INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100 – Annex F.1

(03/93)

PROGRAMMING LANGUAGES

SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

ITU-T Recommendation Z.100 – Annex F.1

(Previously “CCITT Recommendation”)

FOREWORD

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

The World Telecommunication Standardization Conference (WTSC), which meets every four years, established the topics for study by the ITU-T Study Groups which, in their turn, produce Recommendations on these topics.

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

NOTES

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 Similarly, in this reform process, the CCIR and the IFRB have been replaced by the Radiocommunication Sector.

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

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

CONTENTS

SPECIFICATION AND DESCRIPTION LANGUAGE (SDL)

(Helsinki, 1993)

1 Preface

This Formal definition of SDL provides a language definition which supplements the definition given in the recommendation text. This annex is for use by those who require a very precise and detailed definition of SDL such as maintainers of the SDL language and designers of SDL tools.

The formal definition consists of three volumes:

Annex F.1	(This volume) Which states the motivation, describes the overall structure, provides guidelines for how to use the Formal Definition and describes the notation used.
Annex F.2	Which defines the static properties of SDL
Annex F.3	Which defines the dynamic properties of SDL

2 Motivation

Natural languages in general are **ambiguous** and **incomplete**, that is, more than one interpretation can be given to some of the sentences in the language, no matter whether the reader is a computer or a human being.

A definition or specification is **formal** if its meaning (semantics) is unambiguous and complete. As natural languages cannot be used for that purpose, special languages, known as **specification languages** (like SDL and LOTOS) have been developed. An implementation language like CHILL or PASCAL could also be used as a specification language (for instance a compiler specifies formally the semantics of another language), but often it is essential to separate the implementation details, irrelevant for the understanding, from the semantics of a specification.

Formal languages specially suitable for defining languages are known as **meta** languages. For example, The Backus Naur form (BNF) is a meta language specially suitable for defining formally the syntax of programming languages.

In spite of the ambiguity of natural languages, natural languages are usually more readable for human beings than formal languages and can more easily express rationale giving a framework in which the formal specification can be understood. For these reasons both a definition in natural language and a definition in a formal specification language often are given.

This annex constitutes a formal definition of SDL. If any properties of an SDL concept defined in this document, contradicts the properties defined in Z.100 and the concept is consistently defined in Z.100, then the definition in Z.100 takes precedence and this formal definition requires correction.

2.1 The Meta Language

The meta language used in this Formal Definition is Meta-IV [1]. The reasons for choosing this language are the following:

- It builds upon a very strong and extensively researched mathematical foundation.
- It has very convenient and powerful facilities for object manipulations.
- It has a "programming like" notation which means that it is oriented towards programmers and implementors.
- It is in the process of being standardised within the European Community.

- It is well reported in books, proceedings and scientific journals and it has been used in the CCITT manual The Formal Definition of CHILL [2] which also contains a summary of the Meta-IV notation.
- Meta-IV tools are available which allow for syntax checking, visibility analysis, document generation, cross referencing, etc.

In section 5, an informal introduction to the parts of Meta-IV used in the Formal Definition can be found. A complete definition of Meta-IV can be found in [1].

3 Modelling Technique

When considering what is meant by “semantics of SDL” it is convenient (conceptually) to decompose the language definition into several parts:

- The definition of the syntax rules.
- The definition of the static semantic rules (so-called well-formedness conditions) such as which names it is allowed to use at a given place, which kind of values it is allowed to assign to variables, etc.
- The definition of the semantics of the constructs in the language when they are interpreted (the dynamic semantics).

There is no need for including the syntax rules in the Formal Definition as the BNF rules and Syntax diagrams found in Z.100 already serve as formal definitions of the syntax rules, which means that the input to the Formal Definition is a syntactically correct SDL specification. The input is represented by an Abstract Syntax. This abstract syntax is based on the SDL textual concrete syntax parse-tree (BNF rules) with irrelevant details such as separators and lexical rules removed. Therefore, this Abstract syntax is not the Abstract Syntax of Z.100 appearing in the Recommendations which is an abstraction of the SDL model concept.

For example, the Abstract Syntax production rule:

1	*Transition*	::	*Actstmt** [*Termstmt*]

expresses that a *Transition* consists of a possible empty list of *Act*ion *st*ate*m*en*t*s and an optional *Term*inator *st*ate*m*en*t* (the italicized letters also occur in the production rule). The complete set of production rules (so-called Domain Definitions) defining the SDL-syntax on an abstract form is called AS_0. In some respect it defines the language syntax on a more basic level than the syntax rules found in Z.100 since the concrete textual syntax in Z.100 contains a lot of semantic information (it is context sensitive) as opposed to AS_0. It should be noted that AS_0 is an abstraction of the concrete textual syntax. The concrete graphical syntax has not been used for reasons of economy in time and space rather than any difficulty in the task.

As an example, a signal list in Z.100 is defined to be:

<signal list> ::= <signal list item> {, <signal list item>}

<signal list item> ::= <signal identifier> | (<signal list identifier>) | <timer identifier>

whereas the corresponding definitions in AS_0 are:

2	*Signallist*	::	*Signallistitem*$^+$
3	*Signallistitem*	=	*Id* | *Signallistid*

A *Signallist* consists of a list of *Signallistitems.* A *Signallistitem* is either an identifier or a signal list identifier. As opposed to the context sensitive BNF production <signal item> no distinction is made between a signal identifier and a timer identifier in AS_0 because syntactically they are both identifiers as opposed to signal lists which are distinguished by the use of parenthesis.

The starting point for the FD is syntactically correct SDL-specifications. The tasks of the Formal Definition are to:

- Define the well-formedness conditions for SDL-specifications. This task, referred to as the Static Semantics, constitutes Annex F.2.
- Define the dynamic properties for SDL-specifications. This task, referred to as the Dynamic Semantics, constitutes Annex F.3.

The steps are shown in Figure 1. The result from the Static Semantics (i.e. AS_1) is explained below.

The step of translating from the concrete textual syntax to AS_0 is not formally defined, but is derived from the correspondence between names in the two syntaxes as previously illustrated for *Signallist.*

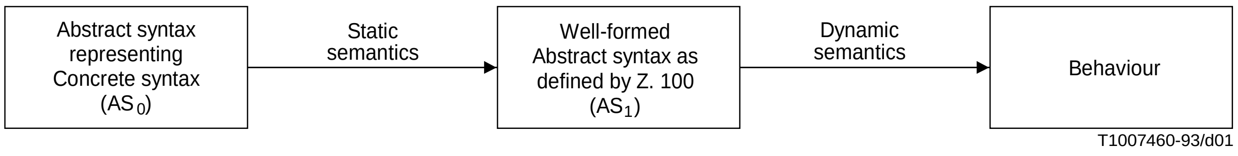
Abstract syntax representing Concrete syntax (AS_0)
Static semantics
Well-formed Abstract syntax as defined by Z. 100 (AS_1)
Dynamic semantics
Behaviour
T1007460-93/d01

FIGURE 1/Z.100

Objective of Static Semantics and Dynamic Semantics

3.1 Static Semantics

In Z.100, the dynamic semantics of the various constructs are defined in terms of an Abstract Syntax. Common subsections, *Semantics*, *Concrete textual grammar*, *Concrete graphical grammar* and *Model* define the concrete syntax rules, state the appropriate well-formedness conditions and relate the concrete syntax rules to other concrete syntax rules (in the *Model)* and to the abstract grammar in Z.100. The abstract grammar is in Z.100 defined using Meta-IV (in the common subsections *Abstract grammar*). The same abstract syntax is used in the Formal Definition (where it is referred to as AS_1). AS_1 is listed in Annex F.3 and it is identical to the abstract syntax of the abstract grammar in Z.100.

In addition to defining the well-formedness conditions, the Static Semantics must therefore define how the AS_0 representation of a specification is transformed into the AS_1 representation, that is, given an AS_0 representation, an AS_1 representation is returned by the Static Semantics if the AS_0 representation was well-formed. The Static Semantics can be regarded as an “abstract compiler” where the AS_0 representation is the source language and the AS_1 representation is the object language.

In addition to AS_0 and AS_1, the Static Semantics uses some internal utility domains, known as the Semantic Domains, which hold the information required at any place about a given entity. For example, when a process definition is transformed, information about its formal parameters is kept in the Semantic Domains and the information is retrieved during transformation of the Create Request action. The AS_0 domains could have been used for that purpose, as the Semantic Domains anyway are deduced from AS_0, but a tree representation is not useful when information of a certain entity (say a process definition) occurring somewhere in the tree is required. Therefore Semantic Domains are usually mappings modelling tables.

For instance, the Semantic Domains include a mapping (further explained in section 5.4.7) of identifiers into some descriptor containing information about the identifiers:

4	*Descriptordict*	=	*Qual* $\xrightarrow{m}$ *Descr*

where *Qual* is the identifier representation used internally in the Formal Definition and *Descr* is any descriptor. The descriptor may for instance be a process descriptor:

5	*Descr*	=	*ProcessD* | …
6	*ProcessD*	::	*ParameterD** *Validinputset Localinputset*

expressing that a *Process D*escriptor contains a list of *Parameter D*escriptors, information about the *Valid input* signal *set* and information about the *Local input* signal *set.* The definitions of these three (sub)descriptors are not shown here.

The transformation itself is performed by a set of Meta-IV functions using the three Domains AS_0, AS_1 and the Semantic Domains.

3.2 The Dynamic Semantics

The task of the Dynamic Semantics is to define the behaviour of an SDL specification on AS_1 form.

The Dynamic Semantics is divided into three major sections:

- The Model for the underlying system (the abstract SDL-machine).
- The Interpretation of the process graphs.
- Transformation of AS_1 into a more appropriate representation; that is, a mapping is constructed (a Semantic Domain) which contains the information required during the interpretation such as information about the sorts of variables, possible communication paths between processes, equivalence classes for types, etc. The mapping is named *Entity-dict* (or more correctly, the domain of the mapping is named *Entity-dict*).

Concurrency in SDL is in the Dynamic Semantics modelled by using **Meta-processes**; that is, concurrently executing Meta-processes in Meta-IV model concurrently executing processes in SDL.

Eight different Meta-process types are used:

- *system*

  To handle the signal routing between SDL process instance sets and the generation of unique PId values.

- *path*

  To handle the non-deterministic delay of (delaying) channels.

- *view*

  To keep track of all revealed variables.

- *timer*

  To keep track of the current time and handle time-outs.

- *process-set-admin*

  To handle all ingoing signals and create requests and to manage the other Meta-processes needed to interpret an SDL process instance set. For each SDL process instance set there exists exactly one instance of *process-set-admin.*

- *input-port*

  Which handles the queueing of signals in an SDL process instance. For each SDL process instance there exists exactly one instance of *input-port.*

- *sdl-process*

  To interpret the behaviour of the body of an SDL process instance. For each SDL process instance there exists exactly one instance of *sdl-process.*

  If the SDL process is decomposed into services *sdl-process* manages the Meta-processes needed to interpret the services.

- *sdl-service*

  To interpret the behaviour of (the body of) an SDL service. For each SDL service instance there exists exactly one instance of *sdl-service.*

There is, in most cases, no shared data between Meta-processes – they interact by transmitting values conveyed by instances (objects) of **Communication Domains** (correspond to the SDL concept signals). The only exception is *sdl-service* instances which may access and modify Meta-IV variables owned by their managing *sdl-process* instance.

Communication Domains are defined in the same way as other domains; for example, objects of the Communication Domain *Input-Signal* are directed to an *sdl-process* instance from its attached *input-port* instance. The Communication Domain is defined like this:

7 *Input-Signal* :: $\textit{Signal-Identifier}_1$ *Value-List Sender-Value*

Instances of *Input-Signal* convey the identifier of the SDL signal which is sent, the list of values conveyed by the SDL signal and the PId value of the sender.

Figures 2 and 3 show the complete "Meta-process interaction scheme". The Meta-processes for interpreting an SDL process instance set is shown as *sdl-process-set* in Figure 2 and detailed in Figure 3. The communication mechanism is synchronous and the notation is known as CSP (see [3] and [4]) (Communicating Sequential Processes).

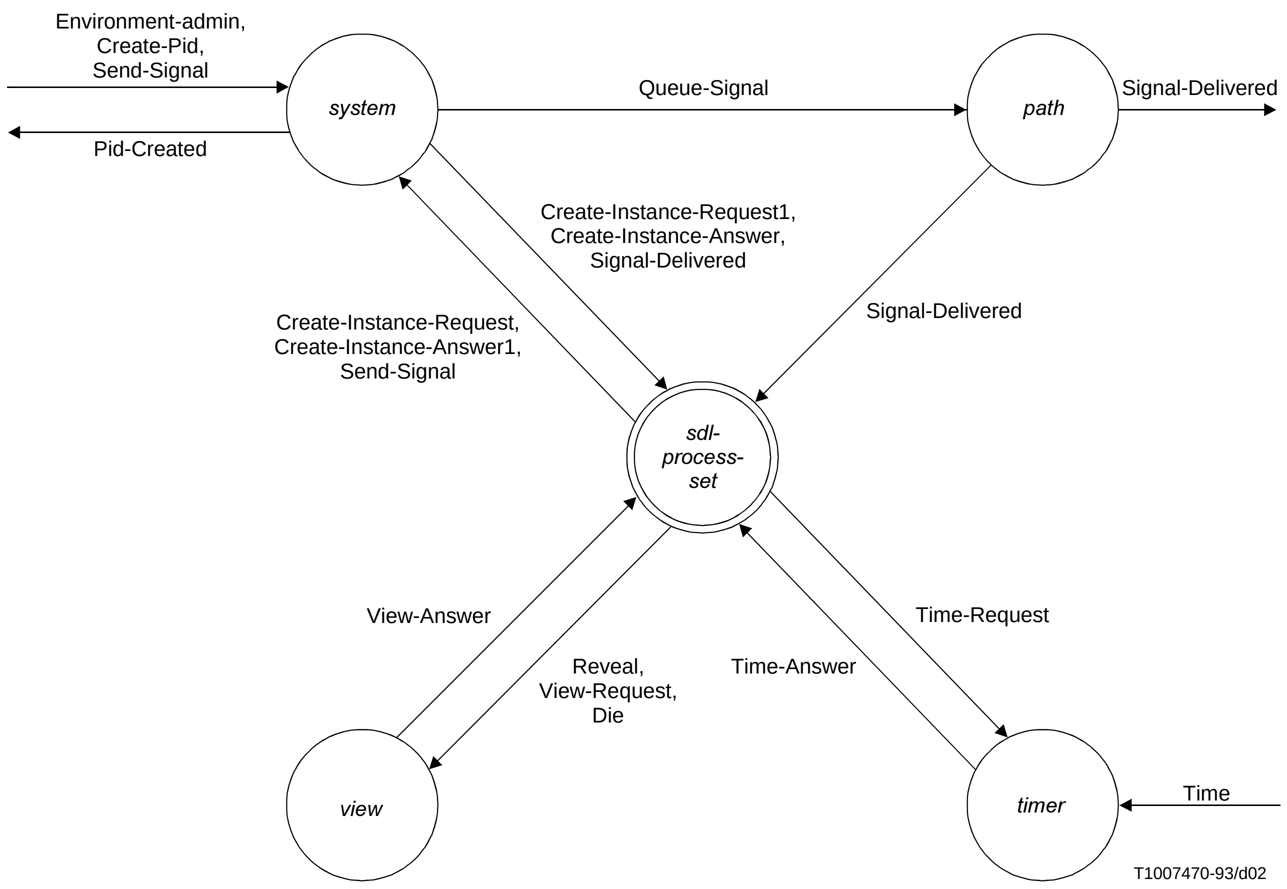
Environment-admin, Create-Pid, Send-Signal
Pid-Created
system
Queue-Signal
path
Signal-Delivered
Create-Instance-Request1, Create-Instance-Answer, Signal-Delivered
Create-Instance-Request, Create-Instance-Answer1, Send-Signal
Signal-Delivered
sdl-process-set
View-Answer
Reveal, View-Request, Die
Time-Answer
Time-Request
view
timer
Time
T1007470-93/d02

FIGURE 2/Z.100

Overall communication scheme

3.3 Example

Figure 4 shows the communication between meta-processes in the formal definition for the following (partial) SDL-process, when a signal ("b") arrives from the environment, and the process responds by sending a signal ("a") back to the environment:

```
…
state S;
    input b;
    output a;
…
```

The communication is informally illustrated by means of a message sequence chart. Path(1) and Path(2) denote two instances of the *path*-processor, corresponding to the path from the environment to the SDL process instance set [Path(l)] and vice versa [Path(2)].

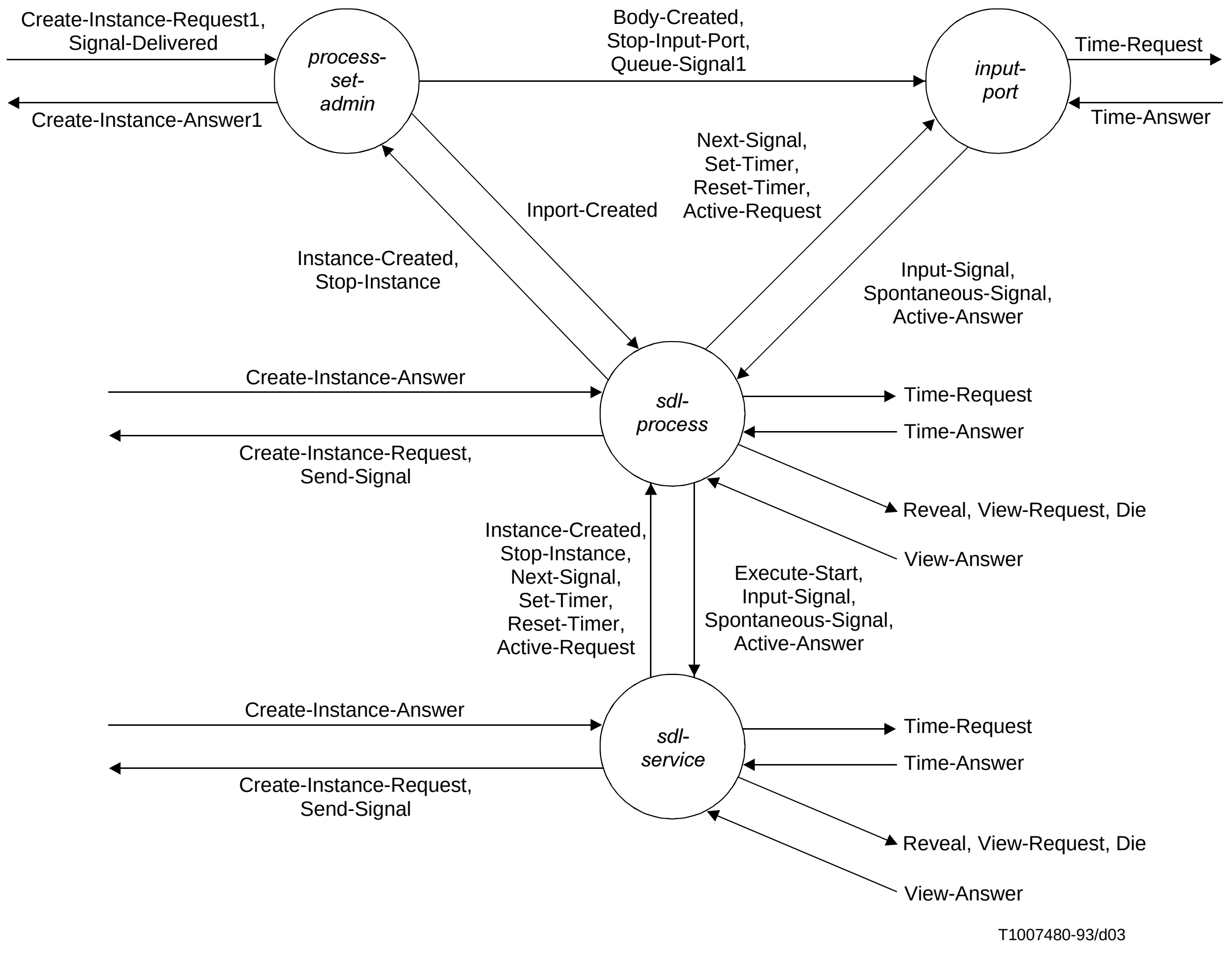

FIGURE 3/Z.100

Communication scheme for SDL process instance set

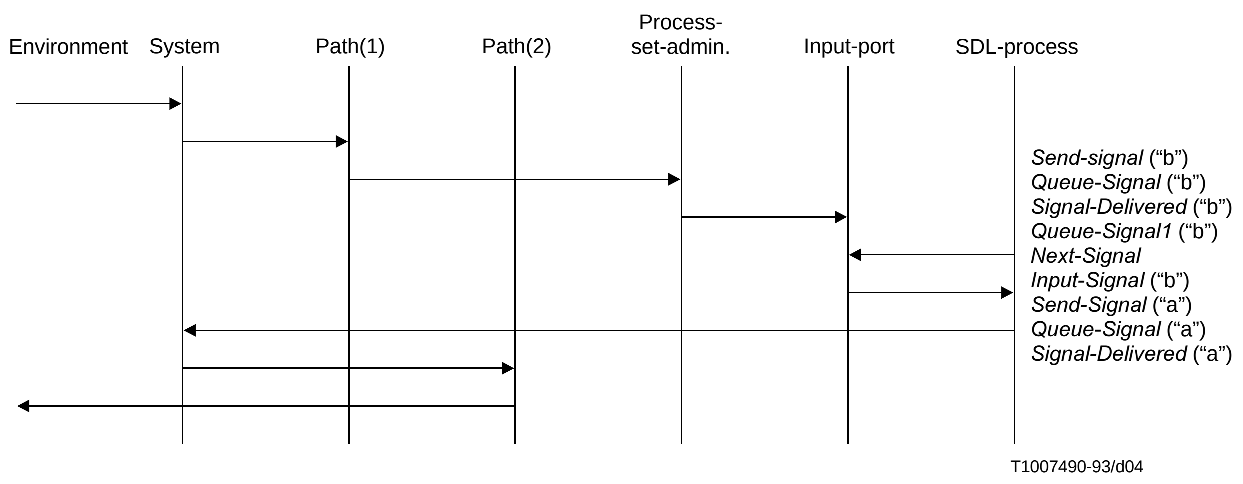

FIGURE 4/Z.100

Example of communication between meta-processes

3.4 Physical Structure of The Formal Definition

The Static Semantics (Annex F.2) is divided into three main parts:

1. The Domain definitions for AS_0.
2. The Domain definitions for the Semantic Domains.
3. The Meta-IV functions checking well-formedness conditions and defining how AS_0 is transformed into AS_1.

Annex F.2 also includes cross-indices on Meta-IV function names and domain names (both defining occurrence and applied occurrences) and a cross index on the well-formedness conditions applied.

The Dynamic Semantics (Annex F.3) is divided into seven major sections:

1. A summary of the Abstract Syntax (AS_1) Domains.
2. Domain definitions for the Communication Domains.
3. Domain definitions for the Semantic Domains (*Entity-dict*).
4. The Meta-process definitions and attached functions for the model of the underlying system.
5. The Meta-process definitions and attached functions for the interpretation of an SDL process and SDL service.
6. The creation of the internal domain *Entity-dict*. *Entity-dict* is used by the SDL processes and SDL services and it is therefore created before any SDL processes and SDL services are interpreted.
7. Some simple general-purpose auxiliary functions on Abstract Syntax Domains.

Annex F.3, like Annex F.2, also contains a number of indices covering domain names, function names, Meta-process names, error conditions, etc.

The volume of material (especially in Annex F.2) might seem frightening at a first glance. However, more than half of the space contains annotations for the Domains, function and process definitions.

The layout for a function and process definition follows a scheme:

1. First, the function or process definition is specified by:
   (a) a heading defining the process or function name and the names of its formal parameters;
   (b) its body (algorithm);
   (c) a *type clause* specifying the type (domain) of the formal parameters and the type of the result (if any).
2. Then follows the itemized (plain English) annotations attached to the process or function definition:

Objective	Explains the purpose of the function or process.
Parameters	Explains the purpose of every formal parameter to the function or process.
Result	Explains the object returned (if any).
Algorithm	Explains, on a line by line base, the algorithm used in the function or process.

Example

The outermost function *definition-of-SDL* from Annex F.2 which ties together the Static Semantics (*transform-system*) and the Dynamic Semantics (by starting the Meta-process *system*) is as follows:

definition-of-SDL(extparms, systemtext, predeftext) $\triangleq$

```
1  (let (systemdef, predef) = construct-AS_0 (systemtext, predeftext) in
2  let (as_1, auxinf) = transform-system (systemdef, predefsorts, extparms) in
3  if as_1 = nil then
4     undefined
5    else
6     (let subsetcut = select-consistent-subset (as_1, extparms) in
7      start system (as_1, subsetcut, auxinf)))
```

type: *External-Information* Sys_0 $Datadef_0^+ \Rightarrow$

Objective Define the properties of SDL

Parameters

extparms	Some *External-Information* (see Annex F.2 Section 2.3).
systemtext	The sequence of lexical elements (i.e. characters) representing the SDL specification.
predeftext	The sequence of lexical elements (i.e. characters) representing the predefined data.

Algorithm

Line 1	Transform the text for the SDL specification (*systemtext*) and the text for the predefined data (*predeftext*) into their AS_0 representations.
Line 2	Transform the system into the abstract syntax form (AS_1 form).
Line 3	If static errors are found (i.e. if no AS_1 representation could be derived) then the behaviour is not defined.
Line 4	If no static errors are found then.
Line 6	Select the set of *Block-identifier*$_1$*s* denoting the consistent subset.
Line 7	Create a system instance, i.e. create a Meta-IV process which behaves like the underlying system.

4 How to Use the Formal Definition

4.1 The SDL Users

The Formal Definition is not intended as a user’s reference manual on SDL. Newcomers on SDL may find the text books more appropriate for achieving an overview of concepts (and their rationale) in the language, while the Z.100 Recommendation itself serves as a reference manual on SDL, but there might be some cases where Z.100 is inadequate. For instance:

- if some properties are missing (e.g. some expected static condition), if some stated properties contradict other properties; or
- if the exact meaning of some stated properties is difficult to understand; or
- if some properties (due to the lack of cross index in Z.100) are difficult to find; or
- if the user wants to achieve a deeper understanding of more complex matters like the abstract SDL machine, when and how to select a consistent subset, resolution by context, the inheritance mechanism, etc.

In such cases the Formal Definition might be a useful supporting document. The user must of course first gain insight in the structure of the Formal Definition, how the functions are organized and what the Domains are used for. A certain amount of knowledge about the Meta-IV notation is also required, but as the functions are extensively annotated, it may be possible to read Meta-IV by reading the functions in conjunction with the annotations after having read the introduction on Meta-IV (section 5 below). When looking up in the Formal Definition, the users may take advantage of using the table of contents and the cross indices.

4.2 The Implementors

As mentioned earlier, the Meta-IV approach allows implementors to derive an implementation systematically (i.e. static analyzer, simulator, etc.) from the Meta-IV specification. For SDL, it is possible to derive a static analyzer from Annex F.2 and a simulator from Annex F.3. It is advised to use the AS_1 representation (generated by the static analyzer) as a basis for simulation. The reasons are that context information for identifiers is missing in AS_0 (they are normally not qualified in AS_0) and that the dynamic semantics of a specification on AS_0 form may be difficult to derive due to the large number of shorthands in SDL (especially for concepts like data types).

It should be noted that the derivation into an implementation is systematical, but it is not mechanical.

The following points must be considered:

- Appropriate datatypes must be found for representing the ideal data types (domains) in Meta-IV such as mappings, lists and sets used in AS_0, AS_1 and the Semantic Domains.
- The initial algebra approach implies that the Formal Definition manipulates infinite objects. Also AS_1 contains infinitely objects. It is therefore necessary to modify AS_1 slightly and to impose restrictions on the use of data types or to use some abstraction technique in which these objects can be encoded.

5 Introduction to Meta-IV

This section contains an informal introduction to Meta-IV and to how Meta-IV has been used in the Formal Definition, i.e. Meta-IV is explained in terms of the Formal Definition (abbreviated as FD) which means that only those parts of Meta-IV which have been used in the FD are explained.

5.1 General Structure

The FD consists of:

- A set of function and process definitions defining the semantics of SDL. Processes (in Meta-IV and in the FD called processors) are used for modelling concurrency and are therefore only used in the Dynamic Semantics. Syntactically, processor definitions look like function definitions (except for the keyword **processor** following the processor name); therefore, the following description of the function concept also applies for processors.
- A set of domain definitions which define the type of the objects manipulated with by the functions. Terms denoting certain groups of domain definitions are introduced in order to classify them logically. We have the AS_0 domains denoting the representation of the concrete syntax, the AS_1 domains denoting the abstract syntax of SDL and the sets of domains *Dict* and *Entity-dict* denoting the “internal” utility domains (semantic domains) of the Static- and Dynamic Semantics respectively. In this section, we will often use “value” as a synonym for object and “type” as a synonym for domain.

Definitions may be specified in any order and names introduced in definitions may be used before they textually are defined.

5.2 Function Definitions

A function definition consists of three parts:

1. The heading starts with the function name and is followed by one or two formal parameter lists. Each formal parameter list is enclosed in parenthesis. There is no formal significance in dividing the parameters between two lists. Often some parameters are put into a separate (second) parameter list if they are of secondary importance for the evaluation. For instance, in the case of the semantic domains which often are used by the functions and supplied in a separate parameter list.
2. The body of the function which can either be an expression or a sequence of statements. A function does not have to deliver any result (see below).
3. The type clause specifying the type of the formal parameters and the type of the result. First, the type of the first parameter list is specified, then the type of the second parameter list (if any) separated by the first parameter list by an arrow ($\rightarrow$ or $\Rightarrow$), then another arrow and then the result.

Example

$f(a, b)\ (d) \triangleq$

1 /∗ *expression* ∗/

type: $DomX\ DomY \rightarrow DomZ \rightarrow DomW$

In this example we have:

f	is the name of the function.
a, *b*, *d*	are formal parameters. *a* and *b* are contained in the first formal parameter list and *d is* contained in the second parameter list. The type of *a* is *DomX*, the type of *b* is *DomY* and the type of *d* is *DomZ.* The type of the result is *DomW.* The domains *DomX, DomY, DomZ* and *DomW* must be defined in some domain definitions.

If the formal parameters or the result are not used in accordance with the type clause, there is an error in the Meta-IV specification. In the example above, informal Meta-IV text (the text enclosed in /* */) is used to denote some Meta-IV expression which for reasons of economy in space has been left out. Informal Meta-IV text is similar to informal text in SDL and it is extensively used in the examples of this section.

Normally, a distinction is made between **applicative** and **imperative** functions. Applicative functions are functions which do not refer to parts of the global state (variables), that is, the result of such functions are only depending on the value of the applied actual parameters. The body of an applicative function is restricted to be an expression as statements impose some change of state. Applicative functions must always deliver a result. Imperative functions are functions which refer to or even change the global state (functions with side effects). If a function is imperative, it must be reflected in the type clause by using $\Rightarrow$ instead of $\rightarrow$ when specifying the result.

That is:

f(*a*, *b*) (*d*) $\triangleq$

1 /∗ *expression referring to the global state or sequence of statements* ∗/

type: *DomX DomY* $\rightarrow$ *DomZ* $\Rightarrow$ *DomW*

In the FD, the Static Semantics and the creation of the internal Domain *Entity-dict* in the Dynamic Semantics are applicative.

5.3 Variable Definitions

Global variables are defined at the outermost level in processor definitions. They are visible to all functions used by the processor defining the variable even though the functions normally are defined outside processor definitions. However, a function which is shared by two or more processors is not allowed to access variables. When several instances of a given processor exist, several instances of variables defined by the processor also exist. (There are no shared variables.)

Variable definitions are introduced by specifying the keyword **dcl** followed by a list of variable names, optionally followed by an initial expression and ending with the type of the variable.

Example

dcl v1 := 5 **type** *Intg*;

dcl v2 **type** *DomD*;

Here we have defined two variables v1 and v2, v1 is of type integer and is initialized to 5. v2 is of type *DomD.* Note that variables can always be distinguished syntactically from other names since they are not italicized. An alternative syntax of variable definitions is:

dcl v1 := 5 **type** *Intg*;
    v2 **type** *DomD*;

The value associated to variables is accessed by using the contents operator which is the keyword **c**.

Example

f () $\triangleq$

1 **c** v1 + **c** v2

type: () $\Rightarrow$ *Intg*

5.4 Domains

Domains are usually defined in the beginning of a document. Domain names can be distinguished syntactically from other names since the first letter is in capital. A domain is defined by specifying the domain name followed by a "::" symbol (or by a "=" in the case of a synonym name as explained in section 5.4.1) and then followed by a domain expression reflecting its properties (for an introduction to the domain notation see also 1.5.1 of Z.100).

Example

8 *Output-node*$_1$ :: *Signal-identifier*$_1$
[*Expression*$_1$]*
[*Signal-destination*$_1$]
Direct-via$_1$

This example is taken from the abstract syntax of SDL (for clarity, all the names of AS$_1$ are suffixed by a "$_1$" in the FD). It defines a **named tree,** that is, a record-like datatype where the name of the recordtype is *Output-node*$_1$ and it's fields are of the type *Signal-identifier*$_1$, [*Expression*$_1$]*, [*Signal-destination*$_1$] and *Direct-via*$_1$.

The most important operator for named trees is the **mk**- (make) operator which is used for composing and decomposing tree objects (i.e. record values).

For example, if a name *sigid* denotes an object of domain *Signal-identifier*$_1$, a name *exprlist* denotes an object of domain [*Expression*$_1$]*, a name *dest* denotes an object of type [*Signal-destination*$_1$] and a name *via* denotes an object of domain *Direct-via*$_1$ then an object of domain *Output-node*$_1$ is constructed by writing:

mk-*Output-node*$_1$ (*sigid, exprlist, dest, via*)

which can be used in Meta-IV expressions. Note that the order in which the arguments are specified in the **mk**- operator is significant. This applies for function calls as well.

Similarly, if we have an object, named *outputnode*, of domain *Output-node*$_1$ and we want to access the fields, we can introduce names for the fields by decomposing it (the same names as above are chosen here):

let mk-*Output-node*$_1$ (*sigid, exprlist, dest, via*) = *outputnode* **in**

/∗ *some expression using the fields* ∗/

By means of the **let** construct we have introduced names to denote the fields in the object *outputnode.* Using the **let** construct is the general way of introducing names for objects (not only in combination with the **mk-** operator). The **let** construct is explained further in section 5.5.

If some of the fields are not used in the expression we can omit the corresponding names in the decomposition. For instance, if *sigid* is not used in the expression, we can write:

let mk-*Output-node*$_1$ (, *exprlist, dest, via*) = *outputnode* **in**

/ some expression using exprlist and dest */*

If we only want to use the *Signal-Identifier*$_1$ in the expression we can alternatively use the field select operator **s-**:

let *sigid* = **s**-*Signal-Identifier*$_1$ (*outputnode*) **in**

/ some expression using sigid */*

The field select operator can only be used if the field can be uniquely determined by mentioning the domain name.

We can choose to decompose (i.e. introduce names for the contained elements) the formal parameters in the function head instead of in the body if we find it more readable. That is:

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

1 /∗ *body of int-create-node* ∗/

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

is equivalent to

int-create-node (*createnode*) (*dict*) $\triangleq$

1 (**let mk**-*Create-request-node*$_1$ (*prid, exprl*) = *createnode* **in**
2 /∗ *body of int-create-node* ∗/)

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

NOTE – In this example we also have a second parameter list containing the formal parameter *dict* of the domain *Entity-dict.*

5.4.1 Synonyms

Using the field select operator is only possible if the field in the domain definition is represented by a name. If for instance we want to use the select operator on the second field of objects of the domain *Output-node*$_1$, we must define *Output-node*$_1$ in a slightly different way:

9	*Output-node*$_1$	::	*Signal-identifier*$_1$
			Valuelist
			[*Signal-destination*$_1$]
			Direct-Via$_1$
10	*Valuelist*	=	[*Expression*$_1$]*

This *Output-node*$_1$ is exactly the same domain as the *Output-node*$_1$ previously defined. The only difference is that we have given the second field a name, i.e. we have defined a synonym or shorthand for the domain expression [*Expression*$_1$] * (the "=" symbol is used when defining synonyms). Often there are other reasons for defining synonyms such as if the same domain expression is used at several places or for the sake of readability. For instance, in the abstract syntax of SDL, we have *Channel-name*$_1$, *Block-name*$_1$, *Process-name*$_1$, etc. which all are synonyms for the domain *Name*$_1$, but which carry information to the reader about the objects represented by the various *Name*$_1$*s* being of certain entity classes. Another typical case is where we have a long list of alternatives. For instance, the abstract syntax for *Expression*$_1$ is

11	*Expression*$_1$	=	*Ground-expression*$_1$ |
			Active-expression$_1$ |
12	*Active-expression*$_1$	=	*Variable-access*$_1$ |
			Conditional-expression$_1$ |
			Operator-application$_1$ |
			Imperative-operator$_1$ |
			Error-expression$_1$ |
13	*Imperative-operator*$_1$	=	*Now-expression*$_1$ |
			Pid-expression$_1$ |
			View-expression$_1$ |
			Timer-active-expression$_1$ |
			Anyvalue-expression$_1$ |
14	*Pid-expression*$_1$	=	*Self-expression*$_1$ |
			Parent-expression$_1$ |
			Offspring-expression$_1$ |
			Sender-expression$_1$ |

which better reflects the grouping of the various kinds of expressions than

15	*Expression*$_1$	=	*Ground-expression*$_1$ |
			Variable-access$_1$ |
			Conditional-expression$_1$ |
			Operator-application$_1$ |
			Now-expression$_1$ |
			Self-expression$_1$ |
			Parent-expression$_1$ |
			Offspring-expression$_1$ |
			Sender-expression$_1$ |
			View-expression$_1$ |
			Timer-active-expression$_1$ |
			Anyvalue-expression$_1$ |
			Error-expression$_1$ |

5.4.2 Unnamed Trees

In some cases, we do not need to name a tree definition. **Unnamed trees** are extensively used in the FD, but they are anonymous since they often do not have to be defined explicitly.

Example

The first line in the definition of *Entity-dict* in the Dynamic Semantics is:

16 *Entity-dict* = (*Qualifier*$_1$ **TYPE**) $\overset{m}{\rightarrow}$ *TypeDD*

which expresses that the *Entity-dict* includes a mapping from the two domains *Qualifier*$_1$ and **TYPE** into some descriptor (*TypeDD*). These two domains constitute an unnamed tree. If a named tree should be used, we would have to rewrite the definition into:

17 *Entity-dict* = *Pair* $\overset{m}{\rightarrow}$ *TypeDD*
18 *Pair* :: *Qualifier*$_1$ **TYPE**

Example

Reachability in the dynamic semantics is defined as

19 *Reachability* = *Reachability-endp Signal-identifier*$_1$-**set** *Path*

Here we have defined a synonym for an unnamed tree containing three fields:

1. A field of the domain *Reachability-endp*.
2. A field which contains a set of signal identifiers.
3. A field of the domain *Path*.

As shown, parentheses are in the domain definitions both used for defining unnamed trees and for grouping alternatives.

Example

The function *make-procedure-formal-parameters* in the Dynamic Semantics is defined as:

make-procedure-formal-parameters(*parml, level*) $\triangleq$

1 /∗ *The body, which is not shown here* ∗/

type: *Procedure-formal-parameters*$_1$* *Qualifier*$_1$ $\rightarrow$ *FormparmDD** *Entity-dict*

This function returns two objects, *FormparmDD** and *Entity-dict* which means that it in fact returns an unnamed tree consisting of two objects.

The **mk**- operator cannot be used on unnamed trees. Composition and decomposition of these is obtained by enclosing the fields in parentheses.

Example

Composition of a *Reachability* object where *a* denotes a *Reachability-endp, b* denotes a signal identifier set and *d* denotes a *Path*:

(*a, b, d*)

if, for the sake of readability, we want to denote the object by a name (it is easier to deal with a name than with (*a, b, d*), especially if (*a, b, d*) is used several times in an expression) then we can again use the **let** construct, that is, the expression:

/∗ *some expression using* “(a, b, d)” ∗/

is equivalent to

(**let** *reach* = (*a, b, d*) **in**
/∗ *some expression using “reach”* ∗/)

The **let** construct is also used for decomposing objects of unnamed trees. For example, a decomposition of a *Reachability* object named *reach* where we for some reason do not use the signal identifier set is:

let (*a, ,d*) = *reach* **in**
/∗ *some expression using a and d* ∗/

When we call a function, it is usual to decompose unnamed trees which are the result of the function call, i.e.:

let (*parmddl, fdict*) = *make-procedure-formal-parameters*(…, …) **in**
/∗ some expression using the function results parmddl and fdict ∗/

is equivalent to:

let *parminf* = *make-procedure-formal-parameters*(…, …) **in**
let (*parmddl, fdict*) = *parminf* **in**
/∗ some expression using the function results parmddl and fdict ∗/

5.4.3 Branching Constructs

In some cases, it must be possible to distinguish a number of tree objects from each other. For instance, objects of the *Imperative-operator*$_1$ synonym previously defined is either a *Now-expression*$_1$, a *Pid-expression*$_1$, a *View-expression*$_1$, etc. With an *Imperative-operator*$_1$ in hand, we must first determine the type of the *Imperative-operator*$_1$ before we can evaluate it. For that purpose, we can use the case expression/statement. For instance, a function which evaluates the imperative SDL expressions could look like:

eval-imperative-expression(*expr*) $\triangleq$

```
 1  cases expr:
 2  (mk-Now-expression$_1$()
 3    → eval-now-expression(),
 4  mk-View-expression$_1$(vid, pidexpr)
 5    → eval-view-expression(vid, pidexpr)
 6  mk-Timer-active-expression$_1$(tid, actlist)
 7    → eval-timer-expression(tid, actlist)
 8  mk-Anyvalue-expression$_1$(sortref)
 9    → eval-anyvalue-expression(sortref),
10  T → eval-pid-expression(expr))
```

type: *Imperative-operator*$_1$ ⇒

Note that we branch on the **type** of the *Imperative-operator*, not on the actual value of the fields in the tree. **T** denotes an “otherwise” clause which is used here because the final alternative in *Imperative-operator*$_1$ (*Pid-expression*$_1$) is a synonym representing four other alternatives which we do not want to distinguish here. The evaluation of these alternatives is deferred to *eval-pid-expression.*

Another way of doing it is by using the boolean operator **is-** which returns **true** if the object given as argument is of a certain domain, e.g.

eval-imperative-expression(*expr*) $\triangleq$

```
 1  if is-Now-expression$_1$(expr) then
 2     eval-now-expression()
 3    else
 4    if is-View-expression$_1$(expr) then
 5        eval-view-expression(s-Variable-identifier$_1$(expr),s-Expression$_1$(expr))
 6       else
 7       if is-Timer-active-expression$_1$(expr) then
 8           (let mk-Timer-active-expression$_1$(tid, actlist) = expr in
 9           eval-timer-expression(tid, actlist))
10         else
11          if is-Anyvalue-expression$_1$(expr) then
12              (let mk-Anyvalue-expression$_1$(sortref) = expr in
13               eval-anyvalue-expression(sortref))
14             else
15              eval-pid-expression(expr)
```

type: *Imperative-operator*$_1$ ⇒

Note that both access to the fields by decomposition (line 8) and access to the fields by means of the field selection operator (line 5) are illustrated here.

As in most other programming and specification languages, it is required that the alternatives in the case expression/statement are “constant” (as they are when we branch on the tree type) which means that if the alternatives are of a dynamic nature (say variables or formal parameters) the if-then-else construct must be used. However, there is another notation for the if-then-else construct, the so-called Mc-Carthy construct which is more convenient if there are many alternatives:

eval-imperative-expression(*expr*) $\triangleq$

```
1   (is-Now-expression₁(expr)
2     → eval-now-expression( ),
3   is-View-expression₁(expr)
4     → (let mk-View-expression₁(vid, pidexpr) = expr in
5           eval-view-expression(vid, pidexpr)),
6   is-Timer-expression₁(expr)
7     → (let mk-Timer-expression₁(tid, actlist) = expr in
8           eval-timer-expression(tid, actlist)),
9   is-Anyvalue-expression₁(expr)
10    → (let mk-Anyvalue-expression₁(sortref) = expr in
11          eval-anyvalue-expression(sortref)),
12  T → eval-pid-expression(expr))
```

type: *Imperative-operator*$_1$ $\Rightarrow$

Note that some FD function names also start with “is-”. These cases can easily be distinguished from the “**is-**” operator since they are not in boldface.

5.4.4 Elementary domains

Meta-IV provides a number of predefined elementary domains. Their notation and the associated operators are described in the following.

5.4.4.1 Boolean

The Meta-IV name *Bool* denotes the domain of truth values, i.e. the set {**true**,**false**}

Operators for Boolean:

Notation	Type			Operation
$\neg$	*Bool*	$\rightarrow$	*Bool*	negate
$\wedge$	*Bool*	$\rightarrow$	*Bool*	and
$\vee$	*Bool*	$\rightarrow$	*Bool*	or
$\supset$	*Bool*	$\rightarrow$	*Bool*	imply
=	*Bool Bool*	$\rightarrow$	*Bool*	equal
$\neq$	*Bool Bool*	$\rightarrow$	*Bool*	different

Example

In terms of Meta-IV expressions, the properties of the *Bool* operators $\neg$, $\wedge$, $\vee$ and $\supset$ can be illustrated as follows:

$\neg a$ = (**if** a **then false else true**)
$a \vee b$ = (**if** a **then true else** b)
$a \wedge b$ = (**if** a **then** b **else false**)
$a \supset b$ = (**if** a **then** b **else true**)

5.4.4.2 Integer

Three domain names are predefined for the integer values:

- The name *Intg* denotes the domain of all integer values, i.e. the set {... –2,–1,0,1,2,...}
- The name N_0 denotes the domain of non-negative integer values, i.e. the set {0,1,2,...}
- The name N_1 denotes the domain of positive integer values, i.e. the set {1,2,...}

Operators for Integer:

Notation	Type			Operation
–	*Intg*	→	*Intg*	negate
–	*Intg Intg*	→	*Intg*	subtract
+	*Intg Intg*	→	*Intg*	add
*	*Intg Intg*	→	*Intg*	multiply
/	*Intg Intg*	→	*Intg*	integer divide
mod	N_0 N_1	→	N_0	modulus
=	*Intg Intg*	→	*Bool*	equal
≠	*Intg Intg*	→	*Bool*	different
<	*Intg Intg*	→	*Bool*	less than
≤	*Intg Intg*	→	*Bool*	less than or equal
>	*Intg Intg*	→	*Bool*	greater than
≥	*Intg Intg*	→	*Bool*	greater than or equal

5.4.4.3 Character

The Meta-IV name *Char* denotes the domain of ASCII character values. For the printable characters, there exist object representations which are enclosed in quotation marks, e.g. “a”, “Z”, “ ”.

Operators for Character:

Notation	Type			Operation
=	*Char Char*	→	*Bool*	equal
≠	*Char Char*	→	*Bool*	different
<	*Char Char*	→	*Bool*	less than
≤	*Char Char*	→	*Bool*	less than or equal
>	*Char Char*	→	*Bool*	greater than
≥	*Char Char*	→	*Bool*	greater than or equal

The relational operators are applied on the associated ASCII numerical values.

For the sake of readability, objects of the domain $Char^{+}$ may be represented by a sequence of characters enclosed in quotation marks, e.g. “abc” is the same as (“a”,“b”,“c”) (see section 5.4.6).

5.4.4.4 Quotation

The Meta-IV name *Quot* denotes the domain of quotations. They are distinct elementary objects and they are represented as any bold-face sequence of uppercase letters and digits, e.g. **ENVIRONMENT**, **REVERSE**.

Operators for Quotations:

Notation	Type	Operation
=	*Quot Quot* → *Bool*	equal
≠	*Quot Quot* → *Bool*	different

As opposed to other domains, objects of *Quot* may occur in domain definitions when only certain object(s) of *Quot* are possible in the given context, for example, in the abstract syntax of Z.100, *Originating-block*$_1$ is defined to be

1 *Originating-block*$_1$ = *Block-identifier*$_1$ | **ENVIRONMENT**

alternatively, *Originating-block*$_1$ could have been defined using *Quot*:

2 *Originating-block*$_1$ = *Block-identifier*$_1$ | *Quot*

however, using ENVIRONMENT in the domain definition is more precise, since this object is the only *Quot* value possible in that context.

5.4.4.5 Token

The Meta-IV name *Token* denotes the domain of tokens. This domain can be considered as consisting of a potentially infinite set of distinct elementary objects for which no representations are required.

Operators for Tokens:

Notation	Type	Operation
=	*Token Token* → *Bool*	equal
≠	*Token Token* → *Bool*	different

Example

Name$_1$ in the abstract syntax of Z.100 is defined to be

1 *Name*$_1$ :: *Token*

The only property needed for *Name*$_1$*s* during interpretation is equality. A *Name*$_1$ therefore consists of a *Token* value (the actual spelling of names is irrelevant).

5.4.4.6 Ellipsis

The Ellipsis domain (represented by ...) denotes an unspecified construct. It is used in domain definitions or in expressions:

- whenever the actual domain or expression is of no importance for the semantics; or
- whenever the elaboration of the domain or expression is outside the scope of the specification.

Example

Informal-text$_1$ in the abstract syntax of Z.100 is defined to be

1	*Informal-text*$_1$	::	…

Informal-text$_1$ cannot be interpreted using Meta-IV. *Informal-text*$_1$ therefore contains some further unspecified object.

5.4.5 Set Domains

A set domain is constructed by postfixing the element domain by the keyword **-set** (the dash is significant). For example:

2	*State-node*$_1$	::	*State-name*$_1$
			Save-signalset$_1$
			Input-node$_1$**-set**
			Spontaneous-transition$_1$**-set**
3	*Save-signalset*	::	*Signal-Identifier*$_1$**-set**

expresses that objects of the domain *State-node*$_1$ consist of a state name, a save signalset, which contains a set of signal identifiers, a set of input nodes and a set of spontaneous transitions. Set values can be constructed by using an explicit set constructor which is an expression list enclosed by braces, i.e.

{1, 3, 5, 1}

denotes an object of the domain *Intg*-**set** and it contains the three *Intg* values 1,3,5. A more usual form is the so-called implicit set constructor where the set includes all those elements which satisfy a certain condition (predicate). For example:

$\{i \in Intg \mid 0 \leq i \leq 5 \vee i \textbf{ mod 2} = 0\}$

defines the set

{0, 1, 2, 3, 4, 5, 6, 8, 10, 12, 14, 16, …}

It reads: The set of those values on the left hand side of the vertical bar (possibly qualified by a value or by a domain) for which the expression on the right hand side of the vertical bar holds.

The empty set is denoted by {}.

In the following explanation of the semantics of the operators on sets, *s* denotes the set {1,3,5}:

$\in$ Membership operator.

Test whether a given element of the element domain is contained in a set, that is, $1 \in s \equiv$ **true** and $2 \in s \equiv$ **false**.

$\notin$ Test whether a given element of the element domain is excluded in a set, that is, $1 \notin s \equiv$ **false** and $2 \notin s \equiv$ **true**.

$\cup$ Union operator.

Join two sets, that is, $\{2,3\} \cup s \equiv \{1,2,3,5\}$ and $s \cup s \equiv s$.

$\cap$ Intersection operator. Return the intersection of two sets, that is, $\{2,3\} \cap s \equiv \{3\}$ and $\{\} \cap s \equiv \{\ \}$.

\ Complement operator.

Exclude a given set of values from a set, that is, $s \setminus \{1,2\} \equiv \{3,5\}$ and $\{1,2\} \setminus s \equiv \{2\}$.

$\subset$ Proper subset operator.

Test whether the elements of a given set are contained in a set, that is, $\{1,5\} \subset s \equiv$ **true**, $s \subset \{1,5\} \equiv$ **false** and $s \subset s \equiv$ **false**.

$\subseteq$ Subset operator.

Test whether the elements of a given set are contained in or equal to a set, that is, $\{1,5\} \subseteq s \equiv$ **true**, $s \subseteq \{1,5\} \equiv$ **false** and $s \subseteq s \equiv$ **true**.

card Cardinality operator.

Return the number of elements in a set, that is, **card** s ≡ 3 and **card** {} ≡ 0.

union Distributed union operator.

The argument is a set of sets and the result is the union of all the sets contained in the argument, that is, **union** $\{s, \{5,6\},\{1,5,8\}\} \equiv s \cup \{5,6\} \cup \{1,5,8\} \equiv \{1,3,5,6,8\}$.

$=, \neq$ Test for equality and inequality of sets.

Example

In terms of Meta-IV expressions, the properties of the set operators $\notin$, $\cup$, $\cap$, $\subset$, $\subseteq$, **card** and **union** can be illustrated as follows:

$element \notin s1 = (\neg(element \in s1))$

$s1 \cup s2 = \{element \mid element \in s1 \vee element \in s2\}$

$s1 \cap s2 = \{element \mid element \in s1 \wedge element \in s2\}$

$s1 \setminus s2 = \{element \mid element \in s1 \wedge element \notin s2\}$

$s1 \subset s2 = (\forall element \in s1)\ (element \in s2) \wedge s1 \neq s2$

$s1 \subseteq s2 = (\forall element \in s1)\ (element \in s2)$

```
card s1 = (if s1 = { }
               then 0
               else (let element ∈ s1 in
                        1 + card (s1 \ {element })))
union s1 = {element | ∃set ∈ s1) (element ∈ set)}
```

The quantifiers ($\forall$ and $\exists$) are explained in section 5.6.

5.4.6 List Domains

A list or tuple domain is constructed by postfixing the element domain by a “*” in the case of a possibly empty list and otherwise by a “+”.

Example

4	$Signal\text{-}definition_1$	::	$Signal\text{-}name_1$
			$Sort\text{-}reference\text{-}identifier_1{}^*$

This domain definition expresses that a signal definition consists of a signal name and a possibly empty list of sort identifiers.

A list value can be constructed by using an explicit tuple constructor. This is an expression list enclosed in angular brackets, i.e.

$\langle 11, 12, 11, 13, 14\rangle$

denotes an object of the domain $Intg^+$ (or $Intg^*$) and it contains 5 ordered elements.

The empty list is denoted by $\langle\rangle$.

There are also implicit list constructors similar to those for sets. For instance, in the function *int-output-node* in the Dynamic Semantic we construct a tuple (*vall*) which contains the values of all the actual parameters (*exprl*) in an output node:

let *vall* = ⟨*eval-expression* (*exprl* [*i*])(*dict*) | 1 ≤ *i* ≤ **len** *exprl*⟩ **in**

which corresponds to an explicit enumeration of all the elements in the list:

```
let vall = ⟨eval-expression(exprl [1])(dict),
           eval-expression(exprl [2])(dict),
           eval-expression(exprl [3])(dict),
           …⟩ in
```

Note that the tuple brackets (⟨ and ⟩) have a different shape than the relational operators < and >.

In the following explanation of the semantics of the operators on lists *l* denotes the list ⟨11,12,11,13,14⟩:

hd	Return the first element (the **he**a**d** of a list). That is, **hd** *l* ≡ 11. The argument to **hd** must not be an empty list (⟨⟩).
tl	Return the list where the first element has been removed (return the **t**ai**l**). That is **tl** *l* ≡ ⟨12,11,13,14⟩.
[*i*]	Return element number *i* in a list. That is, *l*[3] ≡ 11 et *l*[5] ≡ 14. The index value must not be less than 1 or greater than the length of the list.
len	Return the length of a list. That is, **len** *l* ≡ 5.
elems	Return the set which consists of those elements which are in a list. That is, **elems** *l* ≡ {11,12,13,14}.
ind	Return the set of integer objects which are the legal index values for a list. That is, **ind** *l* ≡ {1,2,3,4,5}.
⌢	Concatenate two lists. That is *l* ⌢ ⟨0,1⟩ ≡ ⟨11,12,11,13,14,0,1⟩.
conc	Concatenate all those lists which are elements of the list given as argument. That is, **conc** ⟨⟨0,7⟩,*l*,⟨9⟩⟩ ≡ ⟨0,7,11,12,11,13,14,9⟩.
=, ≠	Test for equality and inequality of lists.

Example

In terms of Meta-IV expressions, the properties of the list operators **hd**, **tl**, **ind**, **elems** and **conc** can be illustrated as follows:

hd *l* = (*if l* = ⟨⟩ **then undefined else *l*** [1])
tl *l* = ⟨*l* [***i***] | 2 ≤ ***i*** ≤ **len *l***⟩
ind *l* = {***i*** | 1 ≤ ***i*** ≤ **len *l***}
elems *l* = {***l*** [***i***] | ***i*** ∈ **ind *l***}
conc *l* = (**if *l*** = ⟨⟩ **then** ⟨⟩ **else hd *l*** ⌢ **conc tl *l***)

5.4.7 Map Domains

A map Domain (i.e. a table) is constructed by specifying the domain of entry objects, followed by the $\underset{m}{\rightarrow}$ operator and followed by the domain of the objects contained in the mapping (the **range** values).

Example

5 *Entity-dict* = (*Identifier*$_1$ **PROCESS**) $\underset{m}{\rightarrow}$ *ProcessDD* ∪
(*Identifier*$_1$ **SERVICE**) $\underset{m}{\rightarrow}$ *ServiceDD* ∪
ENVIRONMENT $\underset{m}{\rightarrow}$ *Reachabilities* ∪
EXPIREDF $\underset{m}{\rightarrow}$ *Is-expiredF* ∪
PIDSORT $\underset{m}{\rightarrow}$ *Sort-identifier*$_1$ ∪
NULLVALUE $\underset{m}{\rightarrow}$ *Value* ∪
TRUEVALUE $\underset{m}{\rightarrow}$ *Value* ∪
FALSEVALUE $\underset{m}{\rightarrow}$ *Value* ∪

For enhancing overview of this example only a part of the definition of *Entity-dict* is shown here. The full definition of the *Entity-dict* mapping can be found in the Dynamic Semantics. It shows how the $\underset{m}{\rightarrow}$ operator is used and also that **composite** mappings can be constructed by using the domain merge operator ∪, that is, given a mapping of domain *Entity-dict*:

- we lookup in the mapping by applying an object of the unnamed tree (*Identifier*$_1$ **PROCESS**) and the result is an object of domain *ProcessDD*; or
- we lookup in the mapping by applying an object of the unnamed tree (*Identifier*$_1$ **SERVICE**) and the result is an object of domain *ServiceDD*; or
- we apply the *Quot* value **ENVIRONMENT** and the result is an object of domain *Reachabilities*; or
- we apply the *Quot* value **EXPIREDF** and the result is an object of domain *Is-expiredF*; or
- we apply the *Quot* value **PIDSORT** and the result is an object of domain *Sort-identifier*$_1$; or

- we apply the *Quot* value **NULLVALUE** and the result is an object of domain *Value*; or
- we apply the *Quot* value **TRUEVALUE** and the result is an object of domain *Value*; or
- we apply the *Quot* value **FALSEVALUE** and the result is an object of domain *Value*.

We can only apply a value if it previously has been put into the mapping object, as opposed to functions where the correspondence between argument values and result values are fixed and defined when the function is defined.

Mapping values can be constructed by using an explicit mapping constructor which is a list of pairs of entry values and range values enclosed in square brackets, i.e.

$[1 \mapsto \mathbf{D},$
$2 \mapsto \mathbf{AA},$
$4 \mapsto \mathbf{BB},$
$9 \mapsto \mathbf{ABC},$
$5 \mapsto \mathbf{XYZ}]$

denotes a mapping value of domain $\mathit{Intg} \underset{m}{\rightarrow} \mathit{Quot}$.

Also implicit mappings may be constructed. For example, the implicit mapping

$[a \mapsto b \mid a \in N_1 \wedge a * a = b]$

is equivalent to the infinite mapping

$[1 \mapsto 1,$
$2 \mapsto 4,$
$3 \mapsto 9,$
$\ldots \mapsto \ldots]$

In the following explanation of the semantics of the operators on mappings *m* denotes the first of the mapping specified explicitly above:

m(*entryvalue*)	Return a value from a mapping, that is, $m(1) \equiv \mathbf{D}$ and $m(9) \equiv \mathbf{ABC}$.
+	Overwrite a mapping with another mapping. This operator is not commutative, that is $m + [0 \mapsto \mathbf{XX}, 1 \mapsto \mathbf{B}] \equiv$ $[0 \mapsto \mathbf{XX}, 1 \mapsto \mathbf{B}, 2 \mapsto \mathbf{AA}, 4 \mapsto \mathbf{BB}, 9 \mapsto \mathbf{ABC}, 5 \mapsto \mathbf{XYZ}]$ whereas $[0 \mapsto \mathbf{XX}, 1 \mapsto \mathbf{B}] + m \equiv$ $[0 \mapsto \mathbf{XX}, 1 \mapsto \mathbf{D}, 2 \mapsto \mathbf{AA}, 4 \mapsto \mathbf{BB}, 9 \mapsto \mathbf{ABC}, 5 \mapsto \mathbf{XYZ}]$
\	Exclude a given set of entry values from a mapping, that is $m\backslash\{1,2,3\}$ is $[4 \mapsto \mathbf{BB}, 9 \mapsto \mathbf{ABC}, 5 \mapsto \mathbf{XYZ}]$
dom	Return the set which contains exactly those entry values which are present in a given mapping, that is $\mathbf{dom}\ m \equiv \{1,2,4,5,9\}$
rng	Return the set which contains exactly those range values which are contained in a given mapping, that is $\mathbf{rng}\ m \equiv \{\mathbf{D},\mathbf{AA},\mathbf{BB},\mathbf{ABC},\mathbf{XYZ}\}$
=, ≠	Test for equality and inequality of two mappings.
merge	From the given set of mappings, return the mapping which is constructed by merging all the mappings contained in the set, that is $\{m, [0 \mapsto \mathbf{WE}], [10 \mapsto \mathbf{D}]\} \equiv$ $[0 \mapsto \mathbf{WE}, 10 \mapsto \mathbf{D}, 1 \mapsto \mathbf{D}, 2 \mapsto \mathbf{AA}, 4 \mapsto \mathbf{BB}, 9 \mapsto \mathbf{ABC}, 5 \mapsto \mathbf{XYZ}]$ If any of the mappings contained in the set have overlapping entries, an arbitrary value among the possible values is chosen.

The empty mapping is denoted by [] (two square brackets very close to each other).

Example

In terms of Meta-IV expressions, the properties of the mapping operators \ , + and **merge** can be illustrated as follows:

$m1 \setminus s = [a \mapsto b \mid a \in \textbf{dom}\, m1 \setminus s \wedge m1(a) = b]$
$m1 + m2 = [a \mapsto b \mid (a \in \textbf{dom}\, m2 \wedge m2(a) = b) \vee (a \in \textbf{dom}\, m1 \setminus \textbf{dom}\, m2 \wedge m1(a) = b)]$

```
merge m1 = (if m1 = { }
                then []
                else (let element ∈ m1 in
                        element + merge m1 \ {element }))
```

5.4.8 Pid Domains

A Pid domain (corresponding to the Pid sort in SDL) is constructed by means of the II symbol. Optionally it may be qualified by the processor type to indicate which kind of Pid values the domain denotes, for example:

6	*Import-Create*	::	Π (*input-port*)

The *Inport-Created* domain (defined in the Dynamic Semantics) contains Pid objects qualified by the processor type *input-port.* The Meta-IV Pid values should not be confused with the SDL Pid values which in SDL are *Ground-term*$_1$*s,* i.e. The domain of the SDL Pid values are defined in the Dynamic Semantics to be:

7	*Pid-Value*	=	*Value*
8	*Value*	=	*Ground-term*$_1$

Meta-IV Pid values are created when applying the **start** statement/expression. It corresponds to the create request action in SDL. For example, when the *system* processor creates an instance of a *timer* processor with the actual parameters *timeinf* and *dict,* it looks like:

Example

start *timer* (*timeinf*)(*dict*)

When the start construct is used as an expression, it creates a processor instance and returns the Meta-IV Pid value of this instance (corresponding to the **offspring** value in SDL). For example, when a *process-set-admin* processor starts an *input-port* processor:

start *input-port*(*offspring, dict* (**EXPIREDF**)*, delayf,* **self**)

an instance of the *input-port* processor is created and the resulting Meta-IV Pid value is used by the *process-set-admin* for identifying the *input-port.* The parameters *offspring, dict(*EXPIREDF*), delayf* and **self** are given to the created instance. Like in SDL, an instance may access its own Pid value by using the **self** expression.

Communication is performed by the synchronous communication primitives **input** and **output.** In the output construct, we can either choose to communicate with a specific processor instance or we can choose to communicate with an unspecified instance of a specific processor type.

Example

output mk-*Some-tree* (*somevalue, someothervalue, …*) **to** *p*

where *p* either denotes a Pid value or *p* is the name of a processor type. The values sent by the processor are usually encapsulated in a named tree object (of some **communication** domain) and such trees can therefore be equated to the signal concept in SDL, i.e. *Some-tree* can be regarded as a signal.

In the input construct, we both specify the communication object we want to receive and the action which should be taken when the object is received. In addition, we may specify a name which after the reception of the object denotes the Pid value of the sending processor (corresponding to **sender** in SDL) or which restricts the possible senders, i.e.

input mk-*Some-tree*(*a, b, d*) **from** *p*
  ⇒ */∗ some statements or an expression ∗/*

After reception of *Some-tree, a,b* and *d* will denote the values conveyed by *Some-tree* and for *p* there are three possible interpretations:

- If *p* is a processor type name, then the input should be received from an instance of that particular processor type.
- If *p* is a name which is not already defined, then this occurrence is the defining occurrence of the name and it is visible in the expression or statements which follow the input clause. It denotes the Meta-IV Pid value of the sender.
- If *p* is an expression, then it must be of the type Π and the input will be received from the processor instance denoted by the expression.

If one of several inputs may be received, a number of input constructs separated by comma are specified and the number is enclosed by braces, i.e.

{**input mk**-*Some-tree*(*a, b, d*) **from** *p*

  ⇒ */* some statements or an expression */,*

input mk-*Some-other-tree*(*a, b, d*) **from** *p*

  ⇒ */* some statements or an expression */* }

In some cases we may want to specify that either an input or an output should be made, depending on which communication first is possible (not applicable in SDL due to the fact that in SDL communication is asynchronous). In such cases, output constructs are included in the set of communication events, i.e.

{**input mk**-*Some-tree*(*a, b, d*) **from** *p*

  ⇒ */* some statements or an expression */,*

input mk-*Some-other-tree*(*a, b, d*) **from** *p*

  ⇒ */* some statements or an expression */,*

output mk-*Something*(*/* expression */, /* expression */*) to *pi* }

Often the **cycle** construct is used in conjunction with input and output, if the communication should be repeated, i.e.

cycle {**input mk**-*Some-tree*(*a, b, d*) **from** *p*

    ⇒ */* some statements or an expression */,*

    input mk-*Some-other-tree*(*a, b, d*) **from** *p*

    ⇒ */* some statements or an expression */,*

    output mk-*Something*(*/* expression */, /* expression */*) **to** *pi* }

which means that after a communication event, the processor instance will take the appropriate action and then start waiting for a new event to happen.

5.4.9 Reference Domains

When a Meta-IV variable is declared by

dcl v **type** *Intg*;

a Meta-IV storage location is allocated and the variable (v) will denote a reference to the location. When the content of the location is accessed, the **c** operator (**c**ontents operator) is used as shown earlier. When the variable is used without the contents operator, the result is a value of the **ref** domain, that is, a reference to the storage location. **ref** domains are specified by using the keyword **ref**, followed by the appropriate domain. For example:

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

The variable descriptor includes a reference to the domain *Stg*. The *VarDD* descriptor is defined in the Dynamic Semantics and it is described further in the associated annotations.

5.4.10 Optional Domains

The square brackets which are extensively used in the domain definitions mean optionality.

Example

```
10  Signal-definition₁                    ::  Signal-name₁
                                              Sort-reference-identifier₁*
                                              [Signal-refinement₁]
```

expresses that in objects of the tree *Signal-definition*$_1$, the object of the domain *Signal-refinement* may or may not be present. If it is not present, the field will contain the type-less value **nil**.

Example

```
(let mk-Signal-definition₁(name, sort, refinement) = /* some Signal-definition₁ object */ in
 if refinement = nil then
    /* some actions */

   else
    (let mk-Signal-refinement₁ (...) = refinement in
      /* some other actions using the signal refinement */))
```

5.5 The let and def Constructs

As shown earlier, the **let** construct can be used for composing and decomposing objects. The **let** construct is more generally used whenever we want some name to denote some specific object (often it is just in order to avoid too complicated and unreadable expressions). The names occurring on the left hand side of the equal sign in the **let** construct are the defining occurrences (except for domain names which must always be defined somewhere in a domain definition). An introduced name can also be used on the right hand side of the equal sign (the name is then recursively defined) and in the expression which follows the **let** construct. In the example below, *name1* is visible (i.e. may be used) in */*expression1*/*, */*expression2*/*, */*expression3*/* and */*expression4*/*, *name2* is visible in */*expression2*/*, */*expression3*/* and */*expression4*/* and *name3* is visible in */*expression3*/* and */*expression4*/.* For the sake of restricting the visibility of the names introduced by a **let**, the **let** construct is enclosed by parenthesis. In the example above, a signal refinement constitutes an expression and it starts with left parenthesis because a **let** construct is used.

There are two ways of specifying a sequence of **let**s:

```
let name1 = /* expression1 */ in
let name2 = /* expression2 */ in
let name3 = /* expression3 */ in
/* expression4 */
```

or

```
let name1 = /* expression1 */,
    name2 = /* expression2 */,
    name3 = /* expression3 */ in
/* expression4 */
```

The first form showing three **let**s is usually used in the FD when the order is important, that is if */*expression2*/* uses *name1* and if */*expression3*/* uses *name2* whereas the second form is used when the various **let**s are independent.

There are several different forms of a **let** construct. We have already seen how it can be used for decomposing objects. Other relevant forms are:

```
let name ∈ setorname1 in
/* some expression using name */
let name be s.t. /* condition using name */ in
/* some expression using name */
let name ∈ setorname2 be s.t. /* condition using name */ in
/* some expression using name */
let name (parameters) = /* function body */ in
/* some expression applying name */
```

The first form reads: Extract an arbitrary value belonging to the set or belonging to the domain denoted by *setorname1* and denote the value by *name.*

The second form reads: Construct a value, i.e. let *name* **be s**uch **t**hat the specified condition holds for the value.

The third form is a combination of the two previous forms, where both restrictions apply. If no such value exists, the specification is erroneous.

The fourth form reads: Construct a local function (called *name*) which has some formal parameters (*parameters*) and a body.

Example

Define the square root of 3:

let $r \in$ *Real* **be s.t**. $r > 0 \wedge r * r =$ **3 in**

Example

Define the factorial function where n is the formal parameter:

let *fact* $(n) =$ **if** $n < 0$ **then error else if** $n = 0$ **then** 1 **else** $n *$ *fact* $(n - 1)$ **in**

When defining a name for an object which is constructed by referring to the global state (i.e. if the name is defined in terms of an imperative expression) the **def** notation is used instead of the **let** notation, that is, the keyword **let** is replaced by the keyword **def**, the equal symbol is replaced by a colon and the keyword **in** is replaced by a semicolon (because the **def** construct is used in statement context, see section 5.7). For instance, if we want to denote a created processor instance value by a name, we write:

(**def** *pid* : **start** *input-port*(*somevalue*);
/ some statements using the pid value */*)

or if we want to decompose the result of an imperative function we write:

(**def mk**-*Some-tree*(*a, b*)*: some-imperative-function*(...);
/ some statements using a and b */*)

There also exist a **def** version of the “be such that” construct:

(**def** $r \in$ *Real* **s.t**. $r > 0 \wedge r * r =$ **c** v1;
*/*some statements using r */*)

where we use **def** because we use a variable (v1) in the evaluation of *r.* It reads: Define a *Real* value *r* **s**uch **t**hat the square of *r* equals the contents of the variable v1.

It should be noted that the names introduced in **let** and **def** are **not** variables. They are names representing a specific value and it is not allowed to assign a new value to such names.

5.6 Quantification

Meta-IV also provides the mathematical quantifiers: the **universal** quantifier represented by the symbol ∀, the **existential** quantifier represented by the symbol ∃ and the **unique** quantifier represented by the symbol ∃!. These quantifiers may be used in **quantified** expressions which return the boolean value true if a specified condition (a **predicate)** on an object is satisfied.

Example

identifiers-defined-on-system-level(*p*) $\triangleq$

1 (∀**mk**-*Identifier*$_1$(*q*,) ∈ *p*) (**len** $q = 1$)

type: *Identifier*$_1$-**set** → *Bool*

This function returns true if and only if for all identifiers (*Identifier*$_1$) in the set *p* it holds that the length of its qualifier (*q*) *is* equal to 1 (the second pair of parenthesis encloses the predicate expression).

Example

one-identifier-defined-on-system-level$(p) \triangleq$

```
1  (∃mk-Identifier_1(q,) ∈ p) (len q = 1)
```

type: $Identifier_1$-**set** $\rightarrow$ *Bool*

This function returns true if and only if there exists at least one identifier ($Identifier_1$) in the set p for which the length of its qualifier (q) *is* equal to 1.

Example

exactly-one-identifier-defined-on-system-level$(p) \triangleq$

```
1  (∃!mk-Identifier_1(q,) ∈ p) (len q = 1)
```

type: $Identifier_1$-**set** $\rightarrow$ *Bool*

This function returns true if and only if there exists exactly one ($Identifier_1$) in the set p for which the length of its qualifier (q) *is* equal to 1.

Alternatively, we can choose to decompose the identifier in the predicate expression instead of in the quantification, that is:

identifiers-defined-on-system-level$(p) \triangleq$

```
1  (∀p'∈ p)
2  ((let mk-Identifier_1(q,) = p' in
3  len q = 1))
```

type: $Identifier_1$-**set** $\rightarrow$ *Bool*

NOTE – Apostrophe and dash are legal characters in Meta-IV names.

5.7 Auxiliary Statements

- Identity statement

  The keyword **I** indicates an empty statement, i.e. a statement which does not do anything.

- Undefined statement/expression

  The keyword **undefined** indicates that no semantics can be given.

- Return statement

  The keyword **return** followed by an expression terminates the elaboration of an imperative function and the result is the given expression.

- Error statement/expression.

  The keyword **error** indicates in the FD a dynamic SDL error.

- Assign statement.

  Like in SDL. The contents operator (**c**) is not used when assigning to variables.

- For and while statement.

  Same (well-known) concept as in CHILL. The statements to be repeated are enclosed in parenthesis.

- Trap and exit statement/expression.

  Trap (handle) any exits caused by an exit statement/expression. If an argument is given to the exit statement, it is only trapped if the expression given matches the value given in the trap exit statement. A special version of the trap exit mechanism: the **tixe** construct have been used in the functions *int-process-graph* and *int-procedure-graph. The* **tixe** construct is explained in the associated annotations.

5.8 Deviations from the notation used in the Formal Definition of CHILL

- In the formal definition of CHILL, the predefined domain names consist of boldface uppercase letters (e.g. **BOOL**, **INTG**) and names denoting semantic domains may consist of uppercase letters only.

  In the formal definition of SDL, all domain names are in italic, the first letter is in uppercase and they contain at least one lowercase letter.

- In the formal definition of CHILL, all objects are finite.

  In the formal definition of SDL, objects may be infinite. The semantics of some of the operators are not well-defined when applied on such objects, e.g. operators like cardinality and equality have not been used on potentially infinite objects.

  In addition, a special constant *infinite* has been used in *transform-process* in Annex F.2 for representing the "unbounded number of instances" in AS_1.

- In the formal definition of SDL, the Meta-IV notation has been extended to include the elementary domain *Char* and the character strings objects (see section 5.4.4.3).

- In the *path* processor in Annex F.3, a so-called "output guard" has been used. The concept is described in the annotations attached to the *Path* processor as well as in [4].

5.9 Example: Demon game specified in Meta-IV

In the following, it is shown how Meta-IV can be used for defining the semantics of Demon game. For further details about Demon game, refer to Z.100/2.9.

Communication *demon* $\rightarrow$ *monitor* and *monitor* $\rightarrow$ *game*

11 *Bump* :: ()

Communication *user* $\rightarrow$ *monitor*

12 *Newgame* :: ()

Communication *game* $\rightarrow$ *monitor*

13 *Gameover* :: Π

Communication *monitor* $\rightarrow$ *game*

14 *Gameoverack* :: ()

Communication *game* $\rightarrow$ *user*

15 *Gameid* :: ()
16 *Win* :: ()
17 *Lose* :: ()
18 *Score* :: *Intg*

Communication *user* $\rightarrow$ *game*

19 *Probe* :: ()
20 *Result* :: ()
21 *Endgame* :: ()

int-demon-game() $\triangleq$

1 **start** *monitor* ()

type: () $\Rightarrow$ ()

```
monitor processor () ≜

(dcl userset := { } type Π-set,
       gameset := { } type Π-set;
cycle (input mk-Newgame () from sender)
            ⇒ if sender ∉ c userset then
                   (def offspring : start game (sender);
                    gameset := c gameset ∪ {offspring};
                    userset := c userset ∪ {sender})
                  else
                   I,
         input mk-Gameover (player) from sender
            ⇒ (gameset := c gameset \ {sender};
                    userset := c userset \ {player};
                    output mk-Gameoverack( ) to sender),
         input mk-Bump( ) from demon
            ⇒ for all pid ∈ gameset do
                    output mk-Bump( ) to pid))

type: () ⇒
```

```
game processor (player) ≜

(dcl count := 0 type Intg;
dcl even := true type Bool;
output mk-Gameid( ) to player;
cycle (input mk-Probe( ) from user
            ⇒ if c even
                    then (output mk-Win( ) to player;
                             count := c count + 1)
                    else (output mk-Lose( ) to player;
                           count := c count – 1),
         input mk-Result( ) from user
            ⇒ output mk-Score(count) to player,
         input mk-Endgame( ) from user
            ⇒ (output mk-Gameover(player) to monitor;
                   input-mk-Gameoverack( ) from monitor
                      ⇒ stop),
         input-mk-Bump( ) from monitor
            ⇒ even := ¬c even))

type: Π ⇒ ()
```

References

[1] BJØRNER (D.) and JONES (C. B.): Formal specification and software development, *Prentice-Hall Publ.*, 1982.

[2] *The Formal Definition of CHILL*, CCITT Manual, ITU, Geneva 1981.

[3] FOLKJÆR (P.) and BJØRNER (D.): A formal model of a generalized CSP-like language, *IFIP 8th World Computer Conference*, Proceedings, North Holland Publ. 1980.

[4] HOARE (C. A. R.): Communicating Sequential Processes, *Prentice-Hall*, 1985.