \textbf{ in} \\
\;\;\mathit{make\text{-}structure\text{-}paths}(\mathit{bset}, \mathit{cset}, \mathit{level})(d')))
\end{array}
$$

type : *System-definition*$_1$ *Block-identifier*$_1$**-set** *Is-expired* *Term-information* $\rightarrow$ *Entity-dict*

Objective Construct the *entitydict* which is used by the *sdl-processes* and by the *system* process. The object is constructed by the *system* process and given as actual parameter every time a new *sdl-process* is started.

Parameters

as_1 tree The abstract syntax representation of a 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 also is in the consistent subset, it is not included in *blockset*

expiredf A function delivering true if a given timer has expired

terminf Some AS_1 identifiers used by the underlying system.

Result An object of the domain *Entity-dict*

Algorithm

Line 1 Decompose the *Term-information* (defined in Annex F.2) which contains the *Identifier$_1$s* of the PiD sort, the NULL literal, the TRUE literal and FALSE literal.

Line 2-6 Create the initial *entitydict* wherein the expired function and the term information are placed.

Line 8 Make the qualifier which denotes the system level.

Line 9 Check that the consistent subset is well-formed and extract the identifiers of the processes which are contained in the consistent subset

Line 10 Create the descriptors for the *Data-type-definition$_1$*, the literals and the operators defined on the system level.

Line 11 Make the *entitydict* contributions for the signals (*sigset*), the syntypes (*synset*) and the blocks (*bset*).

Line 12 Remove the descriptors of the processes which are not in the consistent subset.

Line 13 Insert *Reachabilities* in all the process descriptors (*ProcessDDs*). *make-structure-paths* returns the *entitydict* where they have been inserted.

5.1 Construction of Descriptors for Simple Objects

make-entity(*entity*, *level*)(*dict*) $\triangleq$ **(5.1.1)**

```
cases entity:
 (mk-Timer-definition1(nm, sortlist)
    → dict + [(mk-Identifier1(level, nm), SIGNAL) ↦ mk-SignalDD(sortlist)],
  mk-Signal-definition1(, , )
    → dict + make-signal-dict(entity, level),
  mk-Process-definition1(, , , , , , , , , , , )
    → make-process-dict(entity, level)(dict),
  mk-Procedure-definition1(, , , , , , , )
    → make-procedure-dict(entity, level)(dict),
  mk-Variable-definition1(nm, sort, rev)
    → dict + [(mk-Identifier1(level, nm), VALUE) ↦ mk-VarDD(, sort, rev, )],
  mk-Syn-type-definition1(nm, psort, ran)
    → dict + [(mk-Identifier1(level, nm), SORT) ↦ mk-SyntypeDD(psort, ran)],
  mk-Block-definition1(, , , , , , , , )
    → make-block-dict(entity, level)(dict),
  T → dict)
```

type : $Decl_1\ Qualifier_1 \rightarrow Entity\text{-}dict \rightarrow Entity\text{-}dict$

Objective Return the *Entity-dict* (*dict*) which is updated to contain the contribution for an entity.

Parameters

entity	The AS_1 definition for the entity
level	A qualifier denoting the scopeunit containing the definition

Algorithm Construct the contribution for entity in hand. Note that a timer is treated as a normal signal and that no descriptor is required for view variables (there is no *View-definition*$_1$ entry in the case statement).

make-signal-dict(**mk**-*Signal-definition*$_1$(*nm*, *sortlist*, *refinement*), *level*) $\triangleq$ **(5.1.2)**

```
(let d = [(mk-Identifier1(level, nm), SIGNAL) ↦ mk-SignalDD(sortlist)] in
 if refinement = nil then
   d
  else
   (let mk-Signal-refinement1(subsigset) = refinement in
    let level' = level ⌢ ⟨mk-Signal-qualifier1(nm)⟩ in
    d + merge {make-signal-dict(s-Signal-definition1(sdef), level') | sdef ∈ subsigset}))
```

type : $Signal\text{-}definition_1\ Qualifier_1 \rightarrow Entity\text{-}dict$

Objective Make the *entitydict* contribution for a signal and for its sub-signals. Note that a signal descriptor does not tell whether a signal is a subsignal or not. This is due to the fact that only if the same signals are selected in both ends of a communication link the subsignal selection is well-formed, independently of whether the signals are sub-signals or not.

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-process-dict(pdef, level)(dict) $\triangleq$ **(5.1.3)**

```
1  (let mk-Process-definition₁(nm, inst, f, pset, sigset, tp, synset, vset, , tset, graph) = pdef in
2   let mk-Number-of-instances₁(init, maxi) = inst,
3       pid = mk-Identifier₁(level, nm),
4       level' = level ⁀ ⟨mk-Process-qualifier₁(nm)⟩ in
5   let parm = ⟨mk-Identifier₁(level', s-Variable-name₁(f[i])) | 1 ≤ i ≤ len f⟩ in
6   let parmd = [(parm[i], VALUE) ↦ mk-VarDD(, s-Sort-reference-identifier₁(f[i]), nil, ) |
7                   1 ≤ i ≤ len f] in
8   let dict' = extract-sortdict(tp, level')(dict + parmd) in
9   let dict'' = merge {make-entity(entity, level')(dict') |
10                         entity ∈ (pset ∪ sigset ∪ synset ∪ vset ∪ tset)} in
11  let mk-Process-graph₁(, stateset) = graph in
12  let nodeset = union {statenodeset | mk-State-node₁(, , statenodeset) ∈ stateset} in
13  let insigset = {sigid | mk-Input-node₁(sigid, , ) ∈ nodeset} in
14  let localreach = (pid, insigset, ⟨⟩) in
15  if is-wf-decision-answers(graph, level)(dict) then
16     dict'' + [(pid, PROCESS) ↦ mk-ProcessDD(parm, init, maxi, graph, {localreach})]
17    else
18     exit("§2.7.5: Answers in decision actions are not mutually exclusive"))
```

type : *Process-definition*$_1$ *Qualifier*$_1$ $\rightarrow$ *Entity-dict* $\rightarrow$ *Entity-dict*

Objective Return the *entitydict* contribution for a process and for all its definitions.

Parameters

pdef	The AS_1 process definition
level	A qualifier denoting the scopeunit where the process is defined.

Algorithm

Line 2	Extract the initial number of instances (*init*) and the maximum number of instances (*maxi*).
Line 3	Construct the *Identifier*$_1$ denoting the process
Line 4	Construct the qualifier denoting the scopeunit which is the process
Line 5	Construct the *Identifier*$_1$s for the formal parameters
Line 6	Construct the *Entity-dict* contributions for the formal parameters. Note that they are treated as normal variables.
Line 8	Make the *entitydict* which is updated with the descriptor for the *Data-type-definition*$_1$ defined in the process
Line 9-10	Make the contributions for the contained procedure definitions (*pset*), signal definitions (*sigset*), syntype definitions (*synset*), variable definitions (*vset*), and timer definitions (*tset*)
Line 11	Let *stateset* denote the set of states for the process
Line 12	Let *nodeset* denote the set of input nodes for the process
Line 13	Let *insigset* denote the set of signals received in an input node of the process

Line 14 — Let *localreach* denote the *Reachability* which is used for routing signals to instances of this process type.

Line 15 — The decision actions contained in the process graph must contain mutual exclusive answers

Line 16 — and update the constructed *entitydict* with the descriptor for the process itself. Note that, at this stage, the *Reachability* set for the process only contains the *Reachability* used for routing signals to instances of this process type.

$$make\text{-}procedure\text{-}dict(procdef, level)(dict) \triangleq \tag{5.1.4}$$

```
1  (let mk-Procedure-definition₁(nm, fp, pset, tp, sset, vset, graph) = procdef in
2   let level' = level ⌢ ⟨mk-Procedure-qualifier₁(nm)⟩ in
3   let (fparml, fdict) = make-formal-parameters(fp, level) in
4   let pid = mk-Identifier₁(level, nm) in
5   let dict' = extract-sortdict(tp, level')(dict + fdict) in
6   let dict'' = merge {make-entity(entity, level')(dict') | entity ∈ (pset ∪ sset ∪ vset)} in
7   if is-wf-decision-answers(graph, level)(dict) then
8      dict'' + [(pid, PROCEDURE) ↦ mk-ProcedureDD(fparml, graph)]
9     else
10     exit("§2.7.5: Answers in decision actions are not mutually exclusive"))
```

type : $Procedure\text{-}definition_1\ Qualifier_1 \rightarrow Entity\text{-}dict \rightarrow Entity\text{-}dict$

Objective — Return the *entitydict* contribution for a procedure and for all its definitions.

Parameters

procdef — The AS_1 procedure definition

level — A qualifier denoting the scopeunit where the procedure is defined.

Algorithm

Line 2 — Construct the qualifier denoting the procedure scopeunit

Line 3 — Construct the information about whether the formal parameters are IN or IN/OUT (*fparml*) and the *entitydict* descriptors for the formal parameters (*fdict*).

Line 4 — Construct the qualifier denoting the scopeunit which is the procedure

Line 5 — Same as for a process (see above).

Line 6 — Construct the descriptors for the contained procedure definitions (*pset*), syntype definitions (*sset*), and variable definitions (*vset*).

Line 7 — The decision actions contained in the procedure graph must contain mutual exclusive answers

Line 8 — Construct the descriptor for the procedure itself.

make-formal-parameters(parml, level) ≜ (5.1.5)

```
1   (if parml = ⟨⟩ then
2      (⟨⟩, [])
3      else
4      (let (parmrest, drest) = make-formal-parameters(tl parml, level) in
5       let id = mk-Identifier1(level, s-Variable-name1(hd parml)) in
6       let (p, d) =
7            cases hd parml:
8             (mk-In-parameter1(, sort)
9                → (mk-InparmDD(id), [(id, VALUE) ↦ mk-VarDD(, sort, nil, )]),
10              mk-Inout-parameter1(, )
11               → (mk-InoutparmDD(id), [])) in
12      (⟨p⟩ ⌢ parmrest, d + drest)))
```

type : *Procedure-formal-parameter*$_1$* *Qualifier*$_1$ → *FormparmDD** *Entity-dict*

Objective Construct (recursively) and return a list of descriptors containing information of whether the formal parameters are IN or IN/OUT parameters and also return the *entity* descriptors for them.

Parameters

parml	The AS$_1$ procedure formal parameters
level	A qualifier denoting the scopeunit of the procedure.

Algorithm

Line 1	If at the end of the list of formal parameters then return nothing
Line 4	Construct the descriptors for the rest of the list
Line 5	Make the *Identifier*$_1$ for the first parameter in the list
Line 6-10	Make the parameter descriptor and the *entitydict* contribution for the first parameter joined with those for the rest in the list and return the *entitydict* descriptors for the first parameter joined with those for the rest in the list

make-block-dict(bdef, level)(dict) ≜ (5.1.6)

```
(let mk-Block-definition1(bnm, pdefs, sigdefs, , , datatype, syntype, sub) = bdef in
 let level' = level ⌢ ⟨mk-Block-qualifier1(bnm)⟩ in
 let sortd = extract-sortdict(datatype, level')(dict) in
 let dict' = sortd + merge {make-entity(entity, level')(sortd) | entity ∈ (sigdefs ∪ syntype ∪ pdefs)} in
 if sub = nil then
    dict'
   else
   (let mk-Block-substructure-definition1(snm, bdefs, , , sdefs, tp, syndefs) = sub in
    let level'' = level' ⌢ ⟨mk-Block-substructure-qualifier1(snm)⟩ in
    let sortd' = extract-sortdict(tp, level'')(dict') in
    sortd' + merge {make-entity(entity, level'')(sortd') | entity ∈ (bdefs ∪ sdefs ∪ syndefs)}))
```

type : *Block-definition*$_1$ *Qualifier*$_1$ → *Entity-dict* → *Entity-dict*

Objective Construct and return the *entitydict* descriptors for the entities defined in a block. Note that enclosed signal route definitions, channel definitions, connections etc. are not dealt with here.

Parameters

bdef	An AS$_1$ block definition
level	The defining qualifier for the block

Algorithm

Line 1	Decompose the block definition.
Line 2	Construct the qualifier which denotes the block.
Line 3	Update *entitydict* to include the $Data\text{-}type\text{-}definition_1$ defined in the block.
Line 4	Update *entitydict* to include the signals (*sigdefs*), syntypes (*syntype*) and processes (*pdefs*) defined in the block.
Line 5	If no block substructure is specified then return the *Entity-dict* contribution for the block
Line 8	Decompose the block substructure
Line 9	Construct a qualifier which denotes the level of the block substructure.
Line 10	Update *entitydict* to include the $Data\text{-}type\text{-}definition_1$ defined in the block substructure
Line 11	Return this updated *entitydict* joined with the contributions from the blocks (*bdefs*), signals (*sdefs*) and syntypes (*syndefs*).

$is\text{-}wf\text{-}decision\text{-}answers(graph, level)(dict) \triangleq$ **(5.1.7)**

1 $(\mathbf{let}\ (starttrans, stateset) =$
2 $\quad \mathbf{cases}\ graph:$
3 $\quad\ (\mathrm{mk}\text{-}Procedure\text{-}graph_1(init, stset) \rightarrow (init, stset),$
4 $\quad\ \ \mathrm{mk}\text{-}Process\text{-}graph_1(init, stset) \quad \rightarrow (init, stset))\ \mathbf{in}$
5 $\mathbf{let}\ trans = \mathrm{s}\text{-}Transition_1(starttrans)\ \mathbf{in}$
6 $is\text{-}wf\text{-}transition\text{-}answers(trans, level)(dict) \wedge$
7 $(\forall \mathrm{mk}\text{-}State\text{-}node_1(,, inputs) \in stateset)$
8 $\ ((\forall input \in inputs)(is\text{-}wf\text{-}transition\text{-}answers(\mathrm{s}\text{-}Transition_1(input), level)(dict))))$

type : $(Procedure\text{-}graph_1 \mid Process\text{-}graph_1)\ Qualifier_1 \rightarrow Entity\text{-}dict \rightarrow Bool$

Objective Check that the answers in a decision action of a procedure or process graph are mutual exclusive

Parameters

graph	The procedure or process graph
level	The $Qualifier_1$ denoting the procedure or process

Result True if success

Algorithm

Line 1-4	Let *starttrans* denote the start node of the graph and let *stateset* denote the states of the graph
Line 5	Let *trans* denote the initial transition
Line 6	The answers in the decisions of the initial transitions must be mutual exclusive and
Line 7	For every state it must hold that every input node in the state
Line 8	The transition in the input node contains mutual exclusive answers in the decisions.

is-wf-transition-answers(**mk-***Transition*$_1$(*trans*,), *level*)(*dict*) $\triangleq$ **(5.1.8)**

```
1  (∀mk-Decision-node1(, answerset, elsetrans) ∈ elems trans)
2   ((elsetrans ≠ nil ⊃ is-wf-transition-answers(elsetrans, level)(dict)) ∧
3    (∀mk-Decision-answer1(answer1, trans1), mk-Decision-answer1(answer2, trans2) ∈ answerset)
4     (is-wf-transition-answers(trans1, level)(dict) ∧
5      is-wf-transition-answers(trans2, level)(dict) ∧
6      (answer1 ≠ answer2 ∧ is-Range-condition1(answer1) ∧ is-Range-condition1(answer2) ⊃
7       (let mk-Range-condition1(orid, cset1) = answer1,
8            mk-Range-condition1(, cset2) = answer2 in
9        (∀term ∈ Ground-expression1)
10        ((let dict' = dict + [SCOPEUNIT ↦ level] in
11          trap exit with true in
12          let answerterm1 = eval-ground-expression(make-valuetest-operator(term, orid, cset1))(dict'),
13              answerterm2 = eval-ground-expression(make-valuetest-operator(term, orid, cset2))(dict') in
14          ¬is-equivalent(answerterm1, answerterm2, level)(dict))))))
```

type : *Transition*$_1$ *Qualifier*$_1$ $\rightarrow$ *Entity-dict* $\rightarrow$ *Bool*

Objective Check that every decision action in a transition contains mutual exclusive answers

Parameters

trans	The actions in the transition
level	The qualifier denoting the surrounding scopeunit

Algorithm

Line 1	For every decision node in the action list, line 2-line 14 must hold
Line 2	The transition in the else part must contain mutual exclusive answers and
Line 3	For every two branches in the decision line 4-line 14 must hold
Line 4-5	The transitions in the two branches must contain mutual exclusive answers and
Line 6	If it is different branches and they both contain formal text in the answers then
Line 7-8	Let *orid* denote the *Identifier*$_1$ of the OR operator, *cset1* denote the range conditions of one of the two branch and let *cset2* denote the range conditions of the other of the two branches in hand
Line 9	For every ground term (*term*) it must hold that
Line 12-13	The ground term (*answerterm1*) derived from *term* and the first condition set (line 12) must not be in the same equivalent class as the ground term (*answerterm2*) derived from *term* and the second condition set. *make-valuetest-operator* returns a *Ground-expression* which is evaluated by *eval-ground-expression* to form a ground term. Any *exits* from *eval-ground-expression* are trapped (line 11) since range checks implied from syntypes should not be applied until the decision is interpreted.

5.2 Handling of Abstract Data Types

This section contains the functions handling abstract data types. The entry functions are :

extract-sortdict which is applied during the construction of *entitydict* and which creates the type descriptors, sort descriptors, literal descriptors and operator descriptors.

reduce-term which is used when a term is transferred to another scopeunit such as in actual parameters, assignment to a non-local variable etc.

is-equivalent which is used when two terms should be compared such as in conditional expressions, range check and decision nodes.

extract-sortdict(*typedef*, *l*)(*dict*) $\triangleq$ (5.2.1)

```
 1  (let mk-Data-type-definition_1(tnm, union, sorts, signatureset, eqs) = typedef in
 2   let tid = mk-Identifier_1(l, tnm) in
 3   let (psmap, eqs') =
 4       if union = {} then
 5         ([], {})
 6        else
 7         (let tid' ∈ union in
 8          let mk-TypeDD(pmap, equa) = dict((tid', TYPE)) in
 9          (pmap, equa)) in
10   let literald =
11       [mk-Identifier_1(l, s-Literal-operator-name_1(lit)) ↦ mk-OperatorDD(⟨⟩, s-Result_1(lit)) |
12        lit ∈ signatureset ∧ is-Literal-signature_1(lit)],
13       operatord =
14       [mk-Identifier_1(l, s-Operator-name_1(op)) ↦ mk-OperatorDD(s-Argument-list_1(op), s-Result_1(op)) |
15        op ∈ signatureset ∧ is-Operator-signature_1(op)] in
16   let dict' = dict + [(id, VALUE) ↦ literald(id) | id ∈ dom literald] +
17       [(id, VALUE) ↦ operatord(id) | id ∈ dom operatord] in
18   let sortd = [(id, SORT) ↦ mk-SortDD(tid) | id ∈ sorts] in
19   let sortset = {mk-Identifier_1(l, nm) | nm ∈ (sorts ∪ dom psmap)},
20       sortmap = [sort ↦ make-equivalent-classes(sort)(dict') | sort ∈ sortset] in
21   let equations = eqs ∪ eqs' in
22   let sortmap' = eval-equations(sortmap, equations)(dict) in
23   let dict'' = dict' + sortd + [(tid, TYPE) ↦ mk-TypeDD(sortmap', equations)] in
24   if (∃{mk-Ground-term_1(t), mk-Error-term_1()} ⊂ union rng sortmap')(is-Identifier_1(t)) then
25     exit("§5.4.1.7: Literal is equal to the error term")
26    else
27     if is-wf-values(psmap, sortmap')(dict'') then
28       dict''
29      else
30       exit("Z.100 §5.2.1: Generation or reduction of equivalent classes of the enclosing scopeunit"))
```

type : *Data-type-definition*$_1$ *Qualifier*$_1$ $\rightarrow$ *Entity-dict* $\rightarrow$ *Entity-dict*

Objective Update *entitydict* to contain the descriptor for a *Data-type-definition*$_1$ and for its contained sorts, operators and literals.

Parameters

typedef A *Data-type-definition*$_1$

l The level on which it is defined.

Result The updated *entitydict*

Algorithm

Line 2 Construct the *Identifier*$_1$ of the data type.

Line 3-9 Extract the *Sortmap* and the *Equations*$_1$ of the parent (enclosing) data type definition. If no *Type-identifier*$_1$ is present in *Type-union*$_1$ then it is the system level, otherwise the parent *Sortmap* and *Equations*$_1$ are found in the descriptor of the parent.

Line 10-12 — Construct the descriptors for all the literals defined in the data type definition. They are considered as operators without any arguments.

Line 13-16 — Construct the descriptors for all the operators defined in the data type definition and add them to the existing *Entity-dict* and give them the entity class VALUE.

Line 18 — Construct the descriptors (*sortmap*) for all the sorts defined in the data type definition.

Line 19-20 — Construct the initial *Sortmap* consisting of as many equivalent classes for each *sort* as there are terms for the *sort*, i.e. each equivalent class contains one and only one term. The domain of *sortmap* is the locally defined sorts (*sorts*) and the sorts of the enclosing scopeunit (**dom** *psmap*).

Line 22 — Modify *Sortmap* according to the equations.

Line 24-25 — Ground terms which are literal identifiers and the error term must not belong to the same equivalent class

Line 23 — Update *entitydict* with descriptors for the local sorts and for the data type definition

Line 27 — If the data type is consistent with the data type of the enclosing scopeunit (i.e. no values are changed) then return the updated *entitydict*

is-equivalent(*lterm*, *rterm*, *level*)(*dict*) $\triangleq$ (5.2.2)

```
1  (let (id, TYPE) ∈ dom dict be s.t. s-Qualifier1(id) = level in
2   let mk-TypeDD(sortmap, ) = dict((id, TYPE)) in
3   let termsets = union rng sortmap in
4   let ltermset ∈ termsets be s.t. lterm ∈ ltermset,
5       rtermset ∈ termsets be s.t. rterm ∈ rtermset in
6   if mk-Error-term1() ∈ (ltermset ∪ rtermset)
7     then exit("§5.4.1.7: Operator application is equivalent to the error term")
8     else ltermset = rtermset)
```

type: *Ground-term*$_1$ *Ground-term*$_1$ *Qualifier*$_1$ $\rightarrow$ *Entity-dict* $\rightarrow$ *Bool*

Objective — Test whether two terms belongs to the same equivalent class.

Parameters

lterm,rterm — The two terms which have to be tested for equivalence

level — A qualifier denoting the scopeunit in which context the test should be performed.

Result — True if they are equivalent.

Algorithm

Line 1 — Extract the *Type-identifier*$_1$ for the scopeunit denoted by *level*

Line 2-3 — Construct the set of all equivalent classes for all sorts visible in the scopeunit. (All equivalent classes are disjoint in the entire system)

Line 4-5 — Extract the equivalent class for *lterm* and the equivalent class for *rterm*

Line 6 — None of these equivalent classes may include the error term.

Line 8 — Return true if *lterm* and *rterm* belong to the same equivalent class.

reduce-term(*sortid*, *term*, *level*)(*dict*) $\triangleq$ **(5.2.3)**

```
1   if term = nil then
2     nil
3    else
4    (let sortid' = if is-SortDD(dict((sortid, SORT))) then
5                       sortid
6                      else
7                      s-Parent-sort-identifier1(dict((sortid, SORT))) in
8      let mk-SortDD(tpid) = dict((sortid', SORT)) in
9      let mk-TypeDD(sortmap, ) = dict((tpid, TYPE)) in
10     let (tpid', q) ∈ dom dict be s.t. q = TYPE ∧ s-Qualifier1(tpid') = level in
11     let mk-TypeDD(sortmap', ) = dict((tpid', q)) in
12     let vset ∈ sortmap'(sortid') be s.t. term ∈ vset in
13     let vset' ∈ sortmap(sortid') be s.t. vset' ⊆ vset in
14     let term' ∈ vset' in
15     term')
```

type : *Sort-reference-identifier*$_1$ [*Ground-term*$_1$] *Qualifier*$_1$ $\rightarrow$ *Entity-dict* $\rightarrow$ [*Ground-term*$_1$]

Objective

Convert a term to another term of the same equivalent class such that the chosen term only contains literals and operators which are defined in the scopeunit defining *sortid*. This conversion is required every time a value is transferred to a non-enclosed scopeunit. (For simplicity, the conversion is made every time a value (may) be transferred to another scopeunit, i.e. in assignment, evaluation of actual parameters etc.)

Parameters

sortid	The *Sort-reference-identifier*$_1$ denoting the sort of the term to be converted
term	The *Term*$_1$ to be converted
level	The scopeunit in which *term* is used.

Result

The new term. The result is **nil** if *term* is **nil**(*term* is nil if *reduce-term* is used in an actual parameter evaluating function where the actual parameter is unspecified)

Algorithm

Line 1	If no *Term*$_1$ is specified then return **nil**.
Line 4-7	If *sortid* denotes a *Syntype-identifier*$_1$ (*SyntypeDD*) then extract the parent *Sort-identifier*$_1$.
Line 8	Extract the *Type-identifier*$_1$ (*tpid*) defining the sort.
Line 9	Extract the *Sortmap* (*sortmap*) of the type.
Line 10-11	Extract the *Sortmap* (*sortmap'*) of the type defined in the scopeunit where *term* is used.
Line 12	Extract the equivalent class containing *term* among the equivalent classes belonging to the type defined in the scopeunit where *term* is used.
Line 13	Extract the equivalent class of the defining scopeunit for the sort (*sortid'*) which is contained in the other equivalent class.
Line 14	Return an arbitrary element from this equivalent class.

is-wf-values(survmap, vmap)(dict) ≜ (5.2.4)

```
1 if survmap = [] then
2   is-wf-bool-and-pid(dict, vmap(dict(PIDSORT)))
3   else
4   (∀id ∈ dom survmap)
5    ((let survset = survmap(id),
6          vset = vmap(id) in
7      (∀class ∈ vset)((∃!class' ∈ survset)(class' ⊆ class))))
```

type: *Sortmap Sortmap → Entity-dict → Bool*

Objective Test whether the set of equivalent classes for the enclosing scopeunit is the same set of as for the enclosed scopeunit.

Parameters

survmap	The *Sortmap* denoting the values of the enclosing scopeunit
vmap	The *Sortmap* denoting the values of the enclosed scopeunit

Result True if success.

Algorithm

Line 1-2	If the *Sortmap* of the enclosing scopeunit is empty, it is the system level and it must hold that the Boolean True and False belongs to different equivalent classes and that there exist an infinite number of PiD values. In nested scopeunit these checks are just a special case of the more general condition checked by *is-wf-values.*
Line 4-7	For all *Sort-identifier*$_1$s belonging to the enclosing scopeunit the condition must hold.
Line 5	Extract the equivalent classes applying in the enclosing scopeunit for the sort in hand
Line 6	Extract the equivalent classes applying in the enclosed scopeunit for the sort in hand
Line 7	For every equivalent class in the enclosed scopeunit, it must hold that it includes all the terms of a unique equivalent class of the enclosing scopeunit

make-equivalent-classes(sort)(dict) ≜ (5.2.5)

```
1 (let termset = {term ∈ Ground-term1 | is-of-this-sort(sort, term)(dict)} in
2  let classes = {{term} | term ∈ termset} ∪ {{mk-Error-term1()}} in
3  [sort ↦ classes])
```

type: *Sort-identifier*$_1$ *→ Entity-dict → Sortmap*

Objective Construct all possible *Term*$_1$s for a *Sort-identifier*$_1$ denoted by *sort* and construct the *Sortmap* entry for *sort* where the *Term*$_1$s are put into different equivalent classes.

Result The *Sortmap* contribution for *sort*

Algorithm

Line 1-	Construct a set of *Ground-Term*$_1$s consisting of all possible terms for the sort
Line 2	Put all the terms in the set into different equivalent-classes and put the error term in its own equivalent class.

is-of-this-sort(*sort*, *t*)(*dict*) $\triangleq$ **(5.2.6)**

```
1  (let mk-Ground-term1(term) = t in
2   if is-Identifier1(term) then
3     (let entry = (term, VALUE) in
4      entry ∈ dom dict ∧
5      is-OperatorDD(dict(entry)) ∧
6      s-Argument-list(dict(entry)) = ⟨⟩ ∧
7      s-Result(dict(entry)) = sort)
8    else
9     if is-Conditional-term1(term) then
10      false
11     else
12      (let (id, arglist) = term in
13       let entry = (id, VALUE) in
14       if entry ∈ dom dict ∧ is-OperatorDD(dict(entry)) then
15         (let mk-OperatorDD(sortlist, result) = dict(entry) in
16          len arglist = len sortlist ∧
17          result = sort ∧
18          (∀i ∈ ind arglist)(is-of-this-sort(sortlist[i], arglist[i])(dict)))
19       else
20         false))
```

type : $\mathit{Sort\text{-}identifier}_1\ \mathit{Ground\text{-}term}_1 \rightarrow \mathit{Entity\text{-}dict} \rightarrow \mathit{Bool}$

Objective Test whether a given Term_1 t is a term of the sort denoted by *sort*

Algorithm

Line 2	If the term is an identifier then
Line 4	The identifier must be found in *entitydict*
Line 5-6	as a literal (i.e. as an operator without arguments)
Line 7	and the result sort must be equal to *sort*
Line 9	If the term is a conditional term then it does not represent a value (but the consequence and alternative in the conditional term does)
Line 12	If the term is an operator term then
Line 14	If the identifier in the term can be found in *entitydict* and it denotes an operator then
Line 16	The number of arguments in the descriptor must be equal to the number of arguments present in the term
Line 17	and the result sort found in the descriptor must be equal to *sort*
Line 18	and each argument term found in *term* must be of the appropriate sort according to the argument sortlist found in the descriptor.

eval-equations(*sortmap*, *equations*)(*dict*) $\triangleq$ (5.2.7)

```
1   (let trueterm = dict(TRUEVALUE),
2        falseterm = dict(FALSEVALUE) in
3    let quanteq = {eq ∈ equations | is-Quantified-equations_1(eq)} in
4    let 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-equation_1(eq)
9                   then expand-conditional-term-in-conditions({eq}, trueterm, falseterm)
10                  else {eq} | eq ∈ rest'} in
11   let unquanteqs = {eq ∈ rest'' | is-Unquantified-equation_1(eq)},
12       condeqs = {eq ∈ rest'' | is-Conditional-equation_1(eq)} in
13   let sortmap' = eval-unquantified-equations(sortmap, unquanteqs) in
14   eval-conditional-equations(sortmap', condeqs))
```

type : *Sortmap* $Equations_1$ $\rightarrow$ *Entity-dict* $\rightarrow$ *Sortmap*

Objective Reduce the number of equivalent classes for the sorts visible in a given scopeunit according to a set of equations.

Parameters

sortmap A *Sortmap* containing the equivalent 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-unquantified-equations(*sortmap*, *equations*) $\triangleq$ (5.2.8)

```
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* $Equations_1 \rightarrow$ *Sortmap*

Objective Modify *sortmap* (the equivalent 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 *Terms* 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 equivalent class which contains *lterm*.
Line 8	Extract the equivalent class which contains *rterm*.
Line 9	If the terms denote the same equivalent class then do not update *sortmap* else
Line 12	Define a new set of equivalent classes wherein the two equivalent classes has been unified.
Line 13	Modify *sortmap* to contain the new set of equivalent classes
Line 14	Reduce the number of equivalent classes by using the information obtained by the equation
Line 15	Repeat the operation for the rest of the equations.

eval-deduced-equivalence(sortmap) $\triangleq$ (5.2.9)

```
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* $\rightarrow$ *Sortmap*

Objective Reduce the number of the equivalent classes for sorts by using the information that two terms of a sort are in the same equivalent class.

Parameters

sortmap A *Sortmap* containing the equivalent classes which are to be modified

Result The *Sortmap* where the number of equivalent classes for some of the sorts has been reduced

Algorithm

Line 1 If there exist three equivalent classes *class1, class2, class3* in the *Sortmap* such that *class1* and *class2* are disjoint (*class3* may be equal to *class1* or *class2* or it may denote another equivalent class, even of another sort) and there exist two terms (*term1* and *term2*) in *class3* such that when replacing *term1* by *term2* in a term (*term*) taken from *class1*, a term in *class2* is obtained then

Line 4-13 *class1* and *class2* is merged into one equivalent class

Line 4-6 Let *class1*, *class2*, *class3* denote three such equivalent classes

Line 7 Let *sort* denote the sort of *class1* and *class2*. *class1* and *class2* 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 equivalent classes for the sort have been merged

Line 11 Repeat the operation (with the modified *sortmap*) until no more equivalent classes can be merged *replace-term*(*term*, *oldterm*, *newterm*) $\triangleq$ **(5.2.10)**

```
1  if term = oldterm then
2    newterm
3   else
4    if is-Identifier1(term) then
5      term
6     else
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        (opid, arglist'))
14       else
15        term)
```

type : *Ground-term*$_1$ *Ground-term*$_1$ *Ground-term*$_1$ $\rightarrow$ *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 4	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 equivalent class). Let *op* denote the operator identifier and let *arglist* denote the argument list
Line 8	If there exist 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-quantified-equation(*sortmap*, *quanteqs*) $\triangleq$ **(5.2.11)**

```
(let mk-Quantified-equations1(nmset, sortid, equations) = quanteqs in
 let nm ∈ nmset in
 let mk-Identifier1(level, snm) = sortid in
 let valueid = mk-Identifier1(level ⌢ ⟨mk-Sort-qualifier1(snm)⟩, nm) in
 let allterms = union sortmap(sortid) \ {mk-Error-term1()} in
 let equations' = union {union {insert-term(sortmap, eq, valueid, term) | term ∈ allterms} |
                              eq ∈ equations} in
 if nmset = {nm} then
   equations'
  else
   (let quanteq = mk-Quantified-equations1(nmset \ {nm}, sortid, equations') in
    eval-quantified-equation(sortmap, quanteq)))
```

type : *Sortmap* *Quantified-equations*$_1$ $\rightarrow$ *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 equivalent 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 *Error-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*) $\triangleq$ **(5.2.12)**

```
1   cases equation:
2   (mk-Unquantified-equation1(term1, term2)
3       → {mk-Unquantified-equation1(insert-term-in-term(term1, vid, term),
4                                    insert-term-in-term(term2, vid, term))},
5    mk-Quantified-equations1(, , )
6       → (let equations = eval-quantified-equation(sortmap, equation) in
7          union{insert-term(sortmap, eq, vid, term) | eq ∈ equations}),
8    mk-Conditional-equation1(eqs, eq)
9       → (let mk-Unquantified-equation1(term1, term2) = eq,
10             eqs' = union{insert-term(sortmap, e, vid, term) | e ∈ eqs} in
11         let eq' = mk-Unquantified-equation1(insert-term-in-term(term1, vid, term),
12                                             insert-term-in-term(term2, vid, term)) in
13         {mk-Conditional-equation1(eqs', eq')}),
14   T → {equation})
```

type : *Sortmap Equation*$_1$ *Value-identifier*$_1$ *Ground-term*$_1$ $\rightarrow$ *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*$_1$ which *vid* should be replaced by.

Result A set of equations containing the modified equation. If the equation is a 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 (*term1*,*term2*).

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*) $\triangleq$ (5.2.13)

```
 1  (if is-Ground-term_1(term) ∨ is-Error-term_1(term) then
 2     term
 3    else
 4     (let mk-Composite-term_1(term') = term in
 5      if is-Identifier_1(term') then
 6        if term' = vid then vterm else term
 7       else
 8        if is-Conditional-term_1(term') then
 9          (let mk-Conditional-term_1(cond, t1, t2) = term' in
10           let cond' = insert-term-in-term(cond, vid, vterm),
11               t1' = insert-term-in-term(t1, vid, vterm),
12               t2' = insert-term-in-term(t2, vid, vterm) in
13           let term'' = mk-Conditional-term_1(cond', t1', t2') in
14           if is-Ground-term_1(cond') ∧ is-Ground-term_1(t1') ∧ is-Ground-term_1(t2') then
15             mk-Ground-term_1(term'')
16            else
17             mk-Composite-term_1(term''))
18         else
19          (let (opid, arglist) = term' in
20           let arglist' =
21               ⟨insert-term-in-term(arglist[i], vid, vterm) | 1 ≤ i ≤ len arglist⟩ in
22           if (∃arg ∈ elems arglist)(is-Composite-term_1(arg)) then
23             mk-Composite-term_1((opid, arglist'))
24            else
25             mk-Ground-term_1((opid, arglist')))))
```

type : $Term_1$ $Value\text{-}identifier_1$ $Ground\text{-}term_1 \rightarrow Term_1$

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 8-13	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 14-17	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 19-25	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*) $\triangleq$ **(5.2.14)**

```
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, eqs)
10               → (condeq, eqs)) 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*$_1$ *Literal-operator-identifier*$_1$ *Literal-operator-identifier*$_1$ $\rightarrow$ *Equations*$_1$

Objective Replace every *Conditional-term*$_1$ by two *Conditional-equation*$_1$s.

example :

```
the equation

if a then b else c == 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 which does not contain any *Conditional-terms*

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. *cond1* and *cond2* are the conditions to be tested upon. A condition is **nil** if the term do not contain any conditional terms. *t1',t2'* are the original terms (*t1,t2*) wherein a conditional term has been replaced by the "then" part of the conditional term and *t1'',t2''* 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 *eq1* respective *eq2* has been added as an extra condition. (*condeq1*) contains an equation wherein one of the original terms (*t1* or *t2*) has been replaced by a term containing the "then" part and (*condeq2*) 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-in-terms(t) $\triangleq$ **(5.2.15)**

```
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 corresponding to the "then" part (*arglist′*) and to 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.

expand-conditional-term-in-conditions(*equations*, *trueterm*, *falseterm*) $\triangleq$ **(5.2.16)**

```
(if equations = {} then
   {}
  else
   (let eq ∈ equations in
    let mk-Conditional-equation_1(condset, eq') = eq in
    if (∃cond ∈ condset)
       ((let mk-Unquantified-equation_1(t1, t2) = cond in
         let (, ,cond1) =
              expand-conditional-in-terms(t1),
              (, ,cond2) =
              expand-conditional-in-terms(t2) in
         cond1 ≠ nil ∨ cond2 ≠ nil)) then
      (let (condeq, cond, term, nterm1, nterm2) be s.t. condeq ∈ condset ∧
           (let mk-Unquantified-equation_1(t1, t2) =
                condeq in
            let (t1', t1'', cond1) =
                expand-conditional-in-terms(t1),
                (t2', t2'', cond2) =
                expand-conditional-in-terms(t2) in
            (cond, term, nterm1, nterm2) = (if cond1 = nil
                                             then (cond2, t1, t2', t2'')
                                             else (cond1, t2, t1', t1''))) in
       let eq1 = mk-Unquantified-equation_1(cond, trueterm),
           eq2 = mk-Unquantified-equation_1(cond, falseterm) in
       let condset' = condset \ {condeq} ∪ {eq1, mk-Unquantified-equation_1(term, nterm1)},
           condset'' = condset \ {condeq} ∪ {eq2, mk-Unquantified-equation_1(term, nterm2)} in
       let equations' = equations \ {eq} ∪ {mk-Conditional-equation_1(condset', eq'),
                                            mk-Conditional-equation_1(condset'', eq')} in
       expand-conditional-term-in-conditions(equations', trueterm, falseterm))
     else
      {eq} ∪ expand-conditional-term-in-conditions(equations \ {eq}, trueterm, falseterm)))
```

type : *Conditional-equation*$_1$ *Literal-operator-identifier*$_1$ *Literal-operator-identifier*$_1$ $\rightarrow$ *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 == 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 (*nterm1*), the "else" version of the term in the unquantified equation containing the conditional term (*nterm2*) 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.

$\mathit{eval\text{-}conditional\text{-}equations}(\mathit{sortmap}, \mathit{condequations}) \triangleq$ **(5.2.17)**

1 $\textbf{if}\ (\exists \mathit{condeq} \in \mathit{condequations})(\mathit{restriction\text{-}holds}(\mathit{condeq}, \mathit{sortmap}))\ \textbf{then}$
2 $\quad(\textbf{let}\ \mathit{condeq} \in \mathit{condequations}\ \textbf{be s.t.}\ \mathit{restriction\text{-}holds}(\mathit{condeq}, \mathit{sortmap})\ \textbf{in}$
3 $\quad\ \textbf{let}\ \textbf{mk-}\mathit{Conditional\text{-}equation}_1(, \mathit{eq}) = \mathit{condeq}\ \textbf{in}$
4 $\quad\ \textbf{let}\ \mathit{sortmap}' = \mathit{eval\text{-}unquantified\text{-}equations}(\mathit{sortmap}, \{\mathit{eq}\})\ \textbf{in}$
5 $\quad\ \mathit{eval\text{-}conditional\text{-}equations}(\mathit{sortmap}', \mathit{condequations} \setminus \{\mathit{condeq}\}))$
6 $\ \textbf{else}$
7 $\quad \mathit{sortmap}$

type: $\mathit{Sortmap}\ \mathit{Conditional\text{-}equation}_1\textbf{-set} \rightarrow \mathit{Sortmap}$

Objective Reduce the number of equivalent 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 exist 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 holds

$\mathit{restriction\text{-}holds}(\textbf{mk-}\mathit{Conditional\text{-}equation}_1(\mathit{eqs},), \mathit{sortmap}) \triangleq$ **(5.2.18)**

1 $(\textbf{let}\ \mathit{termpairs} = \{\{\mathit{term1}, \mathit{term2}\} \mid (\exists \mathit{eq} \in \mathit{eqs})(\textbf{mk-}\mathit{Unquantified\text{-}equation}_1(\mathit{term1}, \mathit{term2}) = \mathit{eq})\}\ \textbf{in}$
2 $\ (\forall \mathit{pairs} \in \mathit{termpairs})((\exists \mathit{class} \in \textbf{union rng}\ \mathit{sortmap})(\mathit{pairs} \subset \mathit{class})))$

type: $\mathit{Conditional\text{-}equation}_1\ \mathit{Sortmap} \rightarrow \mathit{Bool}$

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 equivalent class as the left-hand side term.

$is\text{-}wf\text{-}bool\text{-}and\text{-}pid(dict, pidvalueset) \triangleq$ (5.2.19)

1 $(\mathbf{let}\ trueterm = dict(\mathsf{TRUEVALUE}),$
2 $\quad falseterm = dict(\mathsf{FALSEVALUE}),$
3 $\quad \mathbf{mk}\text{-}Identifier_1(level,) = trueterm\ \mathbf{in}$
4 $(\forall s \in Sortmap)(s \subset pidvalueset \supset (\exists n \in N_1)(n > \mathbf{card}\ s)) \wedge$
5 $\neg is\text{-}equivalent(trueterm, falseterm, level)(dict))$

type : $Entity\text{-}dict\ Term\text{-}class\text{-}\mathbf{set} \rightarrow Bool$

Objective Test whether the boolean True belongs to the same equivalent class as the boolean False and whether there exist an infinite number of PiD values

Result True if those conditions are satisfied.

Algorithm

Line 1-2	Construct the ground terms corresponding to the literals True and False
Line 3	Let *level* denote its qualifier.
Line 4	The set of PiD values must be infinite, i.e. for each (finite) subset s of *pidvalueset* there must exist an N_1 value n such that n is greater that the cardinality of the subset.
line 5	The boolean literal True must not belong to the same equivalent class as the boolean literal false.

5.3 Selection of Consistent Subset

select-consistent-subset(*bset*, *subset*, *level*) $\triangleq$ (5.3.1)

```
1  if bset = {} then
2    {}
3    else
4    (let block ∈ bset in
5     let rest = select-consistent-subset(bset \ {block}, subset, level) in
6     let mk-Block-definition1(bnm, pdefs, , , , , , sub) = block in
7     let pset = {mk-Identifier1(level ⌢ ⟨mk-Block-qualifier1(bnm)⟩, s-Process-name1(pdef)) |
8                 pdef ∈ pdefs} in
9     let bid = mk-Identifier1(level, bnm) in
10    if bid ∉ subset then
11      exit("§3.2.1: Sub-block is not in consistent subset")
12     else
13     if sub = nil then
14       rest ∪ pset
15      else
16      (let mk-Block-substructure-definition1(subnm, bset', , , , , ) = sub in
17       let level' =
18            level ⌢ ⟨mk-Block-qualifier1(bnm), mk-Block-substructure-qualifier1(subnm)⟩ in
19      if mk-Identifier1(level, subnm) ∈ subset then
20        rest ∪ select-consistent-subset(bset', subset, level')
21       else
22        if pset = {} then
23          exit("§3.2.1: Leaf block contains no processes")
24         else
25          rest ∪ pset))
```

type : *Block-definition*$_1$**-set** *Block-identifier*$_1$**-set** *Qualifier*$_1$ $\rightarrow$ *Process-identifier*$_1$**-set**

Objective Check that the given set of block identifiers and block substructure identifiers denotes a consistent subset and return the identifiers of the processes contained in the consistent subset. The function traverse recursively through the system definition

Parameters

bset	The set of block definitions for the system definition or for a block substructure definition
subset	The (assumed) consistent subset represented by a set of block identifiers and block substructure identifiers.
level	The *Qualifier*$_1$ of the scopeunit which contains the blocks *bset*

Algorithm

Line 1	When through, return the empty set of process identifiers
Line 4	Let *block* denote the next block definition to be considered
Line 5	Select consistent subset for the rest of the block definitions
Line 6	Let *bnm* denote the block name, let *pdefs* denote the set of process definitions and let *sub* denote the optional block substructure definition
Line 7	Let *pset* denote the set of *Process-Identifier*$_1$s corresponding to *pdefs*.
Line 9-11	The block (or sub-block) must be in the consistent subset
Line 13	If no substructure is present in the block then
line 14	The processes in the block is in the consistent subset
Line 16	If a substructure is specified then let *subnm* denote its name and let *bset'* denote its block definitions

Line 17-20	If the substructure is in the consistent subset then consider the blocks in the substructure else
Line 22-23	There must exist at least one process definition in the block

5.4 Construction of Communication Paths

This section contains the functions which updates every process descriptor (*ProcessDD*) to include a set of *Reachabilitys*

In every scope unit which contains channels between two blocks, the incoming paths for the channels are constructed, the outgoing paths are constructed, the paths are joined and finally the process descriptors associated to the processes contained in the block from the outgoing paths are updated. The incoming paths and outgoing paths (partial paths) contain, before they are joined, a channel at one of the endpoints and a signal route at the other endpoint. The intermediate identifiers are all sub-channel identifiers. *make-structure-paths* is the entry function which is applied in *extract-dict.*

make-structure-paths(*bset*, *cset*, *level*)(*dict*) $\triangleq$ (5.4.1)

```
1   (if cset = {} then
2      dict
3     else
4      (let ch ∈ cset in
5       let mk-Channel-definition₁(nm, mk-Channel-path₁(b1, b2, ), ) = ch in
6       if (b1 = ENVIRONMENT ∨ b2 = ENVIRONMENT) ∧ ¬is-System-qualifier₁(level[len level]) then
7          make-structure-paths(bset, cset \ {ch}, level)(dict)
8         else
9          (let chid = mk-Identifier₁(level, nm) in
10          let (reachset1, dict') = out-going-paths(chid, b1, bset, ⟨⟩)(dict) in
11          let (reachset1', dict'') = out-going-paths(chid, b2, bset, ⟨⟩)(dict') in
12          let reachset2 = in-coming-paths(chid, b2, bset, ⟨⟩)(dict) in
13          let reachset2' = in-coming-paths(chid, b1, bset, ⟨⟩)(dict) in
14          if is-consistent-refinement(reachset1, reachset2) ∧
15            is-consistent-refinement(reachset2, reachset2') then
16             (let d = update-processd(reachset1, reachset2)(dict'') in
17              let d' = update-processd(reachset1', reachset2')(d) in
18              make-structure-paths(bset, cset \ {ch}, level)(d'))
19            else
20             exit("Z.100 §3.3 : Illegal refinement of channel"))))
```

type : *Block-definition*$_1$**-set** *Channel-definition*$_1$**-set** *Qualifier*$_1$ $\rightarrow$ *Entity-dict* $\rightarrow$ *Entity-dict*

Objective For all channels in a scopeunit which are connected to two blocks or are connected to the system environment, update the *Reachabilities* for the processes which are able to send signals via the channels.

Parameters

bset	The block definitions
cset	The channel definitions for a scopeunit
level	A qualifier denoting the scopeunit.

Result The *entitydict* wherein the appropriate *ProcessDD* descriptors have been updated.

Algorithm

Line 1	When through then return the updated *entitydict*

Line 4-5	Take a channel definition from the remaining set of definitions
Line 6-7	If the channel is a sub-channel then do nothing, since sub-channels are handled by *in-coming-path* and *outgoing-path.*
Line 10-11	Extract the *Reachabilities* containing those processes which are capable of sending via the channel and containing the appropriate *Path* and the *entitydict* updated with information of *Reachabilities* corresponding to local communication paths in *b1* (line 10) respectively *b2* (line 11).
Line 12-13	Extract the *Reachabilities* containing those process in the block *b2* respectively *b1* which are capable of receiving via the channel and containing the appropriate *Path.*
Line 14	For both directions, any refinement subset selections must be consistent.
Line 16	Update the process descriptors in *reachset1* respectively *reachset1′* with *Reachabilities* containing of the possible receivers which are deduced from *reachset2* respectively *reachset2′*.
Line 18	Do the same for the rest of the channel definitions.

$is\text{-}consistent\text{-}refinement(reachset1, reachset2) \triangleq$ (5.4.2)

1 $(\mathbf{let}\ sigset1 = \{sig \mid (\exists(, sset,) \in reachset1)(sig \in sset)\},$

2 $\quad sigset2 = \{sig \mid (\exists(, sset,) \in reachset2)(sig \in sset)\}\ \mathbf{in}$

3 $\mathbf{let}\ env1 = \mathbf{card}\ reachset1 = 1 \wedge (\exists(p, ,) \in reachset1)(p = \mathsf{ENVIRONMENT}),$

4 $\quad env2 = \mathbf{card}\ reachset2 = 1 \wedge (\exists(p, ,) \in reachset2)(p = \mathsf{ENVIRONMENT})\ \mathbf{in}$

5 $\neg(\exists \mathbf{mk}\text{-}Identifier_1(qual1,), \mathbf{mk}\text{-}Identifier_1(qual2, nm2) \in sigset1)$

6 $\quad(\mathbf{len}\ qual1 > \mathbf{len}\ qual2 \wedge qual2 \frown \langle \mathbf{mk}\text{-}Signal\text{-}qualifier_1(nm2)\rangle = \langle qual1[i] \mid 1 \leq i \leq \mathbf{len}\ qual2 + 1\rangle) \wedge$

7 $\neg(\exists \mathbf{mk}\text{-}Identifier_1(qual1,), \mathbf{mk}\text{-}Identifier_1(qual2, nm2) \in sigset2)$

8 $\quad(\mathbf{len}\ qual1 > \mathbf{len}\ qual2 \wedge qual2 \frown \langle \mathbf{mk}\text{-}Signal\text{-}qualifier_1(nm2)\rangle = \langle qual1[i] \mid 1 \leq i \leq \mathbf{len}\ qual2 + 1\rangle) \wedge$

9 $(env1 \vee env2 \vee sigset1 = sigset2))$

type : *Reachabilities Reachabilities* $\rightarrow$ *Bool*

Objective Check that the signals in the signal routes of each endpoint of a channel do not include signals on different refinement levels of the same signal and check that the set of signals from the outgoing endpoint of the channel is the same as the set of signals at the incoming endpoint.

Parameters

reachset1	The *Reachabilities* for the outgoing end of the channel.
reachset2	The *Reachabilities* for the incoming end of the channel.

Result **true** if the above described conditions are satisfied.

Algorithm

Line 1	Let *sigset1* denote the set of signals in the outgoing end of the channel.
Line 2	Let *sigset2* denote the set of signals in the incoming end of the channel.
Line 3	Let *env1* be **true** if the outgoing end of the channel is the system environment.
Line 4	Let *env2* be **true** if the incoming end of the channel is the system environment.
Line 5-6	For every two outgoing signals it must hold that they must not be subsignals of each other.
Line 7-8	For every two incoming signals it must hold that they must not be subsignals of each other.
Line 9	Unless one of the endpoints is the system environment it must hold that the set of outgoing signals equals the set of incoming signals.

out-going-paths(*chid*, *b*, *bset*, *path*)(*dict*) $\triangleq$ **(5.4.3)**

```
1  if b = ENVIRONMENT then
2    ({(ENVIRONMENT, ⟨chid⟩)}, dict)
3   else
4    (let mk-Identifier1(level, bnm) = b in
5     let bdef ∈ bset be s.t. s-Block-name1(bdef) = bnm in
6     let mk-Block-definition1(,,, connects, srdefs,,, sub) = bdef in
7     let path' = path ⁀ ⟨chid⟩ in
8     let bqual = level ⁀ ⟨mk-Block-qualifier1(bnm)⟩ in
9     if (∃(mk-Identifier1(qual,), PROCESS) ∈ dom dict)(qual = bqual) then
10      (let mk-Channel-to-route-connection1(ch, routeset) ∈ connects be s.t. ch = chid in
11       let (rset, dict') = make-out-reaches(routeset, srdefs, path')(dict) in
12       (rset, dict'))
13     else
14      (let mk-Block-substructure-definition1(, bset', connects', cset,,,) = sub in
15       let mk-Channel-connection1(cid, cidset) ∈ connects' be s.t. cid = chid in
16       let dict' = make-structure-paths(bset', cset, level)(dict) in
17       make-out-connect-paths(cidset, cset, bset', path')(dict')))
```

type : *Channel-identifier*$_1$ (*Block-identifier*$_1$ | ENVIRONMENT)
Block-identifier$_1$**-set** *Path* $\rightarrow$
Entity-dict $\rightarrow$ *Reachability*-set *Entity-dict*

Objective

Construct the *Reachabilities* corresponding to signals leading out of a block via a given channel (*chid*). The channel is part of the *Paths* in the *Reachabilities.* The constructed (temporary) *Reachabilities* are different from the *Reachabilities* found in the process descriptors because the *Path* is only a partial communication path (the destination part is missing) and because the *Process-identifier*$_1$ in the *Reachabilities* is the sending process. The complementary function *incoming-path* constructs the "inverse" *Reachability* wherein the *Path* is the originating part and in the function *update-processd* the two *Reachabilities* are merged to form the *Reachabilities* which are inserted in the descriptor of the sending process. *outgoing-path* also updates the process descriptors, but only with the *Reachabilities* local to the block from which the channel originates. As several channels may originate from the same block, the process descriptors may be updated several times with the same local *Reachabilities*, but it does not matter as *Reachabilities* is a set.

Parameters

chid	The channel identifier which the function constructs *Paths* from
b	The block identifier which the channel originates from
bset	A set of block definitions among which the block can be found
path	The path constructed so far which leads out of the enclosing block (if the channel is not a sub-channel the path only contains *chid*).

Result

The (temporary) *Reachabilities* and the *entitydict* updated with the communication paths local to the block.

Algorithm

Line 1-2	If the originating endpoint is the system environment then return a *Reachability* containing ENVIRONMENT as the originating endpointand the unchanged *dict.*
Line 4-6	Extract the block definition from *bset* which correspond to the block identifier *b*.
Line 7	Add the channel identifier (*chid*) to the *Path* (which denotes the path from a channel different from a sub-channel to *chid*).

Line 8	Let *bqual* denote the qualifier of the entities defined in the block *bnm*
Line 9-12	If the processes in the block are selected then return the *Reachabilities* containing the processes (*rset*) and the *Entity-dict* updated with *Reachabilities* for the processes capable of sending to other processes in the block.
Line 14-15	Decompose the block substructure definition and the connection point to which the channel (*chid*) is connected.
Line 16	Update *Entity-dict* with *Reachabilities* containing *Paths* which are local to the block substructure. Note that the process descriptors are (harmlessly) updated with the same *Reachabilities* several times if several channels are connected to the block.
Line 17	Continue the creation of *Reachabilities* by entering the sub-blocks connected to the sub-channels *cidset*.

$\textit{make-out-connect-paths}(\textit{cidset}, \textit{cset}, \textit{bset}, \textit{path})(\textit{dict}) \triangleq$ **(5.4.4)**

1 $(\textbf{if}\ \textit{cidset} = \{\}\ \textbf{then}$
2 $\quad(\{\}, \textit{dict})$
3 $\quad\textbf{else}$
4 $\quad(\textbf{let}\ \textit{cid} \in \textit{cidset}\ \textbf{in}$
5 $\quad\ \textbf{let}\ (\textit{reachsetrest}, \textit{dictrest}) = \textit{make-out-connect-paths}(\textit{cidset} \setminus \{\textit{cid}\}, \textit{cset}, \textit{bset}, \textit{path})(\textit{dict})\ \textbf{in}$
6 $\quad\ \textbf{let}\ \textit{cdef} \in \textit{cset}\ \textbf{be s.t.}\ \textbf{s-}\textit{Channel-name}_1(\textit{cdef}) = \textbf{s-}\textit{Name}_1(\textit{cid})\ \textbf{in}$
7 $\quad\ \textbf{let mk-}\textit{Channel-definition}_1(, \textbf{mk-}\textit{Channel-path}_1(\textit{b}1, \textit{b}2,),) = \textit{cdef}\ \textbf{in}$
8 $\quad\ \textbf{let}\ \textit{block} = \textbf{if}\ \textit{b}2 = \mathsf{ENVIRONMENT}\ \textbf{then}\ \textit{b}1\ \textbf{else}\ \textit{b}2\ \textbf{in}$
9 $\quad\ \textbf{let}\ (\textit{rset}, \textit{dict}') = \textit{out-going-paths}(\textit{cid}, \textit{block}, \textit{bset}, \textit{path})(\textit{dictrest})\ \textbf{in}$
10 $\quad\ (\textit{reachsetrest} \cup \textit{rset}, \textit{dict}')))$

type : $\textit{Channel-identifier}_1\textbf{-set}\ \textit{Channel-definition}_1\textbf{-set}\ \textit{Block-definition}_1\textbf{-set}\ \textit{Path} \rightarrow$
$\textit{Entity-dict} \rightarrow \textit{Reachability}\textbf{-set}\ \textit{Entity-dict}$

Objective Construct the temporary *Reachabilities* corresponding to the signals conveyed by the sub-channels of a block substructure. The complementary function is *make-in-connect-paths*.

Parameters

cidset	The set of sub-channels
cset	The set of channel definitions for the block substructure
bset	The set of block definitions for the block substructure
path	The (partial) *Path* leading from the block substructure to the first channel different from a sub-channel.

Result As for *out-going-paths*.

Algorithm

Line 1	When through then return no *Reachabilities*, no signals and the unchanged *Entity-dict*. (The result is created recursively).
Line 4-5	Take a channel (*cid*) from the set of sub-channels and construct (recursively) *Reachabilities* for the rest of the sub-channels.
Line 6-7	Extract the channel definition corresponding to the sub-channel in hand.
Line 8	Extract the originating block of the sub-channel.
Line 9	Construct the temporary *Reachabilities* (*rset*) wherein the processes are those processes which send via the channel (*cid*) and which are contained in the block (*block*) and the *Paths* are *path* updated with the rest of the path from the channel to the signal routes connected to the processes. Furthermore, construct the

Entity-dict which is updated with *Reachabilities* local to the block (*block*).

Line 10 Return the *Reachabilities* and *Entity-dict* (as described above) for the sub-channel in hand, joined with those for all the other sub-channels.

in-coming-paths(*cid*, *block*, *bset*, *path*)(*dict*) $\triangleq$ (5.4.5)

```
 1  if block = ENVIRONMENT then
 2    {(ENVIRONMENT, ⟨cid⟩)}
 3   else
 4    (let mk-Identifier1(qual, bnm) = block in
 5     let bdef ∈ bset be s.t. s-Block-name1(bdef) = bnm in
 6     let mk-Block-definition1(,,, connects, srdefs,,, sub) = bdef in
 7     let path' = path ⌢ ⟨cid⟩ in
 8     let bqual = qual ⌢ ⟨mk-Block-qualifier1(bnm)⟩ in
 9     if (∃(mk-Identifier1(qual,), PROCESS) ∈ dom dict)(qual = bqual) then
10       (let mk-Channel-to-route-connection1(ch, routeset) ∈ connects be s.t. ch = cid in
11         make-in-reaches(routeset, srdefs, path'))
12      else
13       (let mk-Block-substructure-definition1(, bset', connects', cdefs,,,) = sub in
14         let mk-Channel-connection1(ch, cset) ∈ connects' be s.t. ch = cid in
15         make-in-connect-paths(cset, cdefs, bset', path')(dict)))
```

type : *Channel-identifier*$_1$ (*Block-identifier*$_1$ | ENVIRONMENT)
Block-identifier$_1$**-set** *Path* $\rightarrow$ *Entity-dict* $\rightarrow$ *Reachability*-**set**

Objective Construct and return *Reachabilities* wherein the processes are those which are contained in a given block. As opposed to *out-going-paths*, *in-coming-paths* construct "real" *Reachabilities* because they contain receiving processes. The *Paths* in the *Reachabilities* are partial and denotes the paths from the first channel different from a sub-channel to the receiving process. The "other end" of the path is constructed in the complementary function *out-going-paths.*

Parameters

cid	The channel for which the *Reachabilities* are constructed
block	The destination block of the channel
bset	The set of block definitions among which *block* can be found
path	The set of possible paths.

Algorithm

Line 1-2 If the destination endpoint is the system environment then return a set only containing one *Reachability* wherein the receiver is the environment.

Line 4-6 Extract and decompose the block definition corresponding to the block identifier *Block.*

Line 7 Add the channel identifier to the path which is going to be used inside the block.

Line 8 Let *bqual* denote the qualifier of the entities defined in the block *bnm*

Line 9 If there exist a descriptor of a process defined in the block then the substructure is not selected

Line 10-11 Extract and return the *Reachabilities* which correspond to the signal routes (*routeset*) connected to the channel (*ch*).

Line 13-14	Decompose the block substructure definition and the channel connection which connects the channel (*chid*) to the block substructure.
Line 15	Continue the creation of *Reachabilities* by entering the sub-blocks (*bset'*) connected to the sub-channels *cset.*

make-in-connect-paths(*cidset*, *cset*, *bset*, *path*)(*dict*) $\triangleq$ (5.4.6)

1 **if** *cidset* $= \{\}$ **then**
2 $\quad\{\}$
3 $\quad$**else**
4 $\quad$(**let** *cid* $\in$ *cidset* **in**
5 $\quad$ **let** *reachsetrest* $=$ *make-in-connect-paths*(*cidset* $\setminus \{$*cid*$\}$, *cset*, *bset*, *path*)(*dict*) **in**
6 $\quad$ **let** *cdef* $\in$ *cset* **be s.t. s-***Channel-name*$_1$(*cdef*) $=$ **s-***Name*$_1$(*cid*) **in**
7 $\quad$ **let mk-***Channel-definition*$_1$(, **mk-***Channel-path*$_1$($b1$, $b2$,),) $=$ *cdef* **in**
8 $\quad$ **let** *block* $=$ **if** $b2 =$ ENVIRONMENT **then** $b1$ **else** $b2$ **in**
9 $\quad$ **let** *inblockreach* $=$ *in-coming-paths*(*cid*, *block*, *bset*, *path*)(*dict*) **in**
10 $\quad$ *reachsetrest* $\cup$ *inblockreach*)

type : *Channel-identifier*$_1$**-set** *Channel-definition*$_1$**-set**
Block-identifier$_1$**-set** *Path* $\rightarrow$ *Entity-dict* $\rightarrow$ *Reachability***-set**

Objective Construct the *Reachabilities* corresponding to the signals conveyed by the sub-channels for a block substructure. The complementary function is *make-out-connect-paths.*

Parameters

cidset	The set of sub-channels
cset	The set of channel definitions for the block substructure
bset	The set of block definitions for the block substructure
path	The (partial) *Path* leading from the first channel different from a sub-channel to the block substructure.

Algorithm

Line 1	When through then return no *Reachabilities* (the result is created recursively).
Line 4-5	Take a sub-channel identifier from the set of sub-channels and create *Reachabilities* (recursively) for the rest of the sub-channels.
Line 6-7	Take the channel definition from *cset* which correspond to the sub-channel identifier in hand. Note that the information about which signals the channel conveys is not used, since it is the signal routes which determines which signals **actually** is conveyed by the channel.
Line 8	Extract the destination block.
Line 9	Construct the *Reachabilities* (*inblockreach*) wherein the processes are those processes which receive via the channel (*cid*) and which are contained in the block (*block*) the signals are those which are received by the processes via the channel (*cid*) and the appropriate signal routes and the *Paths* are *path* updated with the rest of the path from the channel to the signal routes connected to the processes.
Line 10	Return the *Reachabilities* (as described above) for the sub-channel in hand, joined with those for all the other sub-channels.

make-in-reaches(*routeset*, *srdefs*, *path*) ≜ **(5.4.7)**

```
1  if routeset = {} then
2    ({},{})
3   else
4    (let route ∈ routeset in
5     let reachrest = make-in-reaches(routeset \ {route}, srdefs, path) in
6     let mk-Identifier1(, rnm) = route in
7     let mk-Signal-route-definition1(nm, path1, path2) ∈ srdefs be s.t. nm = rnm in
8     let mk-Signal-route-path1(e1, e2, sigset) = path1 in
9     let (signalset, dest) =
10        if e1 = ENVIRONMENT then
11          (sigset, e2)
12         else
13          if path2 = nil then
14            ({}, e1)
15           else
16            (let mk-Signal-route-path1(, , sigset') = path2 in
17             (sigset', e1)) in
18    reachrest ∪ {(dest, signalset, path ⌢ ⟨route⟩)})
```

type : *Signal-route-identifier*$_1$**-set** *Signal-route-definition*$_1$**-set** *Path* → *Reachabilities*

Objective Construct the *Reachabilities* for a partial *Path* leading to a signal route connection point. There are as many constructed *Reachabilities* as there are signal route identifiers in the signal route connection point. The complementary function handling the outgoing signals is *make-out-reaches.*

Parameters

routeset	The set of signal route identifiers for a signal route connection
srdefs	Their corresponding signal route definitions
path	The *Paths* to which the signal routes are added.

Result The constructed *Reachabilities.*

Algorithm

Line 1	When through, return nothing (the result is created recursively).
Line 4-5	Take a *Signal-route-identifier*$_1$ from *routeset* and construct the *Reachabilities* for the rest of the signal route identifiers.
Line 6-8	Extract and decompose the signal route definition corresponding to the signal route identifier.
Line 9-17	Extract the incoming signals (*signalset*) and the destination process from the signal route definition.
Line 18	Return the *Reachability* corresponding to the signal route identifier in hand, joined with the *Reachabilities* corresponding to the rest of the signal route identifiers.

make-out-reaches(*routeset*, *routedefs*, *path*)(*dict*) $\triangleq$ (5.4.8)

```
1  (if routedefs = {} then
2     ({}, dict)
3    else
4    (let route ∈ routedefs in
5     let (restr, restd) = make-out-reaches(routeset, routedefs \ {route}, path)(dict) in
6     let mk-Signal-route-definition1(rnm, mk-Signal-route-path1(p1, p2, sset), path2) = route in
7     if p1 = ENVIRONMENT ∨ p2 = ENVIRONMENT then
8       if (∃id ∈ routeset)(s-Name1(id) = rnm) then
9         (let id ∈ routeset be s.t. s-Name1(id) = rnm in
10         if path2 = nil then
11           if p1 = ENVIRONMENT then
12             (restr, restd)
13            else
14             (restr ∪ {(p1, sset, ⟨id⟩ ⌢ path)}, restd)
15          else
16          (let mk-Signal-route-path1(, , sset') = path2 in
17           let (originp, sset'') =
18               if p1 = ENVIRONMENT then
19                 (p2, sset')
20                else
21                 (p1, sset) in
22          (restr ∪ {(originp, sset'', ⟨id⟩ ⌢ path)}, restd)))
23       else
24        (restr, restd)
25     else
26      (let mk-Identifier1(level, ) = p1 in
27       (restr, make-local-reach(mk-Identifier1(level, rnm), route)(restd)))))
```

type : *Signal-route-identifier*$_1$-**set** *Signal-route-definition*$_1$-**set** *Path* $\rightarrow$
Entity-dict $\rightarrow$ *Reachability*-**set** *Entity-dict*

Objective Construct the *Reachabilities* for a partial *Path* originating from a signal route connection point. There are as many constructed *Reachabilities* as there are signal route identifiers in the signal route connection point. The complementary function handling the incoming signals is *make-in-reaches*. Furthermore, update the process descriptors in *Entity-dict* with *Reachabilities* corresponding to the signal routes between processes. If the block has several signal route connections, the process descriptors are updated several times.

Parameters

routeset	The set of signal route identifiers for a connection point
routedefs	The set of signal route definitions for the block
path	The partial *Path* originating from the connection point.

Result The constructed *Reachabilities* and the *Entity-dict* updated with *Reachabilities* corresponding to the signal routes between processes.

Algorithm

Line 1	Every signal route definition in the block is considered. When through, return no *Reachabilities* and the unchanged *Entity-dict*.
Line 4	Take a signal route definition from *routeset* and construct the *Reachabilities* (*restr*) and the updated *Entity-dict* (*restd*) for the rest of the signal route definitions.
Line 7-24	Cover the case where the signal route is connected to a channel.

Line 8 and line 24 If the signal route is not mentioned in this connection point (represented by *routeset*) then return the information for the rest of the signal route definitions (i.e. do nothing about the signal route definition in hand).

Line 9 Extract the signal route identifier from *routeset.*

Line 10-14 If the signal route is unidirectional then if the signal route is "incoming" (line 11-12) then do nothing about the signal route definition in hand else return the *Reachabilities* from the rest of the signal route definitions joined with a *Reachability* containing the sending process (*p1*) , the signals conveyed by the signal route (*sset*) and the *Path* where the signal route identifier (*id*) has been added. Also return the possible updated *Entity-dict*

Line 16-22 If the signal route is bidirectional then extract the originating process (*originp*) from the appropriate $\textit{Signal-route-path}_1$ and then do the same as in line 14.

Line 26-27 If the signal route is connecting processes then no new *Reachability* is created, but the process descriptors in *Entity-dict* are updated with *reachabilities* for the communication paths between the two processes (handled in *make-local-reach*).

$$\textit{make-local-reach}(id, \textbf{mk-}\textit{Signal-route-definition}_1(rnm, path1, path2))(dict) \triangleq \qquad (5.4.9)$$

1 $(\textbf{let mk-}\textit{Signal-route-path}_1(p1, p2, sset) = path1 \textbf{ in}$
2 $\textbf{let mk-}\textit{ProcessDD}(parm, init, maxi, graph, inrset) = dict((p1, \mathsf{PROCESS})) \textbf{ in}$
3 $\textbf{let } reach = (p2, sset, \langle id \rangle) \textbf{ in}$
4 $\textbf{let } dict' = dict + [(p1, \mathsf{PROCESS}) \mapsto \textbf{mk-}\textit{ProcessDD}(parm, init, maxi, graph, inrset \cup \{reach\})] \textbf{ in}$
5 $\textbf{if } path2 = \textbf{nil then}$
6 $\quad dict'$
7 $\textbf{else}$
8 $\quad \textit{make-local-reach}(id, \textbf{mk-}\textit{Signal-route-definition}_1(rnm, path2, \textbf{nil}))(dict'))$

type : $\textit{Signal-route-identifier}_1\ \textit{Signal-route-definition}_1 \rightarrow \textit{Entity-dict} \rightarrow \textit{Entity-dict}$

Objective Update one or two process descriptors in *Entity-dict* with *Reachabilities* for the other process endpoint. Two process descriptors are updated only if the signal route is bidirectional.

Parameters

id The identifier of the signal route

$\textit{Signal-route-definition}_1$ The signal route definition containing

- *rnm* The name of the signal route
- *path1* The first $\textit{Signal-route-path}_1$
- *path2* The second (optional) $\textit{Signal-route-path}_1$

Result The Updated *Entity-dict*

Algorithm

Line 1-4 Update *Entity-dict* with *Reachabilities* for one of the directions. The *Reachability* which is added to the process descriptor for the sending process (*p1*) contains the receiving process (*p2*), the signals (*sset*) and a *Path* only containing the signal route identifier (*id*).

Line 5-8 If the signal route is unidirectional then return the updated *Entity-dict* else do the same where the signal route is regarded as unidirectional and the contained $\textit{Signal-route-path}_1$ is the one which has not been treated so far.

update-processd(outrset, inrset)(dict) ≜ (5.4.10)

```
1   if outrset = {} then
2     dict
3    else
4     (let outreach ∈ outrset in
5      let (pid, sigset, path) = outreach in
6      let inrset' =
7          {inr ∈ inrset | (let (,, path') = inr in
8                            path'[len path'] = hd path)} in
9      let mk-ProcessDD(parmd, init, maxi, graph, rset) = dict((pid, PROCESS)) in
10     let reachabilityset = extract-reachabilities(sigset, path, inrset') in
11     let dict' = dict +
12         [(pid, PROCESS) ↦ mk-ProcessDD(parmd, init, maxi, graph, rset ∪ reachabilityset)] in
13     update-processd(outrset \ {outreach}, inrset)(dict'))
```

type : *Reachability*-**set** *Reachability*-**set** → *Entity-dict* → *Entity-dict*

Objective Update process descriptors in *Entity-dict* with the *reachabilities* possible from *Reachabilities* containing outgoing (partial) *Paths* and from *Reachabilities* containing incoming (partial) *Paths*.

Parameters

outrset	The *Reachabilities* containing the outgoing *Paths*
inrset	The *Reachabilities* containing the incoming *Paths*.

Result The Updated *Entity-dict*

Algorithm

Line 1	Each *Reachability* containing outgoing *Paths* is examined. When through the set then return the (updated) *Entity-dict*
Line 4-5	Take a *reachability* from *outrset*.
Line 6	Extract those incoming *Reachabilities* which contain the continuation of the *Path* in the *Reachability* in hand, that is, extract those *Reachabilities* which has the same channel identifier at the end of the *Path* as the channel identifier at the beginning of the *Path* in hand (*path*).
Line 9	Decompose the process descriptor of the sending process.
Line 10	Go through all the incoming *Reachabilities* in order to construct the possible (complete) *Reachabilities*.
Line 11-13	Update *Entity-dict* with the new *Reachabilities* and use this updated *Entity-dict* when treating the rest of the outgoing *Reachabilities*.

extract-reachabilities(sigset, path, inrset) ≜ (5.4.11)

```
if inrset = {} then
  {}
 else
  (let inr ∈ inrset in
   let (pid, sigset', path') = inr in
   let reach = if sigset ∩ sigset' = {} then
                 {}
                else
                 {(pid, sigset ∩ sigset', path ⁀ tl path')} in
   {reach} ∪ extract-reachabilities(sigset, path, inrset \ {inr}))
```

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

Objective Construct *Reachabilities* from an outgoing (partial) *Path* and a set of incoming *Reachabilities.*

Parameters

sigset The set of signals on the outgoing *Path*

path The outgoing *Path*

inrset The set of incoming *Reachabilities.*

Result The constructed *Reachabilities.*

Algorithm

Line 1 When through the set of *Reachabilities*, return nothing.

Line 4-5 Take an incoming *Reachability* from *inrset.*

Line 5-6 Construct a *Reachability* which is empty if the incoming *Reachability* has no signals in common with the outgoing *Path*, otherwise it contains the receiver (*pid*), the intersection of signals possible in the incoming *Reachability* and in the outgoing *Path* and the complete *Path* constructed by concatenating the outgoing path (*path*) with the incoming path (*path*) except for the first element (i.e. the channel identifier which also occurs as the last element in the outgoing path).

Line 10 Return the constructed *Reachability* together with the *Reachabilities* constructed from the rest of the incoming *Reachabilities.*

Domain Index

Function Index

Processor Index

Variable Index

Error Messages

ISBN 92-61-03791-7