INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

Z.100

Annexes C and D
(03/93)

SERIES Z: PROGRAMMING LANGUAGES
Specification and Description Language (SDL)

Initial algebra model and SDL predefined data

ITU-T Recommendation Z.100 – Annexes C and D
Superseded by a more recent version

(Previously "CCITT Recommendation")

ITU-T Z-SERIES RECOMMENDATIONS

PROGRAMMING LANGUAGES

Specification and Description Language (SDL)	**Z.100–Z.109**
Criteria for the use and applicability of formal Description Techniques	Z.110–Z.199
ITU-T High Level Language (CHILL)	Z.200–Z.299
MAN-MACHINE LANGUAGE	Z.300–Z.499
General principles	Z.300–Z.309
Basic syntax and dialogue procedures	Z.310–Z.319
Extended MML for visual display terminals	Z.320–Z.329
Specification of the man-machine interface	Z.330–Z.399
Miscellaneous	Z.400–Z.499

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

ITU-T RECOMMENDATIONS SERIES

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

Superseded by a more recent version

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.

Annexes C and D to ITU-T Recommendation Z.100 were prepared by ITU-T Study Group X (1988-1993) and were approved by the WTSC (Helsinki, March 1-12, 1993).

NOTE

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.

3 Due to the specialized nature of the subject matter contained herein, this annex is published in English only.

CONTENTS

Superseded by a more recent version

ANNEX C

(to Recommendation Z.100)

Initial Algebra Model

C Informal Description

C.1 Introduction

The definition of data in SDL is based ont he data kernel defined in § 5.2 of the Recommendation. Operators and values need to be given some further meaning in addition to the former definition so interpretation can be given to expressions. For example, expressions used in continuous signals, enabling conditions, procedure calls, remote procedure calls, output actions, create requests, assignment statements, set and reset statements, export statements, import statements, decisions, and viewing.

The necessary additional meaning is given to expressions by using the initial algebra formalism which is explained in § C.1 to § C.6 below.

At any point in an SDL specification the last data type hierarchically defined will apply, but there is a set of sorts visible. The set of sorts is the union of all sorts at levels hierarchically above the place in question as explained in § 5.2 of the Recommendation.

(In this section the symbol = is used as an equation equivalence symbol wheras in SDL symbol == is used for equation equivalence so that the symbol = can be used for the equality operator. The symbol = is used in this section as it is the conventional symbol used in published work on initial algebras.)

The meaning and interpretation of data based on initial algebra is explained in three stages:

a) Signatures

b) Terms

c) Values

C.1.1 Representations

The idea that different notations can represent the same concept is commonplace. For instance, it is generally accepted that positive Arabic numbers (1, 2, 3, 4, …) and Roman numerals (I, II, III, IV, …) represent the same set of numbers with the same properties. As another example, it is quite usual to accept that prefix functional notation (plus(1,1)), infix notation (1+1) and reverse polish notation (1 1 +) can all represent the same operator. Furthermore, different users may use different names (perhaps because they are using different languages) for the same concepts so that the pairs {true, false}, {T, F}, {0,1}, {vrai, faux} could be different representations of the Boolean sort.

What is essential is the abstract relationship between identities and not the concrete representation. Thus for numerals what is interesting is the relationship between 1 and 2 which is the same as the relationship between I and II. Also for operators what is of interest is the relationship between the operator identity and other operator identities and the list of arguments. Concrete constructions such as brackets which allow us to distinguish between (a+b)*c and a+(b*c) are only of interest so that the underlying abstract concept can be determined.

These abstract concepts are embodied in an abstract syntax of the concept which may be realised by more than one concrete syntax. For example, the following two concrete examples both describe the same data type properties but in different concrete syntax.

```
newtype bool
literals true, false;
operators "not"  :bool  ->bool;
axioms
  not(true)        ==  false;
  not(not(a))      ==  a;
endnewtype bool;
```

```
newtype int literals zero,  one;
operators plus   :  int,int  ->  int;
          minus  :  int,int  ->  int;

axioms
  plus(zero,a)              ==  a;
  plus(a,b)                 = =  plus(b,a);
  plus(a,plus(b,c))         = =  plus(plus(a,b),c);
  minus(a,a)                = =  zero;
  minus(a,zero)             = =  a;
  minus(a,minus(b,c))       = =  minus(plus(a,c),b);
  minus(minus(a,b),c)       = =  minus(a,plus(b,c));
  plus(minus(a,b),c)        = =  minus(plus(a,c),b);
endnewtype int;

newtype tree literals nil;
 operators
  tip         :  int         ->  tree;
  isnil       :  tree        ->  bool;
  istip       :  tree        ->  bool;
  node        :  tree,tree   ->  tree;
  sum         :  tree        ->  int;
 axioms
  istip(nil)                ==  false;
  istip(tip(i))             ==  true;
  istip(node(t1,t2))        ==  false;
  isnil(nil)                ==  true;
  isnil(tip(i))             ==  false;
  isnil(node(t1,t2))        ==  false;
  sum(node(t1,t2))          ==  plus(sum(t1),sum(2));
  sum(tip(i))               ==  i;
  sum(nil)                  ==  zero;
endnewtype tree;
```

EXAMPLE 1

```
type bool is
sorts bool
opns      true   :          ->   bool
          false  :          ->   bool
          not    :  bool    ->   bool
eqns  ofsort  bool  forall  a:bool
    not(true)      =  false;
    not(not(a))    =  a;
endtype

type int is bool
sorts int
opns      zero   :              ->  int
          one    :              ->  int
          plus   :  int,int     ->  int
          minus  :  int,int     ->  int
eqns  ofsort  int  forall  a,b,c:int
  plus(zero,a)              =  a;
  plus(a,b)                 =  plus(b,a);
  plus(a,plus(b,c))         =  plus(plus(a,b),c);
  minus(a,a)                =  zero;
  minus(a,zero)             =  a;
  minus(a,minus(b,c))       =  minus(plus(a,c),b);
  minus(minus(a,b),c)       =  minus(a,plus(b,c));
  plus(minus(a,b),c)        =  minus(plus(a,c),b)
endtype
```

```
type tree is int
sorts tree
opns    nil    :              -> tree
        tip    : int          -> tree
        isnil  : tree         -> bool
        istip  : tree         -> bool
        node   : tree,tree    -> tree
        sum    : tree         -> int
eqns    ofsort bool forall i:int, t1,t2:tree
        istip(nil)            =  false;
        istip(tip(i))         =  true;
        istip(node(t1,t2))    =  false;
        isnil(nil)            =  true;
        isnil(tip(i))         =  false;
        isnil(node(t1,t2))    =  false
  ofsort int forall i:int, t1,t2:tree
        sum(node(t1,t2))      =  plus(sum(t1),sum(t2));
        sum(tip(i))           =  i;
        sum(nil)              =  zero
endtype
```

EXAMPLE 2

This example is used for illustration. Initially the definition of sorts and literals will be considered. It should be noted that literals are considered to be a special case of operators, that is operators without parameters.

We can introduce some sorts and literals in the first form by

```
newtype int literals zero, one; . . .
newtype bool literals true, false; . . .
newtype tree literals nil; . . .
```

or in the second form by

```
. . .
sorts bool
opns    true   :  -> bool
        false  :  -> bool

. . .
sorts int
opns    zero   :  -> int
        one    :  -> int

. . .
sorts tree
opns    nil    :  -> tree
. . .
```

In the following the second form only will be used as that is closest to the formulation used in many publications on initial algebra. It should be noted that the form of terms is the same in both cases and the most significant difference is the way in which literals are introduced. It should be remembered that it is necessary to adopt a concrete notation to communicate the concepts, but the meaning of the algebras is independent of the notation so that systematic renaming of names (retaining the same uniqueness) and a change from prefix to polish notation will not change the meaning defined by the type definitions.

C.2 Signatures

Associated with each sort will be one or more operators. Each operator has an operator functionality; that is, it is defined to relate one or more input sorts to a result sort.

For example, the following operators can be added to the sorts defined above:

```
. . .
sorts bool
opns      true    :              -> bool
          false   :              -> bool
          not     : bool         -> bool
. . .
sorts int
opns      zero    :              -> int
          one     :              -> int
          plus    : int,int      -> int
          minus   : int,int      -> int
. . .
sorts tree
opns      nil     :              -> tree
          tip     : int          -> tree
          isnil   : tree         -> bool
          istip   : tree         -> bool
          node    : tree,tree    -> tree
          sum     : tree         -> int
. . .
```

The signature of the type which applies is the set of sorts, and the set of operators (both literals and operators with parameters) which are visible.

A signature of a type is called complete (closed) if for every operator in the signature, the sorts of the functionality of the operator are included in the set of sorts of the type.

C.3 Terms and expressions

The language of interest is one which allows expressions which are variables, literals or operators applied to expressions. A variable is a data object which is associated with an expression. Interpretation of a variable can be replaced with interpretation of the expression associated with the variable. In this way variables can be eliminated so that interpretation of an expression can be reduced to the application of various operators to literals.

Thus, on interpretation an open expression (an expression involving variables) becomes a closed expression (an expression without variables) by providing the open expression with actual arguments (that is, closed expressions).

A closed expression corresponds to a ground term.

The set of all possible ground terms of a sort is called the set of ground terms of the sort. For example, for bool as defined above the set of ground terms will contain

{true, false, not(true), not(false), not(not(true)), …}

It can be seen that even for this very simple sort the set of ground terms is infinite.

C.3.1 Generation of terms

Given a signature of a type, it is possible to generate the set of ground terms for that type.

The set of literals of the type are considered to be the basic set of ground terms. Each literal has a sort, therefore each ground term as a sort. For the type being defined above, this basic set of ground terms will be

{zero, one, true, false, nil}

For each operator in the set of operators for the type, ground terms are generated by substituting for each argument all previously generated ground terms of the correct sort for that argument. The result sort of each operator is the sort of the ground term generated by that operator. The resulting set of ground terms is added to the existing set of ground terms to generate a new set of ground terms. For the type above, this is

{zero,	one,	true,	false, nil,
plus(zero,zero),	plus(one,one),	plus(zero,one),	plus(one,zero),
minus(zero,zero),	minus(one,one),	minus(zero,one),	minus(one,zero),
not(true),	not(false),	tip(zero),	tip(one),
isnil(nil),	istip(nil),	node(nil, nil),	sum(nil)}

This new set of ground terms is then taken as the previous set of ground terms for a further application of the last algorithm to generate a further set of ground terms. This set of ground terms will include

{zero, one, true, false, nil,
plus(zero,zero), plus(one,one), plus(zero,one), plus(one,zero), . . .
plus(zero,plus(zero,zero)), plus(zero,plus(one,one)), . . .
plus(zero,sum(nil)), . . .

isnil(node(nil,nil)), istip(node(nil,nil)), node(nil,node(nil,nil)),

. . . , sum(node(nil,nil)) }

This algorithm is applied repeatedly to generate all possible ground terms for the type which is the set of ground terms for the type. The set of ground terms for a sort is the set of ground terms of the type which have that sort.

Normally generation will continue indefinitely yielding an infinite number of terms.

C.4 Values and algebras

Each term of a sort represents a value of that sort. It can be seen from above that even a simple sort such as bool has an infinite number of terms and hence an infinite number of values, unless some definition is given of how terms are equivalent (that is, represent the same value). This definition is given by equations defined on terms. In the absence of istip and isnil the sort bool can be limited to two values by the equations

not(true) = false;
not(false) = true

Such equations define terms to be equivalent and it is then possible to obtain the two equivalent classes of terms

{true, not(false), not(not(true)), not(not(not(false))), . . . }
{false, not(true), not(not(false)), not(not(not(true))), . . . }

Each equivalence class then represents one value and members of the class are different representations of the same value.

Note that unless they are defined equivalent by equations, terms are non-equivalent (that is, they do not represent the same value).

An algebra defines the set of terms which satisfies the signature of the algebra. The equations of the algebra relate terms to one another.

In general there will be more than one representation for each value of a sort in an algebra.

An algebra for a given signature is an initial algebra if and only if any other algebra which gives the same properties for the signature can be systematically transformed onto the initial algebra. (Formally such a transformation is known as a homomorphism.)

Providing not, istip and isnil always produce values in the equivalence classes of true and false, then an initial algebra for bool is the pair of literals

{true, false}

and no equations.

C.4.1 Equations and quantification

For a sort such as bool, where there are only a limited number of values, all equations can be written using only ground terms, that is terms which only contain literals and operators.

When a sort contains many values, writing all the equations using ground terms is not practical and for sorts with an infinite number of values (such as integers), such explicit enumeration becomes impossible. The technique of writing quantified equations is used to represent a possibly infinite set of equations by one quantified equation.

A quantified equation contains value identifiers in terms. Such terms are called composite terms. The set of equations with only ground terms can be derived from the quantified equation by systematically generating equations with each value identifier substituted in the equation by one of the ground terms of the sort of the value identifier. For example:

```
for all b : bool  not(not(b)) = b
```

represents

```
not(not(true))    = true;
not(not(false))   = false
```

An alternative set of equations for bool can now be taken as

```
for all b : bool
  not(not(b))     = b;
  not(true)       = false
```

When the sort of the quantified value identifier is obvious from context, it is usual practice to omit the clause defining the value identifier so that the example becomes

```
not(not(b))       = b;
not(true)         = false
```

C.5 Algebraic specification and semantics (meaning)

An algebraic specification consists of a signature and sets of equations for each sort of that signature. These sets of equations induce equivalence relations which define the meaning of the specification.

The symbol = denotes an equivalence relation that satisfies the reflexive, symmetric and transitive properties and the substitution property.

The equations given with a type allow terms to be placed into equivalence classes. Any two terms in the same equivalence class are interpreted as having the same value. This mechanism can be used to identify syntactically different terms which have the same intended value.

Two terms of the same sort, TERM1 and TERM2, are in the same equivalence class if

a) there is an equation TERM1=TERM2, or

b) one of the equations derived from the given set of quantified equations is TERM1=TERM2, or

c) i) TERM1 is in an equivalence class containing TERMA, and

   ii) TERM2 is in an equivalence class containing TERMB, and

   iii) there is an equation or an equation derived from the given set quantified equations such that TERMA = TERMB, or

d) by substituting a sub-term of TERM1 by a term of the same class as the sub-term producing a term TERM1A it is possible to show that TERM1A is in the same class as TERM2.

By applying all equations the terms of each sort are partitioned into one or more equivalence classes. There are as many values for the sort as there are equivalence classes. Each equivalence class represents one value and every member of a class represents the same value.

C.6 Representation of values

Interpretation of an expression then means first deriving the ground term by determining the actual value of variables used in the expression at the point of interpretation, then finding the equivalence class of this ground term. The equivalence class of this term determines the value of the expression.

Meaning is thus given to operators used in expressions by determining the resultant value given a set of arguments.

It is usual to choose a literal in the equivalence class to represent the value of the class. For instance, bool would be represented by true and false, and natural numbers by 0, 1, 2, 3, etc. When there is no literal then usually a term of the lowest possible complexity (least number of operators) is used. For instance, for negative integers the usual notation is –1, –2, –3, etc.

Appendix A

The formal model of non-parameterised data types[1)]

A.1 Many-sorted algebras

A **many-sorted algebra** A is a 2-tuple <D,O> where

a) D is set of sets, and the elements of D are referred to as the **data carriers** (of A); the elements of a data-carrier **dc** are referred to as **data-values**; and

b) O is a set of total functions, where the domain of each function is a Cartesian product of data carriers of A and the range of one of the data carriers.

A.2 Semantics of data type definitions

A.2.1 General concepts

A.2.1.1 Signature

A **signature** SIG is a tuple <S,OP> where

a) S is a set of **sort-identifiers** (also referred to as sorts); and

b) OP is a set of **operators**.

An operator consists of an **operation-identifier** op, a list of (argument) sorts w with elements in S, and a (range) sort s ∈ S. This is usually written as op:w→s. If w is equal to the empty list, the op:w→s is called a **null-ary** operator or **constant symbol** of sort s.

A.2.1.2 Signature morphism

Let SIG_1=<S_1,OP_1> and SIG_2=<S_2,OP_2> be signatures. A **signature morphism** g:SIG_1→SIG_2 is a pair of mappings

$$g = <gs:S_1 \rightarrow S_2, gop:OP_1 \rightarrow OP_2>$$

such that for all e-opid$_1$ = <opidf$_1$, <gs(e-sidf$_1$), …, gs(e-sidf$_k$)>, gs(e-res), pos > ∈ OP_1

$$gop(\text{e-opid}_1) = <\text{opidf}_2, <(\text{e-sidf}_1), \ldots, (\text{e-sidf}_k)>, (\text{e-res}), \text{pos} >$$

for some operation-identifier opidf$_2$.

A.2.1.3 Terms

Let V be any set of variables and let <S,OP> be a signature. The sets TERM(OP,V,s) of **terms** of sort s ∈ S with operators in OP and variables in V, are defined inductively by the following steps:

a) each variable x:s ∈ V is in TERM(OP,V,s);

---

1) The text of this Appendix has been agreed between CCITT and ISO as a common formal description of the initial algebra model for abstract data types. As well as appearing in this Recommendation, this text (with appropriate terminology, typographical and numbering changes) also appears in ISO IS8807. §§ A.1, A.2.1.1, A.2.1.2, A.2.1.3, A.2.1.4, A.2.1.5, A.2.1.6, A.3, A.4.1, A.4.2, A.4.3, A.4.4, A.4.5, and A.4.6, of this Appendix appear in §§ 5.2, 7.2.2.1, 7.3.2.8, 7.2.2.2, 7.2.2.3, 7.2.2.4, 7.2.2.5, 4.7, 7.4.2.1, 7.4.2.2, 7.4.3, 7.4.3 and 7.4.4 of IS8807 respectively. The terminologies **sort-identifier, operator, variable-identifier, variable, algebraic specification SPEC** and **operations** of this Appendix are replaced by **sort-variable, operation-variable, value-variable, value-variable, data presentation pres** and **functions** respectively in IS8807.

b) each null-ary operator op ∈ OP with res(op)= s is in TERM(OP,V,s);

c) if the terms t_i of sort s_i are in TERM(OP,V,s_i) for i=1,…,n, then for each op∈ OP with arg(op) = <s_1,…,s_n> and res(op)=s, op(t_1,…,t_n) is in TERM(OP,V,s).

If term t is an element of TERM(OP,V,s) then s is called the sort of t, denoted as sort(t). The set TERM(OP,s) of **ground terms** of sort s ∈ S is defined as the set TERM(OP,{},s).

A.2.1.4 Equations

An **equation** of sort s with respect to a signature <S,OP> is a triple <V,L,R> where

a) V is a set of variable-identifiers; and

b) L,R ∈ TERM(OP,V,s); and

c) s ∈ S.

An equation e'=<{},L',R'> is a **ground instance** of an equation e=<V,L,R>, if L',R' can be obtained from L,R for each variable v:s in V, replacing all occurrences of that variable in L,R by the same ground term with sort s.

The notation L=R is used for the ground instance <{},L,R> of an equation.

NOTE – Also an equation <V,L,R> may be written L=R if no semantical complications are thus introduced.

A.2.1.5 Conditional equations

A **conditional equation** of sort s with respect to the signature <S, OP> is a triple <V,Eq,e>, where

a) V is a set of variable-identifiers; and

b) Eq is a set of equations with respect to <S, OP>, with variables in V; and

c) e is an equation of sort s with respect to <S, OP>, with variables in V.

A.2.1.6 Algebraic specifications

An **algebraic specification** SPEC is a triple <S,OP,E> where

a) <S,OP> is a signature; and

b) E is a set of conditional equations with respect to <S,OP>.

A.3 Derivation systems

A **derivation system** is a 3-tuple D=<A,Ax,I> with:

a) A a set, the elements of which are called **assertions**;

b) A⊇Ax the set of **axioms**;

c) I a set of **interface rules**.

Each interference rule R∈ I has the following format

$$R: \frac{P_1,\ldots,P_n}{Q}$$

where P_1,...,P_n,Q∈ A.

A **derivation** of an assertion P in a derivation system D is a finite sequence s of assertions satisfying the following conditions:

a) the last element of s is P;

b) if Q is an element of s, then either $Q \in Ax$, or there exists a rule $R \in I$

$$R: \frac{P_1,\ldots,P_n}{Q}$$

with $P_1,...,P_n$ elements of s preceding Q.

If there exists a derivation of P in a derivation system D, this is written D |–P. If D is uniquely determined by context this may be abbreviated to |–P.

A.4 Semantics of algebraic specifications

All occurrences of a set of sorts S, a set operations OP, and a set of equations E in A.4 refer to a given algebraic specification SPEC=<S,OP,E> as defined in A.2.1.6.

In order to define the semantics of an algebraic specification SPEC, a derivation system associated with SPEC is used. This derivation system is defined in A.4.1-A.4.3. Using this derivation system a relation on the set of ground terms with respect to <S,OP,E> and congruence classes are defined in A.4.4 and A.4.5. This relation is used in A.4.6 to define an algebra (see A.1) that represents the data type that is specified by <S,OP,E>.

A.4.1 Axioms generated by equations

Let ceq be a conditional equation. The set of axioms generated by ceq, notation Ax(ceq), is defined as follows:

a) if ceq=<V,Eq,e> with $Eq \neq \{\}$, then Ax(ceq)={}; and

b) if ceq=<V,{},e> then Ax(ceq) is the set of all ground instances of e (see A.2.1.3).

A.4.2 Inference rules generated by equations

Let ceq be a conditional equation. The set of inference rules generated by ceq, notation Inf(ceq), is defined as follows:

a) if ceq=<V,{},e>, then Inf(ceq)={}; and

b) if ceq=<V,{$e_1,\ldots,e_n$ }, e> with n>0, then Inf(ceq) contains all rules of the form

$$\frac{e_1',\ldots,e_n'}{e'}$$

where $e_1',\ldots, e_n',e'$ are ground instances of $e_1,\ldots, e_n,e$ respectively, that are obtained by, for each variable x occurring in V, replacing all occurrences of that variable in $e_1,\ldots, e_n,e$ by the same ground term with sort **sort(x)**.

A.4.3 Generated derivation system

The **derivation system** D=<A,Ax,I> (see A.3) generated by an algebraic specification SPEC=<S,OP,E> is defined as follows:

a) A is the set of all ground instances of equations w.r.t. <S,OP>; and

b) $Ax=\cup\{Ax(ceq) \mid ceq \in E\} \cup ID$,
with ID={t=t | t is a ground term}; and

c) $I=\cup\{Inf(ceq) \mid ceq \in E\} \cup SI$,
where SI is given by the following schemata:

i) $\dfrac{t_1=t_2}{t_2=t_1}$ for all ground terms t_1, t_2; and

ii) $\dfrac{t_1=t_2,\ t_2=t_3}{t_1=t_3}$ for all ground terms t_1, t_2, t_3; and

iii) $\dfrac{t_1=t_1',\ldots,\ t_n=t_n'}{op(t_1,\ldots,t_n)=op(t_1',\ldots,t_n')}$

for all operators $op:s_1,\ldots,s_n \rightarrow s \in OP$ with $n>0$ and all ground terms of t_i,t_i' of sort s_i for $i=1,...,n$.

A.4.4 Congruence relation generated by an algebraic specification

Let D be the derivation system generated by an algebraic specification SPEC=<S,OP,E>. Two ground terms t_1 and t_2 are called **congruent** with respect to SPEC, notation $t_1 \equiv_{SPEC} t_2$, if

$$D \mid\!- t_1=t_2$$

A.4.5 Congruence classes

The **SPEC-congruence class** [t] of a ground term t is the set of all terms congruent to t with respect to SPEC, i.e.

$$[t] = \{t' \mid t \equiv_{SPEC} t'\}$$

A.4.6 Quotient term algebra

The semantical interpretation of an algebraic specification SPEC=<S,OP,E> is the following many-sorted algebra Q=<Dq,Oq>, called the **quotient term algebra**, where

a) Dq is the set $\{Q(s) \mid s \in S\}$ where
$Q(s)= \{[t] \mid t$ is ground term of sort $s\}$ for each $s \in S$; and

b) Oq is the set of operations $\{op' \mid op \in OP\}$, where the op' are defined by
$op'([t_1],\ldots, [t_n]) = [op(t_1,\ldots,t_n)]$.

Superseded by a more recent version

Annex D

(to Recommendation Z.100)

SDL Predefined data

This Annex defines data sorts and data generators defined in the implicit package Predefined.

NOTE – § 5.3.1.1 of the Recommendation defines the syntax and precedence of special operators (infix and monadic), but the semantics of these operators are defined by the data definitions in this section.

The remaining part of the annex is given in SDL.

```
/*      C.1         Boolean sort            */

/*      C.1.1       Definition              */

newtype   Boolean
    literals True, False;
    operators

        "not"   :     Boolean              ->  Boolean;
          /*
         "="    :     Boolean, Boolean     ->  Boolean; The "=" and "/=" operators
         "/="   :     Boolean, Boolean     ->  Boolean; are implied. See § 5.3.1.4
          */
        "and"   :     Boolean, Boolean     ->  Boolean;
        "or"    :     Boolean, Boolean     ->  Boolean;
        "xor"   :     Boolean, Boolean     ->  Boolean;
          "=>"  :     Boolean, Boolean     ->  Boolean;

    axioms
        not (True)              = = False;
        not (False)             = = True;

        True  and True          = = True;
        True  and False         = = False;
        False and True          = = False;
        False and False         = = False;

        True or True      = =   True;
        True or False     = =   True;
        False or True     = =   True;
        False or False    = =   False;

        True xor True           = = False;
        True xor False          = = True;
        False xor True          = = True;
        False xor False         = = False;

        True  => True           = = False;
        True  => False          = = False;
        False => True           = = True;
        False => False          = = True;
endnewtype Boolean;
```

```
/*    C.1.2    Usage                    */

/*

The Boolean sort is used to represent true and false values. Often it is used as the result of a comparison.

The Boolean sort is used by many of the short-hand forms of data in SDL such as axioms without the "= =" symbol, and
the implicit equality operators "=" and "/=".
*/

/*    C.2      Character sort           */

/*    C.2.1    Definition               */

newtype  Character
   literals
      NUL,  SOH,  STX,  ETX,  EOT,  ENQ,  ACK,  BEL,
      BS,   HT,   LF,   VT,   FF,   CR,   SO,   SI,
      DLE,  DC1,  DC2,  DC3,  DC4,  NAK,  SYN,  ETB,
      CAN,  EM,   SUB,  ESC,  IS4,  IS3,  IS2,  IS1,
      ' ',  '!',  '"',  '#',  '¤',  '%',  '&',  '''',
      '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',
      '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
      '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',
      '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
      'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
      'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
      'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',
      '`',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
      'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
      'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
      'x',  'y',  'z',  '{',  '|',  '}',  '¯',  DEL;
/* '''' is an apostrophe,          ' ' is a space,          '¯' is an overline or tilde */

   operators
         /*
      "="     :   Character, Character ->  Boolean; The "=" and "/=" operators
      "/="    :   Character, Character ->  Boolean; are implied. See § 5.3.1.4
         */
      "<"     :   Character, Character ->  Boolean;
      "<="    :   Character, Character ->  Boolean;
      ">"     :   Character, Character ->  Boolean;
      ">="    :   Character, Character ->  Boolean;
      Num     :   Character            ->  Integer;
      Chr     :   Integer              ->  Character;

   axioms
   Num (NUL)  ==  0;    Num (SOH)  ==  1;    Num (STX)  ==  2;    Num (ETX)  ==  3;
   Num (EOT)  ==  4;    Num (ENQ)  ==  5;    Num (ACK)  ==  6;    Num (BEL)  ==  7;
   Num (BS)   ==  8;    Num (HT)   ==  9;    Num (LF)   ==  10;   Num (VT)   ==  11;
   Num (FF)   ==  12;   Num (CR)   ==  13;   Num (SO)   ==  14;   Num (SI)   ==  15;
   Num (DLE)  ==  16;   Num (DC1)  ==  17;   Num (DC2)  ==  18;   Num (DC3)  ==  19;
   Num (DC4)  ==  20;   Num (NAK)  ==  21;   Num (SYN)  ==  22;   Num (ETB)  ==  23;
   Num (CAN)  ==  24;   Num (EM)   ==  25;   Num (SUB)  ==  26;   Num (ESC)  ==  27;
   Num (IS4)  ==  28;   Num (IS3)  ==  29;   Num (IS2)  ==  30;   Num (IS1)  ==  31;
   Num ('  ')  ==  32;   Num (' ! ')  ==  33;   Num (' " ')  ==  34;   Num (' # ')  ==  35;
   Num (' ¤ ')  ==  36;   Num (' % ')  ==  37;   Num (' & ')  ==  38;   Num (' ' ')  ==  39;
   Num (' ( ')  ==  40;   Num ((' ) ')  ==  41;   Num (' * ')  ==  42;   Num (' + ')  ==  43;
   Num (' , ')  ==  44;   Num (' - ')  ==  45;   Num (' . ')  ==  46;   Num (' / ')  ==  47;
   Num (' 0 ')  ==  48;   Num (' 1 ')  ==  49;   Num (' 2 ')  ==  50;   Num (' 3 ')  ==  51;
   Num (' 4 ')  ==  52;   Num (' 5 ')  ==  53;   Num (' 6 ')  ==  54;   Num (' 7 ')  ==  55;
   Num (' 8 ')  ==  56;   Num (' 9 ')  ==  57;   Num (' : ')  ==  58;   Num (' ; ')  ==  59;
   Num (' < ')  ==  60;   Num (' = ')  ==  61;   Num (' > ')  ==  62;   Num (' ? ')  ==  63;
   Num (' @ ')  ==  64;   Num (' A ')  ==  65;   Num (' B ')  ==  66;   Num (' C ')  ==  67;
   Num (' D ')  ==  68;   Num (' E ')  ==  69;   Num (' F ')  ==  70;   Num (' G ')  ==  71;
   Num (' H ')  ==  72;   Num (' I ')  ==  73;   Num (' J ')  ==  74;   Num (' K ')  ==  75;
```

```
    Num (' L ')    ==  76;    Num (' M ')    ==  77;    Num (' N ')    ==  78;    Num (' O ')    ==  79;
    Num (' P ')    ==  80;    Num (' Q ')    ==  81;    Num (' R ')    ==  82;    Num (' S ')    ==  83;
    Num (' T ')    ==  84;    Num (' U ')    ==  85;    Num (' V ')    ==  86;    Num (' W ')    ==  87;
    Num (' X ')    ==  88;    Num (' Y ')    ==  89;    Num (' Z ')    ==  90;    Num ('[ ')     ==  91;
    Num (' \ ')    ==  92;    Num (' ] ')    ==  93;    Num (' ^ ')    ==  94;    Num (' _ ')    ==  95;
    Num (' ‘ ')    ==  96;    Num (' a ')    ==  97;    Num (' b ')    ==  98;    Num (' c ')    ==  99;
    Num (' d ')    == 100;    Num (' e ')    == 101;    Num (' f ')    == 102;    Num (' g ')    == 103;
    Num (' h ')    == 104;    Num (' i ')    == 105;    Num (' j ')    == 106;    Num (' k ')    == 107;
    Num (' l ')    == 108;    Num (' m ')    == 109;    Num (' n ')    == 110;    Num (' o ')    == 111;
    Num (' p ')    == 112;    Num (' q ')    == 113;    Num (' r ')    == 114;    Num (' s ')    == 115;
    Num (' t ')    == 116;    Num (' u ')    == 117;    Num (' v ')    == 118;    Num (' w ')    == 119;
    Num (' x ')    == 120;    Num (' y ')    == 121;    Num (' z ')    == 122;    Num (' { ')    == 123;
    Num (' | ')    == 124;    Num (' } ')    == 125;    Num (' ¯ ')    == 126;    Num (DEL)      == 127;

    /* definition of = and /= is implied */

    for all a,b in Character (
    for all i in package Predefined Integer (
    /* definition of <, >, <=, and >= */
    a < b    ==  Num(a) <   Num(b);
    a > b    ==  Num(a) >   Num(b);
    a <= b   ==  Num(a) <=  Num(b);
    a >= b   ==  Num(a) >=  Num(b);
    /* definition of Chr */
    Chr(Num(a))    ==   a;
    Chr(i+128)     ==   Chr(i);));
endnewtype    Character;

/*      C.2.2     Usage       */

/*

The Character sort defines character strings of length 1, where the characters are those of the International Alphabet No. 5. These are defined either as strings or as abbreviations according to the International Reference Version of the alphabet. The printed representation may vary according to national usage of the alphabet.


There are 128 different literals and values defined for Character. The ordering of the values and equality and inequality are defined.
*/

/*      C.3     String generator        */

/*      C.3.1   Definition              */

generator String (type Itemsort, literal Emptystring)
    /* Strings are "indexed" from one */
    literals Emptystring;
    operators
        MkString    :   Itemsort        -> String;      /* make a string from an item */
        Length      :   String          -> package Predefined Integer;
                                                        /* length of string           */
        First       :   String          -> Itemsort;    /* first item in string       */
        Last        :   String          -> Itemsort;    /* last item in string        */
        "/ /"       :   String, String  -> String;      /* concatenation              */
        Extract!    :   String, package Predefined Integer
                                        -> Itemsort;    /* get item from string       */
        Modify!     :   String, package Predefined Integer, Itemsort
                                        -> String;      /* modify value of string     */
        Substring   :   String, package Predefined Integer, package Predefined Integer
                                        -> String;      /* get substring from string  */
            /* substring (s, i, j) gives a string of length j starting from the ith element */
```

```
axioms
        for all item, itemi, itemj, item1, item2, in Itemsort (
        for all, s, s1, s2, s3 in String (
        for all i, j, in package Predefined Integer (
        /* constructors are Emptystring, MkString, and "/ / " */
        /* equalities between constructor terms */
        s / / Emptystring     = = s;
        Emptystring / / s     = = s;
        (s1 / /  s2) / / s3       = = s1 / / (s2 / / s3);

        /* definition of Length by applying it to all constructors */
        type String Length (Emptystring)              = = 0;
        type String Length (MkString (item))         = = 1;
        type String Length (s1  / / s2)               = = Length (s1) + Length (s2);

        /*   definition of Extract! by applying it to all constructors,
             Error! cases handled separately */
        Extract! (MkString (item), 1)                       = = item;
        i <= Length (s1) = => Extract! (s1 / / s2, i)       = = Extract! (s1, i);
        i > Length (s1) = => Extract! (s1 / / s2, i)        = = Extract! (s2, i-Length (s1));
        i <=0 or i>Length(s) = => Extract! (s, i)           = = Error!;

        /* definition of First and Last by other operations */
        First (s)   = = Extract! (s, 1);
        Last (s)    = = Extract! (s, Length (s));

        /*   definition of Substring (s, i, j) by induction on j,
             Error! cases handled separately */
        i>0 and i–1 <=Length (s)                    = => Substring (s, i, 0) = = Emptystring;
        i>0 and j>0 and i+j–1 <=Length (s)     = =>
             Substring (s, i, j) = = Substring (s, i, j–1) / / MkString (Extract! (s, i+j–1));
        i<=0 or j<0 or i+j–1>Length (s)          = => Substring (s, i, j) = = Error!;

        /*   definition of Modify! by other operations */
        Modify! (s, i, item) = = Substring (s, 1, i–1) / / MkString (item) / /
             Substring (s, i+1), Length (s)-i);)));
endgenerator String;

/*        C.3.2      Usage      */

/*
A string generator can be used to define a sort which allows strings of any item sort to be constructed. The most common use will be for the Charstring defined below.

The Extract! and Modify! operators will typically be used with the shorthands defined in § 5.3.3.4 and § 5.4.3.1 for accessing the values of strings and assigning values to strings.
*/

/*        C.4        Charstring sort     */

/*        C.4.1      Definition          */

newtype Charstring String (Character, ' ')
        adding literals nameclass
             ''''  ( ('  ' : '&') or''''' or (' (' : ¯ ') ) + ' ' ' ';
             /* character strings of any length of any characters from a space ' '
                to an overline ' ¯ ' */
             /* equations of the form
                  'ABC' = = 'AB' / /  'C';
                  are implied – see § 5.3.1.2 */
        map for all c in Character literals (
             for all charstr in Charstring literals (
                  Spelling (charstr) = = Spelling (c) = =>
                       charstr = = Mkstring (c) ;));
                  /* string 'A' is formed from character 'A' etc. */
endnewtype Charstring;
```

```
/*      C.4.2     Usage     */

/*
```

The Charstring sort defines strings of characters. A Charstring literal can contain printing characters and spaces. A non printing character can be used as a string by using Mkstring, for example Mkstring(DEL).

Example:

```
synonym newline_prompt Charstring = Mkstring(CR) / / Mkstring(LF) / / '$>';

*/

/*      C.5       Integer sort      */

/*      C.5.1     Definition        */

newtype Integer
    literals nameclass (' 0 ' : ' 9 ') * (' 0 ' : ' 9 ');
        /* optional number sequence before one of the numbers 0 to 9 */

    operators
        "–"       :   Integer              -> Integer;
        "+"       :   Integer, Integer     -> Integer;
        "–"       :   Integer, Integer     -> Integer;
        "*"       :   Integer, Integer     -> Integer;
        "/"       :   Integer, Integer     -> Integer;
        "mod"     :   Integer, Integer     -> Integer;
        "rem"     :   Integer, Integer     -> Integer;

            /* The "=" and "/=" operator signatures are implied see § 5.3.1.4
        "="       :   Integer, Integer     -> Boolean;
        "/="      :   Integer, Integer     -> Boolean;
            */
        "<"       :   Integer, Integer     -> Boolean;
        ">"       :   Integer, Integer     -> Boolean;
        "<="      :   Integer, Integer     -> Boolean;
        ">="      :   Integer, Integer     -> Boolean;
        Float     :   Integer              -> Real;
                  /* axioms in newtype Real definition */
        Fix       :   Real                 -> Integer;
                  /* axioms in newtype Real definition */
        noequality;
    axioms
    for all a,b,c in Integer (
    /* constructors are 0, 1, +, and unary – */
    /* equalities between constructor terms */
    (a + b) + c             == a + (b + c);
    a + b                   == b + a;
    0 + a                   == a;
    a + (– a)               == 0;
    (– a) + (– b)           == – (a + b);
    type Integer – 0        == 0;
    – (– a)                 == a;

    /* definition of binary "–" by other operations */
    a – b                   == a + (– b);

    /* definition of "*" by applying it to all constructors */
    0 * a                   == 0;
    1 * a                   == a;
    (– a) * b               == – (a * b);
    (a + b) * c             == a * c + b * c;
```

```
    /* definition of "<" by applying it to all constructors */
    a < b                   == 0 < (b – a);
    type Integer 0 < 0      == False;
    type Integer 0 < 1      == True;
    0 < a                   == True == > 0 < (– a) ==  False;
    0 < a and 0 < b         == True == > 0 < (a + b) ==  True;

    /* definition of ">", "=", "/=", "<=", and ">=" by other operations */
    a > b                   == b < a;
    a = b                   ==  not (a < b or a > b);
    a /= b                  ==  not (a = b);
    a <= b                  ==  a < b or a = b;
    a >= b                  ==  a > b or a = b;

    */ definition of "/" by other operations */
    a / 0 ==  Error!;
    a >= 0 and b > a  ==  True  ==> a / b  ==  0;
    a >= 0 and b <= a and b > 0  ==  True  ==> a / b  ==  1 + (a-b) / b;
    a >= 0 and b < 0  ==  True  ==>  a / b  ==  (a / (– b));
    a < 0 and b < 0   ==  True  ==> a / b  ==  (– a) / (– b);
    a < 0 and b > 0   ==  True  ==> a / b  ==  – ((– a) / b);

    /* definition of "rem" by other operations */
    a rem b             ==  a – b * (a/b);

    /* definition of "mod" by other operations */
    a >= 0 and b > 0                                ==>  a mod b ==  a rem b;
    b < 0 ==> a mod b                               ==  a mod (–b);
    a < 0 and b > 0 and a rem b = 0                 ==> a mod b ==  0;
    a < 0 and b > 0 and a rem b < 0                 ==> a mod b ==  b + a rem b;
    a mod 0                                         ==   error!;

    /* definition of literals */
    type Integer 2     == 1 + 1;          type Integer 3   ==  2 + 1;
    type Integer 4     == 3 + 1;          type Integer 5   ==  4 + 1;
    type Integer 6     == 5 + 1;          type Integer 7   ==  6 + 1;
    type Integer 8     == 7 + 1;          type Integer 9   ==  8 + 1; );

    map /* literals other than 0 to 9 */
    for all a,b,c in Integer literals (
    Spelling(a) == Spelling(b) / / Spelling(c), Length (Spelling(c)) == 1
                ==> a  == b * (9 + 1) + c;);
endnewtype Integer;

/*      C.5.2     Usage       */

/*
```

The Integer sort is used for mathematical integers with decimal notation.

```
*/

/*      C.6       Natural syntype          */

/*      C.6.1     Definition               */

syntype Natural = Integer constants >= 0 endsyntype Natural;

/*      C.6.2     Usage                    */

/*
```

The natural syntype is used when positive integers only are required. All operators will be the integer operators but when a value is used as a parameter or assigned the value is checked. A negative value will be an error.

```
*/
```

```
/*      C.7     Real sort                       */

/*      C.7.1   Definition                      */

newtype Real
    literals nameclass ( (' 0 ' : ' 9 ') * (' 0 ' : ' 9 ') ) or ( (' 0 ' : ' 9 ') * ' . ' ('0 ' : ' 9 ') + );
    operators
        "–"     :   Real            -> Real;
        "+"     :   Real, Real      -> Real;
        "–"     :   Real, Real      -> Real;
        "*"     :   Real, Real      -> Real;
        "/"     :   Real, Real      -> Real;
                /*  The "=" and " /= " operator signature are implied
                    – see § 5.3.1.4
        "="     :   Real, Real      -> Boolean;
        "/="    :   Real, Real      -> Boolean;
                */
        "<"     :   Real, Real      -> Boolean;
        ">"     :   Real, Real      -> Boolean;
        "<="    :   Real, Real      -> Boolean;
        ">="    :   Real, Real      -> Boolean;
        noequality;
    axioms
    for all r,s in Real (
    for all a,b,c,d in Integer (
    /* constructors are Float and "/" */
    /* equalities between constructor terms allow to reach always a form
        Float (a) / Float (b) where b > 0 */
    r  / Float (0)                          == Error!;
    r  / Float (1)                          == r;
    c /= 0  ==> Float (a)  /  Float (b)     == Float (a*c)  /  Float (b*c);
    b /= 0 and d  /=  0  ==>
         (Float (a)  /  Float (b) )  /  Float (c)   /  Float (d) ) ==
                        Float (a*d)  /  Float (b*c);

    /* definition of unary "–" by applying it to all constructors */
    – (Float (a) / Float (b))     ==  Float (– a) / Float (b);

    /* definition of "+" by applying it to all constructors */
    (Float (a) / Float (b)) + (Float (c) / Float (d)) ==
                          Float (a*d + c*b) / Float (b*d);

    /* definition of binary "–" by other operations */
    r – s ==  r + (– s);

    /* definition of "*" by applying it to all constructors */
    (Float (a) / Float (b)) * (Float (c) / Float (d)) == Float (a*c) / Float (b*d);

    /* definition of "<" by applying it to all constructors */
    b > 0 and d > 0 ==> (Float (a) / Float (b)) < (Float (c) / Float (d)) == a * d < c * b;

    /* definition of ">", "=", "/=", "<=", and ">=" by other operations */
    r > s ==  s < r;
    r = s ==  not (r < s or r > s);
    r /= s ==  not (r = s);
    r <= s ==  r < s or r = s;
    r >= s ==  r > s or r = s;

    /* definition of Fix by applying it to all constructors */
    a >= b and b > 0 ==>
          Fix (Float (a) / Float (b)) == Fix (Float (a-b) / Float (b)) + 1;
    b > a and a >= 0 ==>
          Fix (Float (a) / Float (b)) == 0;
```

```
    a < 0 and b > 0 ==>
        Fix (Float (a) / Float (b)) == – Fix (Float (– a) / Float (b)) – 1;));

map
    for all r, s in Real literals (
    for all i, j in Integer literals (
        Spelling (r)                    == Spelling (i)      ==> r     == Float (i);
        Spelling (r)                    == Spelling (i)      ==> i     == Fix (r);
        Spelling (r)                    == Spelling (i) // Spelling (s),
          Spelling (s)                  == '.' // Spelling (j) ==> r == Float (i) + s;
        Spelling (r)                    == '.' // Spelling (i), Length (Spelling (i)) == 1
          ==> r == Float (i) / 10;
        Spelling (r)            == '.' // Spelling (i) // Spelling (j),
        Length (Spelling (i))   == 1, Spelling (s) == '.' // Spelling (j)
          ==> r == (Float (i)  +  s) / 10; ));
endnewtype Real;

/*      C.7.2   Usage           */

/*
```

The real sort is used to represent real numbers. The real sort can represent all numbers which can be represented as one integer divided by another. Numbers which cannot be represented in this way (irrational numbers – for example $\sqrt{2}$) are not part of the real sort. However, for practical engineering a sufficiently accurate approximation can usually be used. Defining a set of numbers which includes all irrationals is not possible without using additional techniques.

```
*/

/*      C.8     Array generator         */

/*      C.8.1   Definition              */

generator Array (type Index, type Itemsort)
    operators
        Make!     : Itemsort                -> Array;
        Modify!   : Array, Index, Itemsort  -> Array;
        Extract!  : Array, Index            -> Itemsort;
    axioms
    for all item, item1, item2, itemi, itemj in Itemsort (
    for all i, j, ipos in Index (
    for all a, s in Array (
        type Array Extract! (Make! (item), i)       == item;
        i = j ==>
            Modify! (Modify! (s, i, item1), j, item2)     == Modify! (s, i, item2);
        i = j ==>
            Extract! (Modify! (a, i, item), j)            == item;
        i = j == False ==>
            Extract! (Modify! (a, i, item), j)            == Extract! (a, j);
        i = j == False ==>
            Modify! (Modify! (s, i, itemi), j, itemj)     == Modify! (Modify! (s, j, itemj), i, itemi);
        /* equality */
        type Array Make! (item1) = Make! (item2)      == item1 = item2;
        a=s == True, i = j == True, itemi = itemj ==>
            Modify! (a, i, itemi) = Modify! (s, j, itemj) == True;
        Extract! (a, i) = Extract! (s, i) == False ==> a = s == False;)));
endgenerator Array;

/*      C.8.2   Usage           */

/*
```

The array generator can be used to define one sort which is indexed by another. For example:

```
        newtype indexbychar Array (Character, Integer)
        endnewtype indexbychar;
```

defines an array containing integers and indexed by characters.

Arrays are usually used in combination with the shorthand forms of Modify! and Extract! defined in § 5.4.3.1 and § 5.3.3.4 for indexing, and Make! defined in § 5.3.3.6. For example:

```
        dcl charvalue indexbychar;
        …….
        task charvalue ('A') := charvalue ('B') –1;
*/

/*      C.9     Powerset generator      */

/*      C.9.1   Definition              */

generator Powerset (type itemsort)
    literals Empty;
    operators
        "in"    : Itemsort, Powerset    -> package Predefined Boolean;
                                                /* is member of             */
        Incl    : Itemsort, Powerset    -> Powerset;    /* include item in set  */
        Del     : Itemsort, Powerset    -> Powerset;    /* delete item from set */
        "<"     : Powerset, Powerset    -> package Predefined Boolean;
                                                /* is proper subset of      */
        ">"     : Powerset, Powerset    -> package Predefined Boolean;
                                                /* is proper superset of    */
        "<="    : Powerset, Powerset    -> package Predefined Boolean;
                                                /* is subset of             */
        ">="    : Powerset, Powerset     -> package Predefined Boolean;
                                                /* is superset of           */
        "and"   : Powerset, Powerset    -> Powerset;    /* intersection of sets */
        "or"    : Powerset, Powerset    -> Powerset;    /* union of sets        */
axioms
    for all i, j in Itemsort (
    for all p,ps,a,b,c in Powerset (
    /* constructors are Empty and Incl */
    /* equalities between constructor terms */
    Incl (i, Incl (j,p))                == Incl (j, Incl (i, p));
    i = j ==> Incl (i, Incl (j, p))     == Incl (i, p);

    /* definition of "in" by applying it to all constructors */
    i in type Powerset Empty            == False;
    i in Incl (j, ps)                   == i=j or i in ps;

    /* definition of Del by applying it to all constructors */
    type Powerset Del (i, Empty)    ==  Empty;
    i = j ==> Del (i, Incl (j, ps))     ==  Del (i, ps);
    i /= j ==> Del (i, Incl (j, ps))    ==  Incl (j, Del (i, ps));

    /* definition of "<" by applying it to all constructors */
    a < type Powerset Empty             ==  False;
    type Powerset Empty < Incl (i, b)   ==  True;
    Incl (i, a) < b                     ==  i  in b and Del (i, a) < Del (i, b);

    /* definition of ">" by other operations */
    a > b ==  b < a;

    /* definition of "=" by applying it to all constructors */
    Empty = Incl (i, ps)    == False;
    Incl (i, a) = b         ==  i in b and Del (i, a) = Del (i, b);

    /* definition of "<=" and ">=" by other operations */
    a <= b                  ==  a < b or a = b;
    a >= b                  ==  a > b or a = b;
```

```
    /* definition of "and" by applying it to all constructors */
    Empty and b                         == Empty
    i in b        ==> Incl (i, a) and b    ==  Incl (i, a and b);
    not (i in b) ==> Incl (i, a) and b    ==  a and b;

    /* definition of "or" by applying it to all constructors */
    Empty or b      == b;
    Incl (i, a) or b   ==  Incl (i, a or b);));
endgenerator Powerset;

/*      C.9.2   Usage          */
/*
```

Powersets are used to represent mathematical sets. For example:

```
        newtype Boolset Powerset (Boolean) endnewtype Boolset;
```

can be used for a variable which can be empty or contain (True), (False) or (True, False).

```
*/

/*      C.10    PId sort        */

/*      C.10.1  Definition      */

newtype PId
    literals Null;
    operators
        unique! : PId -> PId;
            /* The "=" and "/=" operator signatures are implied – see § 5.3.1.4
        "=" : PId, PId              -> Boolean;
        "/=" : PId, PId             -> Boolean;
            */
    axioms
        for all p, p1, p2 in PId (
        unique! (p)           = Null              == False;
        unique! (p1)          = unique! (p2)      ==  p1 = p2 );
default Null;
endnewtype PId;

/*      C.10.2      Usage               */

/*
```

The PId sort is used for process identities. Note that there are no other literals than the value Null. When a process is created the underlying system uses the unique! operator to generate a new unique value.

```
*/

/*      C.11        Duration sort       */

/*      C.11.1      Definition          */

newtype Duration
    literals nameclass (' 0 ' : ' 9 ') + or ((' 0 ' : ' 9 ') * ' . ' (' 0 ' : ' 9 ') +);
    operators
    duration!       : Real                  -> Duration;
    "+"             : Duration, Duration    -> Duration;
    "–"             : Duration              -> Duration;
    "–"             : Duration, Duration    -> Duration;
    ">"             : Duration, Duration    -> Boolean;
    "<"             : Duration, Duration     -> Boolean;
    ">="            : Duration, Duration    -> Boolean;
```

```
    "<="            : Duration, Duration      -> Boolean;
    "*"             : Duration, Real          -> Duration;
    "*"             : Real, Duration          -> Duration;
    "/"             : Duration, Real          -> Duration;
    noequality
axioms
    /* constructor is duration */
    for all a, b in Real (
    for all d, e in Duration (
    /* definition of "+" by applying it to all constructors */
    duration! (a) + duration! (b) ==  duration! (a + b);

    /* definition of unary "–" by applying it to all constructors */
    – duration! (a) ==  duration! (–a);

    /* definition of binary "–" by other operations */
    d – e ==  d + (–e);

    /* definition of "=", "/=", ">", "<", ">=", and "<=" by applying it to all constructors */
    duration! (a) = duration! (b)       ==  a = b;
    duration! (a) /= duration! (b)      ==  a /= b;
    duration! (a) > duration! (b)       ==  a > b;
    duration! (a) < duration! (b)       ==  a < b;
    duration! (a) >= duration! (b)      ==  a >= b;
    duration! (a) <= duration! (b)      ==  a <= b;

    /* definition of "*" by applying it to all constructors */
    duration! (a) * b                   ==  duration! (a * b);
    a * d                               ==  d * a;

    /* definition of "/" by applying it to all constructors */
    duration! (a) / b                   ==  duration! (a / b);));

map
    for all d in Duration literals (
    for all r in Real literals (
    Spelling (d) == Spelling (r) ==> d = duration! (r);));
endnewtype Duration;

/*      C.11.2      Usage               */

/*
```

The duration sort is used for the value to be added to the current time to set timers. The literals of the sort duration are the same as the literals for the real sort. The meaning of one unit of duration will depend on the system being defined.

Durations can be multipled and divided by reals.

```
*/

/*      C.12        Time sort                   */

/*      C.12.1      Definition                  */

newtype Time
    literals nameclass (' 0 ' : ' 9 ') + or ((' 0 ' : ' 9 ') * ' . ' (' 0 ' : ' 9 ') +);
    operators
    time!   : Duration          -> Time;
    "<"     : Time, Time        -> Boolean;
    "<="    : Time, Time        -> Boolean;
    ">"     : Time, Time        -> Boolean;
    ">="    : Time, Time        -> Boolean;
```

```
    "+"     : Time, Duration        -> Time;
    "+"     : Duration, Time        -> Time;
    "–"     : Time, Duration        -> Time;
    "–"     : Time, Time            -> Duration;
    noequality
    axioms
    /* constructor is time! */
    for all t, u in Time (
    for all a, b in Duration (
    /* definition of ">", "=" by applying it to all constructors */
    time! (a) > time! (b)       ==  a > b;
    time! (a) = time! (b)       ==  a = b;

    /* definition of "/=", "<", "<=", ">=" by other operations */
    t /= u                      ==  not (t = u);
    t < u                       ==  u > t;
    t <= u                      ==  (t < u) or (t = u);
    t >= u                      ==  (t > u) or (t = u);

    /* definition of "+" by applying it to all constructors */
    time! (a) + b               ==  time! (a + b);
    a + t                       ==  t + a;

    /* definition of "–" : Time, Duration by other operations */
    t – b                       ==  t + (–b);

    /* definition of "–
" : Time, Time by applying it to all constructors */
    time! (a) – time! (b)       ==  a – b;));

    map
    for all d in Duration literals (
    for all t in Time literals (
    Spelling (d) ==  Spelling (t) == > t ==  time! (d); ));

endnewtype Time;

/*      C.12.2      Usage               */

/*
```

The **now** expression returns a value of the time sort. A time value may have a duration added or subtracted from it to give another time. A time value subtracted from another time value gives a duration. Time values are used to set the expiry time of timers.

The origin of time is system dependent. A unit of time is the amount of time represented by adding one duration unit to a time.

*/