
**Information technology — Portable
Common Tool Environment (PCTE) —**

Part 3:
Ada programming language binding

*Technologies de l'information — Environnement d'outil courant
portable (PCTE) —*

Partie 3: Liant de langage de programmation Ada



Contents

1 Scope	1
2 Conformance	1
3 Normative references	1
4 Definitions	2
5 Formal notations	2
6 Outline of the Standard	2
7 Binding strategy	2
7.1 Ada programming language standard	2
7.2 General principles	2
7.3 Dynamic memory management	3
7.4 Complex entities as parameters	4
7.5 Character strings	4
7.6 Error conditions	4
7.7 Implementation limits	4
8 Datatype mapping	5
8.1 Mapping of PCTE datatypes to LI datatypes	5
8.1.1 Mapping of predefined PCTE datatypes	5
8.1.2 Mapping of private PCTE datatypes	6
8.1.3 Mapping of complex PCTE datatypes	7
8.1.4 New LI datatype generators	7
8.2 Mapping of LI datatypes to Ada datatypes	8
8.2.1 LI datatype: boolean	8
8.2.2 LI datatype: pcte-integer	8
8.2.3 LI datatype: pcte-natural	9
8.2.4 LI datatype: pcte-float	9
8.2.5 LI datatype: pcte-time	10
8.2.6 LI datatype: octet	10
8.2.7 LI datatype: pcte-text	11
8.2.8 LI datatype generator: pcte-sequence	11
8.2.9 LI datatype generator: bounded-set	14

© ISO/IEC 1998

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from the publisher.

ISO/IEC Copyright Office • Case postale 56 • CH-1211 Genève 20 • Switzerland

Printed in Switzerland

8.2.10	LI datatype: record	15
8.2.11	LI datatype: private	16
8.2.12	LI enumerated datatype: pcte-xxx	16
8.3	Deriving Ada subprogram semantics from the abstract specification	16
8.4	Package Pcte	17
9	Object managment	36
9.1	Object management datatypes	36
9.2	Link operations	36
9.3	Object operations	40
9.4	Version operations	49
10	Schema management	51
10.1	Schema management datatypes	52
10.2	Update operations	53
10.3	Usage operations	59
10.4	Working schema operations	61
11	Volumes, devices, and archives	63
11.1	Volume, device, and archive datatypes	63
11.2	Volume, device, and archive operations	64
12	Files, pipes, and devices	69
12.1	File, pipe, and device datatypes	70
12.2	File, pipe, and device operations	70
13	Process execution	73
13.1	Process execution datatypes	73
13.2	Process execution	76
13.3	Security operations	79
13.4	Profiling operations	81
13.5	Monitoring operations	81
14	Message queues	82
14.1	Message queue datatypes	83
14.2	Message queue operations	85
15	Notification	88
15.1	Notification datatypes	88
15.2	Notification operations	88
16	Concurrency and integrity control	89
16.1	Concurrency and integrity control datatypes	89
16.2	Concurrency and integrity control operations	89

17 Replication	91
17.1 Replication datatypes	91
17.2 Replication operations	91
18 Network connection	92
18.1 Network connection datatypes	93
18.2 Network connection operations	93
18.3 Foreign system operations	95
18.4 Time operations	95
19 Discretionary security	95
19.1 Discretionary security datatypes	95
19.2 Discretionary access control operations	98
19.3 Discretionary security administration operations	99
20 Mandatory security	101
20.1 Mandatory security datatypes	101
20.2 Mandatory security operations	101
20.3 Mandatory security administration operations	102
20.4 Mandatory security operations for processes	104
21 Auditing	105
21.1 Auditing datatypes	105
21.2 Auditing operations	114
22 Accounting	119
22.1 Accounting datatypes	119
22.2 Accounting operations	122
22.3 Consumer identity operations	124
23 References	124
24 Limits	124
25 Errors	126
Annex A - The object orientation module	137
Index of abstract operations	147
Index of Ada subprograms	153
Index of Ada datatypes	159

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

International Standard ISO/IEC 13719-3 was prepared by ECMA (as Standard ECMA-162) and was adopted, under a special “fast-track procedure”, by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, in parallel with its approval by national bodies of ISO and IEC.

This second edition cancels and replaces the first edition (ISO/IEC 13719-3:1995), which has been technically revised.

ISO/IEC 13719 consists of the following parts, under the general title *Information technology - Portable Common Tool Environment (PCTE)*:

- *Part 1: Abstract specification*
- *Part 2: C programming language binding*
- *Part 3: Ada programming language binding*
- *Part 4: IDL binding (Interface Definition Language)*

Annex A forms an integral part of this part of ISO/IEC 13719.

Information technology — Portable Common Tool Environment (PCTE) —

Part 3:

Ada programming language binding

1 Scope

This part of ISO/IEC 13719 defines the binding of the Portable Common Tool Environment (PCTE) interfaces, as specified in ISO/IEC 13719-1, to the Ada programming language.

A number of features are not completely defined in ISO/IEC 13719-1, some freedom being allowed to the implementor. Some of these features are specified as implementation limits. Some constraints are placed on these implementation limits by this part of ISO/IEC 13719. These constraints are specified in clause 24.

PCTE is an interface to a set of facilities that forms the basis for constructing environments supporting systems engineering projects. These facilities are designed particularly to provide an infrastructure for programs which may be part of such environments. Such programs, which are used as aids to system development, are often referred to as tools.

2 Conformance

An implementation of PCTE conforms to this part of ISO/IEC 13719 if it conforms to 2.2 of ISO/IEC 13719-1, where the binding referred to there is taken to be the Ada language binding defined in clauses 1 to 5 and 8 to 25 of this part of ISO/IEC 13719. All other parts of this part of ISO/IEC 13719 are provided as assistance to the reader and are not normative.

The Ada language binding defined in this part of ISO/IEC 13719 conforms to 2.1 of ISO/IEC 13719-1.

3 Normative references

The following standards contain provisions which, through reference in this text, constitute provisions of this part of ISO/IEC 13719. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of ISO/IEC 13719 are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below. Members of IEC and ISO maintain registers of currently valid International Standards.

ISO/IEC 13719-1:1998,	<i>Information technology - Portable Common Tool Environment (PCTE) - Part 1: Abstract specification.</i>
ISO/IEC 13719-2:1998,	<i>Information technology - Portable Common Tool Environment (PCTE) - Part 2: C programming language binding.</i>
ISO 8601:1988,	<i>Data elements and interchange formats - Information interchange - Representation of dates and times.</i>
ISO/IEC 8652:1995,	<i>Information technology - Programming languages - Ada.</i>

ISO/IEC TR 10182:1993, *Information technology - Programming languages, their environments and system software interfaces - Guidelines for language bindings.*

ISO/IEC 11404:1996, *Information technology - Programming languages, their environments and system software interfaces - Language-independent datatypes.*

4 Definitions

All technical terms used in this part of ISO/IEC 13719, other than a few in widespread use, are defined in the body of this part of ISO/IEC 13719 or in the referenced documents.

5 Formal notations

All datatypes and subprogram definitions are expressed using ISO/IEC 8652 conformant syntax.

6 Outline of the Standard

Clause 7 describes the strategy used to develop this binding specification.

Clause 8 defines the mapping from the datatypes that are used in the abstract specification to Ada programming language datatypes.

Clauses 9 to 22 define the bindings of datatypes and operations in the corresponding clauses of ISO/IEC 13719-1. The extensions for fine-grain objects are added at the end of clause 11.

Clause 23 defines the binding of object and type references, as specified in ISO/IEC 13719-1 23.1.2 and 23.2. Because of the package structure, this clause consists of a cross-reference to the definitions which are in 8.4.

Clause 24 defines the binding of the implementation limit subprograms described in ISO/IEC 13719-1, clause 24.

Clause 25 defines the binding of the error conditions specified in ISO/IEC 13719-1, clause 25, and defines binding-defined error conditions for the Ada binding.

Annex A, which is normative, contains the extensions for object orientation, corresponding to annex G of ISO/IEC 13719-1.

7 Binding strategy

7.1 Ada programming language standard

The Ada package specifications were designed to conform to ISO/IEC 8652.

7.2 General principles

The following general principles were applied when generating the binding in this part of ISO/IEC 13719.

ISO/IEC TR 10182 should be followed as far as possible for binding method 1: provide a completely defined procedural interface.

Each operation in ISO/IEC 13719-1 should be represented by one subprogram in this part of ISO/IEC 13719 unless there are specific reasons to the contrary.

All Ada identifiers should be in lower case except for predefined identifiers, named constant values, and enumeration literals. Since the Ada standard is insensitive to case this is for typographical consistency between ISO/IEC 13719-1, ISO/IEC 13719-2, and this part of ISO/IEC 13719.

Nondefining occurrences of the names of Ada subprograms and types should use the fully qualified form, so as to identify all package dependences.

All the Ada packages should have names that begin with 'Pcte_' to ensure they are unique within an Ada Library System. The choice of case of the characters of 'Pcte' is for typographical consistency with ISO/IEC 13719-2.

An abstract operation with name of the form 'TYPE_VERB_PHRASE' should be mapped to an Ada subprogram 'verb_phrase' declared by a package called 'Pcte_type'. For example, 'PROCESS_SET_WORKING_SCHEMA' is mapped to 'Pcte_process.set_working_schema'

When a package hierarchy is required, it should be compatible with the abstract specification clause organisation. For example, 'ACCOUNTING_LOG_READ' is mapped to 'Pcte_accounting.log.read'.

Names should be retained from ISO/IEC 13719-1 as far as possible.

All additional names should be chosen appropriately for their meanings.

Each operation that can return errors should have an additional **in** parameter of an access type designating an object into which error indications can be returned. This allows the subprograms to be procedures or functions as appropriate.

Wherever practical, types introduced for passing complex data entities between a caller and a subprogram should be private or limited private. Limited private types should be used unless the basic operations on entities of such types are safe and consistent with ISO/IEC 13719-1.

All simple parameter types in ISO/IEC 13719-1 that represent attribute value types should be mapped to corresponding Ada types defined by this binding.

All simple parameter types in ISO/IEC 13719-1 that do not represent attribute value types should be mapped to predefined types or subtypes or derived types of predefined types.

7.3 Dynamic memory management

A type defined in this part of ISO/IEC 13719 for which an object is created dynamically is always limited private, and subprograms are provided to construct, access and discard such objects.

It is the responsibility of the Ada program that declares a variable of such an Ada type to ensure that its 'construct' subprogram is called before the variable is read; the construct subprogram always initializes the value of the variable. Use of a variable of this type before calling its construct subprogram, or after calling its discard subprogram always causes an error to be generated.

It is the responsibility of the Ada program that defines variables of such Ada types to ensure that the 'discard' subprogram is called when the variable is no longer needed and before the immediate scope of the variable is left. If such a variable goes out of scope without being discarded there is the possibility that the memory allocated to it cannot be recovered, which may result in the exception `STORAGE_ERROR` being raised subsequently.

If a constructed variable is constructed again without having been discarded, no error is reported; such usages of the interface are regarded as akin to allowing such variables to go out of scope before they are discarded.

7.4 Complex entities as parameters

Some complex entities to be passed into or retrieved from an operation are defined as sets or sequences of a base type in ISO/IEC 13719-1. Where such sets and sequences are bounded and constrainable by the user they are mapped to Ada language array types.

Where such sets and sequences are unbounded they are mapped to limited private types declared by nested packages. These limited private types are defined with operations for construction with initial value, access and discarding. A value is assigned to an object of one of these types either as a parameter of mode **out** or **in out** of a subprogram corresponding to an abstract operation of ISO/IEC 13719-1, or by use of a subprogram of the appropriate nested package.

Thus the data values contained in sequences can be easily manipulated using Ada language facilities, while allowing the implementation to choose the best implementation.

All the operations that have a sequence as an **in out** parameter create the elements of the sequence and return it as a result of the operation. The user does not need to create the elements of the sequence in advance: the user needs only to declare a sequence variable and pass that variable.

7.5 Character strings

Values of datatype `String` are manipulated as the Ada type `Pcte.string` which is a subtype of the predefined type `STANDARD.STRING`.

7.6 Error conditions

This part of ISO/IEC 13719 allows any error to be recorded as an error value or to be raised as an exception. This is achieved by providing a limited private type, `Pcte_error.handle`, into objects of which type error values, established by a subprogram defined in this part of ISO/IEC 13719, can be recorded. These handles may also be set to cause any error to be raised as an exception instead of recording it as an error value.

7.7 Implementation limits

ISO/IEC 13719-1 defines a set of limits that must be honoured by all implementations of a Language Binding. Clause 24 defines the Ada names for these limits and the operations by which they can be retrieved.

8 Datatype mapping

This clause defines the mapping of the parameter and result datatypes of the operations of ISO/IEC 13719-1 (*PCTE datatypes*) to the parameter and result datatypes of the operations of this part of ISO/IEC 13719 (*Ada datatypes*).

PCTE datatype names are printed in normal characters, the names of parameters to operations of ISO/IEC 13719-1 are printed in italics and the names of the operations themselves are printed in all upper case characters. PCTE datatype constructors are printed in bold. LI datatype names are printed in italics. Ada datatype names are printed in normal characters, as are the names of Ada subprograms and their parameter names. Enumeration literals, exception names and the names of constant objects are printed in all upper case characters. Ada reserved words are printed in bold.

The mapping from PCTE datatypes to Ada datatypes is done in two stages, via *LI datatypes* defined in ISO/IEC DIS 11404.

8.1 Mapping of PCTE datatypes to LI datatypes

As far as possible the names of PCTE datatypes are retained, with minor typographical changes, for the corresponding LI datatypes, but some new names are introduced.

The general strategy of this mapping is as follows.

- To select for each PCTE datatype a LI datatype definition which matches the requirements of the PCTE datatype defined in ISO/IEC 13719-1. The LI datatype definition is, where possible, a primitive LI datatype, and otherwise a generated LI datatype.
- To define new datatype generators where needed.
- To map PCTE datatypes with the same properties to the same LI datatype.

8.1.1 Mapping of predefined PCTE datatypes

The mapping of these PCTE datatypes is as defined in ISO/IEC 13719-1, clause 23 and is summarized in table 1.

Table 1 - Mapping of attribute value types

PCTE datatype	LI datatype
Boolean	<i>boolean</i>
Integer	<i>pcte-integer = integer range (lb1 .. ub1)</i>
Natural	<i>pcte-natural = integer range (0 .. ub1)</i>
Float	<i>pcte-float = real (10, f1) range (lb2 .. ub2)</i>
Time	<i>pcte-time = time (second, 10, f2) range (lb3 .. ub3)</i>
Octet	<i>octet</i>
Text	<i>pcte-text = characterstring (repertoire)</i>
Enumerated type xxx=VALUE1 VALUE2 ...	<i>pcte-xx = enumerated (value1, value2, ...)</i>

8.1.2 Mapping of private PCTE datatypes

Table 2 - Mapping of other primitive PCTE datatypes

PCTE datatype	LI datatype
Address	<i>see below</i>
Contents_handle	<i>contents-handle = private</i>
Handler	<i>not used</i>
Message_handle	<i>not used</i>
Message_type	<i>see below</i>
Object_reference	<i>object-reference = private</i>
Link_reference	<i>link-reference = private</i>
Type_reference	<i>type-reference = private</i>
Position_handle	<i>position-handle = private</i>
Profile_handle	<i>profile-handle = private</i>

The PCTE datatype Address is mapped directly to a subtype of the predefined Ada type SYSTEM.ADDRESS.

The PCTE datatype Message_type is mapped directly to an Ada record type Pcte_message.message_type.

8.1.3 Mapping of complex PCTE datatypes

PCTE sequence datatypes are mapped via the new datatype generator *pcte-sequence* (see 8.1.4).

PCTE set datatypes are divided into bounded set types and unbounded set types. Bounded set types have values which are sets of enumeration values with at most 32 possible elements; all others are unbounded set types. Bounded set types are mapped via the new LI datatype generator *Bounded-set*. Unbounded set types are mapped via the new LI datatype generator *pcte-sequence*; the order of elements in the sequence is ignored by the operation. For *Message_types* see 14.1.

When used as input parameter of an operation in clauses 9 to 22, a sequence which represents a PCTE unbounded set may contain repeated elements. The effect for the operation is as though each element occurred only once.

When returned as the result of an operation in clauses 9 to 22, an unbounded set has no repeated elements. The order of the elements in the sequence is arbitrary except for the procedure *normalize*, see 8.4.

PCTE map datatypes are notionally mapped via a new LI datatype generator *Map*; their mappings to Ada datatypes are defined directly (see 19.1).

PCTE union datatypes other than enumerations are notionally mapped via the datatype generator *Choice*. The only such PCTE datatypes are auditing-record, accounting-record and value-type, their mappings to Ada datatypes are defined directly (see 21.1, 22.1, 9.1 respectively).

PCTE composite and product datatypes (except composite datatypes with a single Token field, for such see 8.1.2) are mapped to the datatype generator *Record*. The component types of the composite or product type are mapped as defined by the rules of 8.1. The PCTE datatype *Message* is treated specially, see 14.1.

8.1.4 New LI datatype generators

Pcte-sequence

Description: *Pcte-sequence* is a datatype generator derived from *Sequence* by adding further characterizing operations. The values of a *pcte-sequence*, called its elements, are indexed sequentially from 1.

pcte-sequence of (*base*) = new sequence of (*base*)

The characterizing operations are: *Equal*, *IsEmpty*, *Head*, *Tail*, *Empty* and *Append* from *Sequence*, plus *Get*, *Put*, *Delete*, *InsertSequence*, *AppendSequence*, *LengthOf*, *IndexOf*, and *Normalize*.

Get (*s* : sequence of *base*, *i* : ordinal) : *base*
is the element of *s* with index *i*.

Put *s* : sequence of *base*, *e* : *base*, *i* : ordinal) : *base*
is the sequence formed from *s* by inserting *e* as an element of *s* immediately before the element with index *i*.

Delete (*s* : sequence of *base*, *i* : ordinal) : sequence of *base*
is the sequence formed from *s* by deleting the element with index *i*.

InsertSequence (s1, s2 : sequence of *base*, i : ordinal) : sequence of *base*
is the sequence formed from s1 by inserting the sequence s2 immediately before the element with index i.

AppendSequence (s1, s2 : sequence of *base*) : sequence of *base*
is the sequence formed from s1 by appending the sequence s2 at the end.

LengthOf (s : sequence of *base*) : natural
is the number of elements of s.

IndexOf (s : sequence of *base*, e : *base*) : natural
is the index of the first occurrence of the element e in the sequence s if the element is a member of the sequence, and 0 otherwise.

Normalize (s : sequence of *base*) : sequence of *base*
is the sequence formed from s by ordering the elements in an implementation-defined canonical order and deleting duplicate elements.

Bounded-set

Description: Bounded-set is a datatype generator derived from *Set* by restricting the cardinality of the values to 32 or less.

bounded-set of *base* = new set of (*base*) : size (0 .. 32)

The characterizing operations are: IsIn, Subset, Equal, Difference, Union, Intersection, Empty, SetOf, Select from *Set*.

8.2 Mapping of LI datatypes to Ada datatypes

8.2.1 LI datatype: boolean

The LI datatype *boolean* is mapped to the boolean Ada datatype Pcte.boolean.

subtype boolean **is** STANDARD.BOOLEAN

Characterizing operations

Operation	Ada Operation
Equal (x,y)	x = y
Not (x)	not x
And (x,y)	x and y
Or (x,y)	x or y

8.2.2 LI datatype: pcte-integer

The LI datatype *pcte-integer* is mapped to the integer Ada datatype Pcte.integer.

type integer **is**
range MIN_INTEGER_ATTRIBUTE .. MAX_INTEGER_ATTRIBUTE;

Characterizing operations

Operation	Ada Operation
Equal (x, y)	$x = y$
Add (x, y)	$x + y$
Multiply (x, y)	$x * y$
Negate (x)	$-x$
NonNegative (x)	$x \geq 0$
InOrder (x, y)	$x \leq y$

8.2.3 LI datatype: pcte-natural

The LI datatype *pcte-natural* is mapped to the integer Ada datatype Pcte.natural.

type natural **is range** 0 .. Pcte.integer'LAST;

Characterizing operations

Operation	Ada Operation
Equal (x, y)	$x = y$
Add (x, y)	$x + y$
Multiply (x, y)	$x * y$
InOrder (x, y)	$x \leq y$

8.2.4 LI datatype: pcte-float

The LI datatype *pcte-float* is mapped to the Ada datatype Pcte.float.

type float **is digits** MAX_DIGITS_FLOAT_ATTRIBUTE
range MIN_FLOAT_ATTRIBUTE .. MAX_FLOAT_ATTRIBUTE;

Characterizing operations

Operation	Ada Operation
Equal (x,y)	$x = y$
Add (x, y)	$x + y$
Multiply (x, y)	$x * y$
Negate (x)	$-x$
Reciprocal (x)	$1.0/x$
NonNegative (x)	$x \geq 0.0$
InOrder (x,y)	$x \leq y$

8.2.5 LI datatype: pcte-time

The LI datatype *pcte-time* is mapped to the Ada datatype `Pcte.calendar.time`.

type time is private;

The range and accuracy of `Pcte.calendar.time` is implementation-defined but must respect the conditions for conformance with ISO/IEC 13719-1. Conversions are supported between `CALENDAR.TIME` and `Pcte.calendar.time`. A specific accuracy (year, month, day, hour, minute, second, fractions of a second) is obtained by using such components of `Pcte.calendar` as are required through the functions `year`, `month`, `day`, `seconds`, and `time_of`, and the procedure `split` in the Ada package `Pcte.calendar`, together with appropriate arithmetic for the determination of hours, minutes and seconds.

Characterizing operations

Operation	Ada Operation
Equal (x,y)	$x = y$
InOrder (x,y)	$x \leq y$
Difference (x,y)	$x - y$
Extend.res1tores2 (x)	<code>Pcte.calendar.extend (x)</code>
Round.res1tores2 (x)	<code>Pcte.calendar.round (x)</code>

8.2.6 LI datatype: octet

The LI datatype *octet* is mapped to the Ada datatype `STANDARD.CHARACTER`.

Characterizing operations

Operation	Ada Operation
Equal (x,y)	x = y

8.2.7 LI datatype: pcte-text

The LI datatype *pcte-text* is mapped to the Ada datatype Pcte.text.

subtype text is STANDARD.STRING;

Characterizing operations

Operation	Ada Operation
Head (s)	s(s'FIRST)
Tail (s)	s(s'FIRST+1 .. s'LAST)
Equal (s1, s2)	s1 = s2
Empty (s)	""
Append (s, c)	s & c
IsEmpty (s)	s'LENGTH = 0

8.2.8 LI datatype generator: pcte-sequence

The LI datatypes of the family *pcte-sequence* are mapped to types declared by Ada packages. In general, each LI datatype *sequence of xxx* is mapped to a package called yyys (the plural of yyy) where yyy is the mapping of xxx, e.g. for an arbitrary type named element:

package elements is

type sequence is **limited private**;

- The initial value of an object of type sequence is an empty sequence, i.e. one of
- length 0. To discard a sequence so that the associated storage can be recovered,
- all the elements of the sequence are deleted by means of procedure discard.

function get (

```
list      : elements.sequence;
index     : Pcte.natural := Pcte.natural'FIRST;
status    : Pcte_error.handle := EXCEPTION_ONLY)
return   element;
```

- elements.get returns the element with the given index in the given sequence. If the
- index is not less than the number of elements of the sequence, then the error
- INVALID_INDEX is raised.

procedure insert (

```

list   : in out  elements.sequence;
item   : in      element;
index  : in      Pcte.natural := Pcte.natural'LAST;
status : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

-- elements.insert inserts the given element in the given sequence immediately before
-- the element with the given index, or if the index is not less than the number of
-- elements of the sequence, the given element is appended after the last element.

```

procedure replace (

```

list   : in out  elements.sequence;
item   : in      element;
index  : in      Pcte.natural := Pcte.natural'LAST;
status : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

-- elements.replace replaces the element with the given index by the given element, or if
-- the index is not less than the number of elements of the sequence, the given element
-- is appended after the last element.

```

procedure append (

```

list   : in out  elements.sequence;
item   : in      element;
status : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

-- elements.append appends the element to the sequence.

```

procedure delete (

```

list   : in out  elements.sequence;
index  : in      Pcte.natural := Pcte.natural'FIRST;
count  : in      Pcte.positive := Pcte.positive'LAST;
status : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

-- elements.delete deletes up to the given count elements from the element with the
-- given index from the given sequence. The number of elements deleted is the lesser
-- of count and the number of elements from the element of list with the given index to
-- the end.

```

procedure copy (

```

into_list : in out  elements.sequence;
from_list  : in      elements.sequence;
into_index : in      Pcte.natural := Pcte.natural'LAST;
from_index : in      Pcte.natural := Pcte.natural'FIRST;
count      : in      Pcte.positive := Pcte.positive'LAST;
status     : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

-- elements.copy adds up to the given count elements from the element with index
-- from_index of from_list to into_list. The elements are inserted immediately before
-- the element of into_list with index into_index, or, if into_index is not less than the
-- number of elements of into_list, are appended to the end of into_list. The number of
-- elements added is the lesser of count and the number of elements from the element of
-- from_list with index from_index to the end.

```

```

function length_of (
    list      : elements.sequence;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.natural;

-- elements.length_of returns the number of elements in the given sequence.

function index_of (
    list      : elements.sequence;
    item      : element;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.integer;

-- elements.index_of returns the index of the first occurrence of the given element in the
-- given sequence if the element is a member of the sequence, and -1 otherwise.

function are_equal (
    first     : elements.sequence;
    second    : elements.sequence;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.boolean;

-- elements.are_equal returns TRUE if the two sequences first and second have the
-- same number of elements and their corresponding elements are equal, and FALSE
-- otherwise.

procedure normalize (
    list      : in out elements.sequence;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

-- elements.normalize reorders the elements of the given sequence in an
-- implementation-defined canonical order, and deletes any duplicate elements.

procedure discard (
    list      : in out elements.sequence;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

private
    implementation-defined

end elements;

```

The Ada generated datatype sequence declared by the package is limited private. Objects of this type may only be manipulated by the subprograms defined in that package; in particular:

- assignment is not supported;
- all elements of such objects always have defined values.

Characterizing operations

Operation	Ada Operation
Equal(s1,s2)	are_equal(s1, s2)
Head(s)	get(s)
IsEmpty(s)	length_of(s) = 0
s:=Tail(s)	delete(s,1,1);
s:=Append(s,e)	append(s,e);
s:=Empty()	delete(s);
Get(s,i)	get(s,i)
s:=Put(s,e,i)	insert(s,e,i);
s:=Delete(s,i)	delete(s,i,1);
s1:=InsertSequence(s1,s2,i)	copy(s1,s2,i);
s1:=AppendSequence(s1,s2)	copy(s1,s2);
LengthOf(s)	length_of(s)
IsMember(s,e)	index_of(s,e) /= 0
s:=Normalize(s)	normalize(s);

8.2.9 LI datatype generator: bounded-set

The LI datatype *bounded-set of base* is mapped to an Ada datatype defined as an array of Pcte.boolean, with an enumeration index type mapping the names of element values to positions in the array. Each LI datatype *bounded set of xxx* is mapped to an Ada datatype called *set_of_xxxs* (where xxxs is the plural of xxx). An array value of the Ada datatype represents a set including a element value if the corresponding entry in the array has the value TRUE. Thus a bounded set of base type xxx with possible element values VAL1, VAL2, ... VALn is mapped as follows:

```
type xxx is (VAL1, VAL2, ... VALn);
type xxxs is array (xxx) of Pcte.boolean;
```

Characterizing operations

Operation	Ada Operation
IsIn (s,e)	s (e)
Subset (s1,s2)	for i in xxx loop if s1(i) and not s2(i) then return FALSE; end if ; end loop ; return TRUE;
Equal (s1,s2)	s1 = s2
Difference (s1, s2)	s1 and not s2
Union (s1,s2)	s1 or s2
Intersection (s1,s2)	s1 and s2
Empty ()	xxxs'(others => FALSE)
SetOf (e)	xxxs'(e => TRUE, others => FALSE)
Select (s)	j : xxx; for i in xxx loop j := i; exit when s(j); end loop ; return j;

8.2.10 LI datatype: record

The LI record datatype xxx = *record of (COMPONENT-1: type-1, COMPONENT-2: type-2, ...)* is mapped to the Ada record datatype:

type xxx is record

```

component_1:  type_1;
component_2:  type_2;
component_3:  type_3;
...;

```

end record;

Characterizing operations

Operation	Ada Operation
Equal (x, y)	x = y
FieldSelect.component (x)	x.component;
Aggregate (component_1, component_2, ...)	(component_1, component_2, ...)

8.2.11 LI datatype: private

The LI datatype private is mapped in each case to an Ada private type.

8.2.12 LI enumerated datatype: pcte-xxx

The LI datatype *pcte-xxx*, defined as *enumerated (val1, val2, ...)*, corresponds to the PCTE enumeration datatype xxx (where the values of xxx are VAL1, VAL2, ...). It is mapped to the Ada datatype xxx, defined as follows.

type xxx is (VAL1, VAL2, ...);

Characterizing operations

Operation	Ada Operation
Equal (x, y)	x = y
InOrder (x, y)	x <= y
Successor (x)	x'SUCC

8.3 Deriving Ada subprogram semantics from the abstract specification

Each Ada subprogram corresponds to the abstract operation in ISO/IEC 13719-1 whose name precedes the subprogram declaration, as a comment with the form of a clause heading from ISO/IEC 13719-1 e.g.: -- 11.2.9 VOLUME_MOUNT.

In cases where there is not a one-to-one correspondence between abstract operations and Ada subprograms, an explanation is given as a comment before the first Ada subprogram declaration. Where Ada subprogram declarations are out of the order with respect to order of ISO/IEC 13719-1, a cross-reference is given.

The semantics of an Ada subprogram is generally the same as that of the corresponding abstract operation, and is derived as follows in general. Any exceptions are described where they occurs in clause 9 to 22.

- Ada formal parameters of mode **in** correspond to abstract operation parameters of the same names.

- Ada formal parameter types correspond to the PCTE datatypes of the corresponding abstract operation parameters as defined in clause 8.
- Returned values corresponding to results of abstract operations are returned either as function results or as parameters of mode **out** or **in out**. In the latter case, if the result of an operation is a single non-optional value, then the Ada result parameter has the same name.
- Where an abstract operation has optional parameters, and in other cases where the above rules cannot be applied, an explanation is in general given as a comment immediately after the subprogram declaration. There are two general methods of mapping optional parameters and results where no explanation is given with the subprogram. One is where an optional parameter is of a string type and its absence is indicated by providing an empty string value. The other is where an optional parameter or result is dealt with by providing two overloaded subprograms, one with and one without the corresponding parameter.
- In order to use good Ada style, formal parameters with default values (corresponding to optional parameters of the abstract operation) are placed at the end of the parameter list. Otherwise the order of parameters is the same as for the abstract operation.
- In general the names of parameters and results are the same as in the abstract operation. Where that is not possible, because the name in the abstract operation is an Ada reserved word, or in the case of the name 'status' which is used specially in the Ada Binding, the characters 'pcte_' are prefixed to the name in the abstract operation.
- A parameter or result of a union type (excluding enumeration types) is usually handled by providing a set of overloaded subprograms, one for each member of the type union. No explanation is given with the subprogram declarations for such cases.
- Threads in the Abstract Specification are mapped onto tasks in the Ada binding. A single PCTE process executes by the execution of a single Ada program, even when that program includes multiple tasks. Within the process, Ada tasks execute in parallel (proceed independently) in accordance with the rules in ISO 8652 9(5).
- When a task executes a PCTE operation which depends on the occurrence of some event, that task is suspended pending the occurrence of that event. In this case, other tasks may continue to execute and to execute PCTE operations, subject to the Ada tasking rules. The suspended task has no delaying effect on other PCTE operations if the operations to be performed are unrelated.

8.4 Package Pcte

with Pcte_error, CALENDAR;

package Pcte is

use Pcte_error

----- PCTE basic types -----

subtype boolean **is** STANDARD.BOOLEAN;

-- Pcte.boolean corresponds to the PCTE datatype Boolean.

type integer **is range** *implementation-defined*;

```

-- Pcte.integer corresponds to the PCTE datatype Integer.
type natural is range 0 .. Pcte.integer'LAST;
-- Pcte.natural corresponds to the PCTE datatype Natural.
type float is digits implementation-defined range implementation-defined;
-- Pcte.float corresponds to the PCTE datatype Float.
subtype text is STANDARD.STRING;
-- Pcte.text corresponds to the PCTE datatype Text.
subtype string is STANDARD.STRING;
subtype string_length is STANDARD.NATURAL;
-- Pcte.string corresponds to the PCTE datatype String.
subtype key is Pcte.text;
-- Pcte.key corresponds to the PCTE datatype Key.
subtype actual_key is Pcte.text;
-- Pcte.actual_key corresponds to the PCTE datatype Actual_key.
subtype link_name is Pcte.text;
-- Pcte.link_name corresponds to the PCTE datatype Link_name.
subtype name is Pcte.text;
-- Pcte.name corresponds to the PCTE datatype Name.
subtype type_name is Pcte.text;
-- Pcte.type_name corresponds to the PCTE datatype Type_name.
subtype type_name_in_sds is Pcte.text (1..MAX_NAME_SIZE);
-- Pcte.type_name corresponds to the PCTE datatype Type_name_in_sds.
subtype positive is STANDARD.POSITIVE;
-- Pcte.positive is used to define a character position within Pcte.string or Pcte.text.
package calendar is
    type time is private;
    -- Pcte.calendar.time corresponds to the PCTE datatype Time; it is an Ada private
    -- type defined with its associated subprograms in this package Pcte.calendar. The
    -- subtypes and subprograms have the same meanings as their namesakes in package
    -- CALENDAR.
    DEFAULT_TIME: constant time;
    Pcte.calendar.DEFAULT_TIME is the default value of time attributes.
    subtype year_number is STANDARD.CALENDAR.YEAR_NUMBER
        range 1980 .. 2044;

```

subtype month_number **is** STANDARD.CALENDAR.MONTH_NUMBER;

subtype day_number **is** STANDARD.CALENDAR.DAY_NUMBER;

subtype day_duration **is** STANDARD.CALENDAR.DAY_DURATION;

function clock **return** Pcte.calendar.time;

function year (
 date : Pcte.calendar.time)
 return Pcte.calendar.year_number;

function month (
 date : Pcte.calendar.time)
 return Pcte.calendar.month_number;

function day (
 date : Pcte.calendar.time)
 return Pcte.calendar.day_number;

function seconds (
 date : Pcte.calendar.time)
 return Pcte.calendar.day_duration;

procedure split (
 date : **in** Pcte.calendar.time;
 year : **out** Pcte.calendar.year_number;
 month : **out** Pcte.calendar.month_number;
 day : **out** Pcte.calendar.day_number;
 seconds : **out** Pcte.calendar.day_duration);

function time_of (
 year : Pcte.calendar.year_number;
 month : Pcte.calendar.month_number;
 day : Pcte.calendar.day_number;
 seconds : Pcte.calendar.day_duration := 0.0)
 return Pcte.calendar.time;

function "+" (
 left : Pcte.calendar.time;
 right : STANDARD.DURATION)
 return Pcte.calendar.time;

function "+" (
 left : STANDARD.DURATION;
 right : Pcte.calendar.time)
 return Pcte.calendar.time;

function "-" (
 left : Pcte.calendar.time;
 right : STANDARD.DURATION)
 return Pcte.calendar.time;

```

function "-" (
  left      : Pcte.calendar.time;
  right     : Pcte.calendar.time)
return     STANDARD.DURATION;

function "<" (
  left      : Pcte.calendar.time;
  right     : Pcte.calendar.time)
return     Pcte.boolean;

function "<=" (
  left      : Pcte.calendar.time;
  right     : Pcte.calendar.time)
return     Pcte.boolean;

function ">" (
  left      : Pcte.calendar.time;
  right     : Pcte.calendar.time)
return     Pcte.boolean;

function ">=" (
  left      : Pcte.calendar.time;
  right     : Pcte.calendar.time)
return     Pcte.boolean;

function extend (
  date      : Pcte.calendar.time)
return     STANDARD.CALENDAR.TIME;

-- Pcte.calendar.extend converts the value of the parameter date to the type
-- CALENDAR.TIME.

function round (
  date      : STANDARD.CALENDAR.TIME)
return     Pcte.calendar.time;

-- Pcte.calendar.round converts the value of the parameter date to the type
-- Pcte.calendar.time.

TIME_ERROR : exception renames STANDARD.CALENDAR.TIME_ERROR;

private
  implementation-defined
end calendar;

```

----- PCTE references -----

package reference is

```

type  object_ref  is limited private;
type  link_ref    is limited private;
type  type_ref    is limited private;

```

```

subtype attribute_ref is type_ref;
CURRENT_PROCESS      : constant Pcte.reference.object_ref;
LOCAL_WORKSTATION    : constant Pcte.reference.object_ref;
LOCAL_EXECUTION_SITE : constant Pcte.reference.object_ref;
subtype pathname is Pcte.string;
-- Pcte.reference.pathname corresponds to the PCTE datatype Pathname.
subtype relative_pathname is Pcte.string;
-- Pcte.reference.relative_pathname corresponds to the PCTE datatype
-- Relative_pathname.
type evaluation_point is (NOW, FIRST_USE, EVERY_USE);
-- Pcte.reference.evaluation_point corresponds to the PCTE datatype
-- Evaluation_point.
type evaluation_status is (INTERNAL, EXTERNAL, UNDEFINED);
-- Pcte.reference.evaluation_status corresponds to the PCTE datatype
-- Evaluation_status. The value UNDEFINED is binding-defined and signifies an
-- object reference which is unset.
type reference_equality is (EQUAL_REFS, UNEQUAL_REFS, EXTERNAL_REFS);
-- Pcte.reference.reference_equality corresponds to the PCTE datatype
-- Reference_equality.
type key_kind is (NATURAL_KEY, STRING_KEY);
type key_value (
    kind      : Pcte.reference.key_kind := NATURAL_KEY;
    string_length : Pcte.string_length := 0)
is record
    case kind is
        when NATURAL_KEY => natural_value : Pcte.natural;
        when STRING_KEY => string_value   : Pcte.text (1..string_length);
    end case;
end record;

```

----- Object Reference Operations -----

-- 23.2.1 OBJECT_REFERENCE_COPY

```

procedure copy (
    reference      : in    Pcte.reference.object_ref;
    point          : in    Pcte.reference.evaluation_point;
    new_reference  : out   Pcte.reference.object_ref;
    status         : in    Pcte_error.handle := EXCEPTION_ONLY);

```

-- 23.2.2 OBJECT_REFERENCE_GET_EVALUATION_POINT

```
function get_evaluation_point (
    reference : Pcte.reference.object_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.reference.evaluation_point;
```

-- 23.2.3 OBJECT_REFERENCE_GET_PATH

```
function get_path (
    reference : Pcte.reference.object_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.reference.pathname;
```

-- 23.2.4 OBJECT_REFERENCE_GET_STATUS

```
function get_status (
    reference : Pcte.reference.object_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.reference.evaluation_status;
```

-- 23.2.5 OBJECT_REFERENCE_SET_ABSOLUTE

```
procedure set_absolute (
    pathname      : in      Pcte.reference.pathname;
    point         : in      Pcte.reference.evaluation_point;
    new_reference : out     Pcte.reference.object_ref;
    status        : in      Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.2.6 OBJECT_REFERENCE_SET_RELATIVE

```
procedure set_relative (
    reference      : in      Pcte.reference.object_ref;
    pathname      : in      Pcte.reference.relative_pathname;
    point         : in      Pcte.reference.evaluation_point;
    new_reference : out     Pcte.reference.object_ref;
    status        : in      Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.2.7 OBJECT_REFERENCE_UNSET

```
procedure unset (
    reference : in out Pcte.reference.object_ref;
    status    : in     Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.2.8 OBJECT_REFERENCES_ARE_EQUAL

```
function are_equal (
    first_reference : Pcte.reference.object_ref;
    second_reference : Pcte.reference.object_ref;
    status          : Pcte_error.handle := EXCEPTION_ONLY)
return            Pcte.reference.reference_equality;
```

----- Link Reference Operations -----

-- 23.3.1 LINK_REFERENCE_COPY

```
procedure copy (
    reference      : in    Pcte.reference.link_ref;
    point          : in    Pcte.reference.evaluation_point;
    new_reference  : out   Pcte.reference.link_ref;
    status         : in    Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.3.2 LINK_REFERENCE_GET_EVALUATION_POINT

```
function get_evaluation_point (
    reference : Pcte.reference.link_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.reference.evaluation_point;
```

-- 23.3.3 LINK_REFERENCE_GET_KEY

```
function get_key (
    reference : Pcte.reference.link_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.key;
```

-- 23.3.4 LINK_REFERENCE_GET_KEY_VALUE

```
function get_key_value (
    reference : Pcte.reference.link_ref;
    index     : Pcte.natural;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.reference.key_value;
```

-- 23.3.5 LINK_REFERENCE_GET_NAME

```
function get_name (
    reference : Pcte.reference.link_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.link_name;
```

-- 23.3.6 LINK_REFERENCE_GET_STATUS

```
function get_status (
    reference : Pcte.reference.link_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.reference.evaluation_status;
```

-- 23.3.7 LINK_REFERENCE_GET_TYPE

```
procedure get_type (
    reference      : in    Pcte.reference.link_ref;
    type_reference : out   Pcte.reference.type_ref;
    status         : in    Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.3.8 LINK_REFERENCE_SET

```
procedure set (
  link_name      : in    Pcte.link_name;
  point          : in    Pcte.reference.evaluation_point;
  new_reference  : out   Pcte.reference.link_ref;
  status         : in    Pcte_error.handle := EXCEPTION_ONLY);
```

```
procedure set (
  link_name      : in    Pcte.reference.type_ref;
  point          : in    Pcte.reference.evaluation_point;
  new_reference  : out   Pcte.reference.link_ref;
  status         : in    Pcte_error.handle := EXCEPTION_ONLY);
```

```
procedure set (
  link_name      : in    Pcte.reference.type_ref;
  link_key       : in    Pcte.key;
  point          : in    Pcte.reference.evaluation_point;
  new_reference  : out   Pcte.reference.link_ref;
  status         : in    Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.3.9 LINK_REFERENCE_UNSET

```
procedure unset (
  reference : in out Pcte.reference.link_ref;
  status    : in    Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.3.10 LINK_REFERENCES_ARE_EQUAL

```
function are_equal (
  first_reference  : Pcte.reference.link_ref;
  second_reference : Pcte.reference.link_ref;
  status          : Pcte_error.handle := EXCEPTION_ONLY)
return           Pcte.reference.reference_equality;
```

----- Type reference operations -----

-- 23.4.1 TYPE_REFERENCE_COPY

```
procedure copy (
  reference      : in    Pcte.reference.type_ref;
  point          : in    Pcte.reference.evaluation_point;
  new_reference  : out   Pcte.reference.type_ref;
  status         : in    Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.4.2 TYPE_REFERENCE_GET_EVALUATION_POINT

```
function get_evaluation_point (
  reference : Pcte.reference.type_ref;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
return     Pcte.reference.evaluation_point;
```

-- 23.4.3 TYPE_REFERENCE_GET_IDENTIFIER

```
function get_identifier (
    reference : Pcte.reference.type_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.type_name;
```

-- 23.4.4 TYPE_REFERENCE_GET_NAME

```
function get_name (
    reference : Pcte.reference.type_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.type_name;
```

```
function get_name (
    sds       : Pcte.reference.object_ref;
    reference : Pcte.reference.type_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.type_name;
```

-- 23.4.5 TYPE_REFERENCE_GET_STATUS

```
function get_status (
    reference : Pcte.reference.type_ref;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.reference.evaluation_status;
```

-- 23.4.6 TYPE_REFERENCE_SET

```
procedure set (
    name           : in    Pcte.type_name;
    point          : in    Pcte.reference.evaluation_point;
    new_reference  : out   Pcte.reference.type_ref;
    status         : in    Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.4.7 TYPE_REFERENCE_UNSET

```
procedure unset (
    reference : in out Pcte.reference.type_ref;
    status    : in     Pcte_error.handle := EXCEPTION_ONLY);
```

-- 23.4.8 TYPE_REFERENCES_ARE_EQUAL

```
function are_equal (
    first_reference    : Pcte.reference.type_ref;
    second_reference   : Pcte.reference.type_ref;
    status             : Pcte_error.handle := EXCEPTION_ONLY)
return               Pcte.reference.reference_equality;
```

private

implementation-defined

end reference;

subtype object_reference **is** Pcte.reference.object_ref;

subtype type_reference **is** Pcte.reference.type_ref;

subtype link_reference **is** Pcte.reference.link_ref;

subtype attribute_reference **is** Pcte.reference.attribute_ref;

-- These types correspond to the PCTE datatypes X_reference.

----- PCTE types related to attributes -----

type enumeral_position **is range** 0 .. Pcte.natural'LAST;

type value_type **is** (INTEGER_TYPE, NATURAL_TYPE, BOOLEAN_TYPE,
TIME_TYPE, FLOAT_TYPE, STRING_TYPE, ENUMERAL_TYPE);

-- Pcte.value_type corresponds to the PCTE datatype Value_type. ENUMERAL_TYPE

-- indicates an enumeral type, defined by its position in the corresponding attribute type.

type attribute_value (

type_is : Pcte.value_type := INTEGER_TYPE;

string_length : Pcte.string_length := 0)

is record

case type_is **is**

when BOOLEAN_TYPE =>

boolean_value : Pcte.boolean;

when NATURAL_TYPE =>

natural_value : Pcte.natural;

when INTEGER_TYPE =>

integer_value : Pcte.integer;

when FLOAT_TYPE =>

float_value : Pcte.float;

when TIME_TYPE =>

time_value : Pcte.calendar.time;

when STRING_TYPE =>

string_value : Pcte.string(1..string_length);

when ENUMERAL_TYPE =>

enumeral_value : Pcte.enumeral_position;

end case;

end record;

type attribute_assignment (

type_is : Pcte.value_type := INTEGER_TYPE;

string_length : Pcte.string_length := 0)

is record

attribute : Pcte.type_reference;

value : Pcte.attribute_value(type_is, string_length);

end record;

- Pcte.attribute_assignment corresponds to the PCTE datatype Attribute_assignment.
- Pcte.attribute_value is the type of the value field, and is discriminated by the type of the attribute value.

----- PCTE types related to links -----

type category **is** (COMPOSITION_LINK, EXISTENCE_LINK, REFERENCE_LINK, DESIGNATION_LINK, IMPLICIT_LINK);

- Pcte.category corresponds to the PCTE datatype Category.

type categories **is array** (Pcte.category) **of** Pcte.boolean;

- Pcte.categories corresponds to the PCTE datatype Categories.

type object_scope **is** (ATOMIC, COMPOSITE);

- Pcte.object_scope corresponds to the PCTE datatype Object_scope.

----- PCTE types related to objects -----

subtype exact_identifier **is** Pcte.text;

- Pcte.exact_identifier corresponds to the PCTE datatype Exact_identifier.

----- sequence packages -----

- The semantics of the operations of these packages are defined in 8.2.8.

package object_references **is**

type sequence **is limited private**;

- Pcte.object_references.sequence corresponds to the PCTE datatype Object_references.

function get (

list : Pcte.object_references.sequence;
 index : Pcte.natural := Pcte.natural'FIRST;
 status : Pcte_error.handle := EXCEPTION_ONLY)
return Pcte.object_reference;

procedure insert (

list : **in out** Pcte.object_references.sequence;
 item : **in** Pcte.object_reference;
 index : **in** Pcte.natural := Pcte.natural'LAST;
 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

procedure replace (

list : **in out** Pcte.object_references.sequence;
 item : **in** Pcte.object_reference;
 index : **in** Pcte.natural := Pcte.natural'LAST;
 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

```

procedure append (
    list      : in out  Pcte.object_references.sequence;
    item      : in      Pcte.object_reference;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

procedure delete (
    list      : in out  Pcte.object_references.sequence;
    index     : in      Pcte.natural := Pcte.natural'FIRST;
    count     : in      Pcte.positive := Pcte.positive'LAST;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

procedure copy (
    into_list  : in out  Pcte.object_references.sequence;
    from_list  : in      Pcte.object_references.sequence;
    into_index : in      Pcte.natural := Pcte.natural'LAST;
    from_index : in      Pcte.natural := Pcte.natural'FIRST;
    count      : in      Pcte.positive := Pcte.positive'LAST;
    status     : in      Pcte_error.handle := EXCEPTION_ONLY);

function length_of (
    list      : Pcte.object_references.sequence;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.natural;

function index_of (
    list      : Pcte.object_references.sequence;
    item      : Pcte.object_reference;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.integer;

function are_equal (
    first     : Pcte.object_references.sequence;
    second    : Pcte.object_references.sequence;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.boolean;

procedure normalize (
    list      : in out  Pcte.object_references.sequence;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

procedure discard (
    list      : in out  Pcte.object_references.sequence;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

private
    implementation-defined

end object_references;

package link_references is
    type sequence is limited private;

```

-- Pcte.link_references.sequence corresponds to the PCTE datatype Link_references.

function get (

list : Pcte.link_references.sequence;
 index : Pcte.natural := Pcte.natural'FIRST;
 status : Pcte_error.handle := EXCEPTION_ONLY)
return Pcte.link_reference;

procedure insert (

list : **in out** Pcte.link_references.sequence;
 item : **in** Pcte.link_reference;
 index : **in** Pcte.natural := Pcte.natural'LAST;
 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

procedure replace (

list : **in out** Pcte.link_references.sequence;
 item : **in** Pcte.link_reference;
 index : **in** Pcte.natural := Pcte.natural'LAST;
 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

procedure append (

list : **in out** Pcte.link_references.sequence;
 item : **in** Pcte.link_reference;
 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

procedure delete (

list : **in out** Pcte.link_references.sequence;
 index : **in** Pcte.natural := Pcte.natural'FIRST;
 count : **in** Pcte.positive := Pcte.positive'LAST;
 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

procedure copy (

into_list : **in out** Pcte.link_references.sequence;
 from_list : **in** Pcte.link_references.sequence;
 into_index : **in** Pcte.natural := Pcte.natural'LAST;
 from_index : **in** Pcte.natural := Pcte.natural'FIRST;
 count : **in** Pcte.positive := Pcte.positive'LAST;
 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

function length_of (

list : Pcte.link_references.sequence;
 status : Pcte_error.handle := EXCEPTION_ONLY)
return Pcte.natural;

function index_of (

list : Pcte.link_references.sequence;
 item : Pcte.link_reference;
 status : Pcte_error.handle := EXCEPTION_ONLY)
return Pcte.integer;

```

function are_equal (
    first      : Pcte.link_references.sequence;
    second     : Pcte.link_references.sequence;
    status     : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.boolean;

procedure normalize (
    list       : in out Pcte.link_references.sequence;
    status     : in     Pcte_error.handle := EXCEPTION_ONLY);

procedure discard (
    list       : in out Pcte.link_references.sequence;
    status     : in     Pcte_error.handle := EXCEPTION_ONLY);

private
    implementation-defined

end link_references;

package type_references is
    type sequence is limited private;
    -- Pcte.type_references.sequence corresponds to the PCTE datatype Type_references.

    function get (
        list      : Pcte.type_references.sequence;
        index     : Pcte.natural := Pcte.natural'FIRST;
        status    : Pcte_error.handle := EXCEPTION_ONLY)
        return     Pcte.type_reference;

    procedure insert (
        list      : in out Pcte.type_references.sequence;
        item      : in     Pcte.type_reference;
        index     : in     Pcte.natural := Pcte.natural'LAST;
        status    : in     Pcte_error.handle := EXCEPTION_ONLY);

    procedure replace (
        list      : in out Pcte.type_references.sequence;
        item      : in     Pcte.type_reference;
        index     : in     Pcte.natural := Pcte.natural'LAST;
        status    : in     Pcte_error.handle := EXCEPTION_ONLY);

    procedure append (
        list      : in out Pcte.type_references.sequence;
        item      : in     Pcte.type_reference;
        status    : in     Pcte_error.handle := EXCEPTION_ONLY);

    procedure delete (
        list      : in out Pcte.type_references.sequence;
        index     : in     Pcte.natural := Pcte.natural'FIRST;
        count     : in     Pcte.positive := Pcte.positive'LAST;
        status    : in     Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure copy (
  into_list      : in out  Pcte.type_references.sequence;
  from_list      : in       Pcte.type_references.sequence;
  into_index     : in       Pcte.natural := Pcte.natural'LAST;
  from_index     : in       Pcte.natural := Pcte.natural'FIRST;
  count          : in       Pcte.positive := Pcte.positive'LAST;
  status         : in       Pcte_error.handle := EXCEPTION_ONLY);

```

```

function length_of (
  list          : Pcte.type_references.sequence;
  status        : Pcte_error.handle := EXCEPTION_ONLY)
return         Pcte.natural;

```

```

function index_of (
  list          : Pcte.type_references.sequence;
  item          : Pcte.type_reference;
  status        : Pcte_error.handle := EXCEPTION_ONLY)
return         Pcte.integer;

```

```

function are_equal (
  first         : Pcte.type_references.sequence;
  second        : Pcte.type_references.sequence;
  status        : Pcte_error.handle := EXCEPTION_ONLY)
return         Pcte.boolean;

```

```

procedure normalize (
  list          : in out  Pcte.type_references.sequence;
  status        : in       Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure discard (
  list          : in out  Pcte.type_references.sequence;
  status        : in       Pcte_error.handle := EXCEPTION_ONLY);

```

private

implementation-defined

end type_references;

package type_names_in_sds **is**

type sequence **is limited private**;

```

-- Pcte.type_names_in_sds.sequence corresponds to the PCTE datatype
-- Type_names_in_sds.

```

```

function get (
  list          : Pcte.type_names_in_sds.sequence;
  index         : Pcte.natural := Pcte.natural'FIRST;
  status        : Pcte_error.handle := EXCEPTION_ONLY)
return         Pcte.type_name_in_sds;

```

```

procedure insert (
    list      : in out    Pcte.type_names_in_sds.sequence;
    item      : in        Pcte.type_name_in_sds;
    index     : in        Pcte.natural := Pcte.natural'LAST;
    status    : in        Pcte_error.handle := EXCEPTION_ONLY);

procedure replace (
    list      : in out    Pcte.type_names_in_sds.sequence;
    item      : in        Pcte.type_name_in_sds;
    index     : in        Pcte.natural := Pcte.natural'LAST;
    status    : in        Pcte_error.handle := EXCEPTION_ONLY);

procedure append (
    list      : in out    Pcte.type_names_in_sds.sequence;
    item      : in        Pcte.type_name_in_sds;
    status    : in        Pcte_error.handle := EXCEPTION_ONLY);

procedure delete (
    list      : in out    Pcte.type_names_in_sds.sequence;
    index     : in        Pcte.natural := Pcte.natural'FIRST;
    count     : in        Pcte.positive := Pcte.positive'LAST;
    status    : in        Pcte_error.handle := EXCEPTION_ONLY);

procedure copy (
    into_list  : in out    Pcte.type_names_in_sds.sequence;
    from_list  : in        Pcte.type_names_in_sds.sequence;
    into_index : in        Pcte.natural := Pcte.natural'LAST;
    from_index : in        Pcte.natural := Pcte.natural'FIRST;
    count      : in        Pcte.positive := Pcte.positive'LAST;
    status     : in        Pcte_error.handle := EXCEPTION_ONLY);

function length_of (
    list      : Pcte.type_names_in_sds.sequence;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.natural;

function index_of (
    list      : Pcte.type_names_in_sds.sequence;
    item      : Pcte.type_name_in_sds;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.integer;

function index_of (
    list      : Pcte.type_names_in_sds.sequence;
    item      : Pcte.type_name_in_sds;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
    return     Pcte.integer;

procedure normalize (
    list      : in out    Pcte.type_names_in_sds.sequence;
    status    : in        Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure discard (
    list      : in out  Pcte.type_names_in_sds.sequence;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

private

implementation-defined

end type_names_in_sds;

package attribute_references **is**

type sequence **is limited private**;

```

-- Pcte.attribute_references.sequence corresponds to the PCTE datatype
-- Attribute_references.

```

```

function get (
    list      : Pcte.attribute_references.sequence;
    index     : Pcte.natural := Pcte.natural'FIRST;
    status    : Pcte_error.handle := EXCEPTION_ONLY)
return      Pcte.attribute_reference;

```

```

procedure insert (
    list      : in out  Pcte.attribute_references.sequence;
    item      : in      Pcte.attribute_reference;
    index     : in      Pcte.natural := Pcte.natural'LAST;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure replace (
    list      : in out  Pcte.attribute_references.sequence;
    item      : in      Pcte.attribute_reference;
    index     : in      Pcte.natural := Pcte.natural'LAST;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure append (
    list      : in out  Pcte.attribute_references.sequence;
    item      : in      Pcte.attribute_reference;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure delete (
    list      : in out  Pcte.attribute_references.sequence;
    index     : in      Pcte.natural := Pcte.natural'FIRST;
    count     : in      Pcte.positive := Pcte.positive'LAST;
    status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure copy (
    into_list  : in out  Pcte.attribute_references.sequence;
    from_list  : in      Pcte.attribute_references.sequence;
    into_index : in      Pcte.natural := Pcte.natural'LAST;
    from_index : in      Pcte.natural := Pcte.natural'FIRST;
    count      : in      Pcte.positive := Pcte.positive'LAST;
    status     : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

function length_of (
  list      : Pcte.attribute_references.sequence;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
return     Pcte.natural;

function index_of (
  list      : Pcte.attribute_references.sequence;
  item      : Pcte.attribute_reference;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
return     Pcte.integer;

function are_equal (
  first     : Pcte.attribute_references.sequence;
  second    : Pcte.attribute_references.sequence;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
return     Pcte.boolean;

procedure normalize (
  list      : in out Pcte.attribute_references.sequence;
  status    : in     Pcte_error.handle := EXCEPTION_ONLY);

procedure discard (
  list      : in out Pcte.attribute_references.sequence;
  status    : in     Pcte_error.handle := EXCEPTION_ONLY);

private
  implementation-defined

end attribute_references;

```

package attribute_assignments **is**

type sequence **is limited private**;

-- Pcte.attribute_assignments.sequence corresponds to the PCTE datatype
 -- Attribute_assignments.

```

function get (
  list      : Pcte.attribute_assignments.sequence;
  index     : Pcte.natural := Pcte.natural'FIRST;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
return     Pcte.attribute_assignment;

```

```

procedure insert (
  list      : in out Pcte.attribute_assignments.sequence;
  item      : in     Pcte.attribute_assignment;
  index     : in     Pcte.natural := Pcte.natural'LAST;
  status    : in     Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure replace (
  list      : in out  Pcte.attribute_assignments.sequence;
  item      : in      Pcte.attribute_assignment;
  index     : in      Pcte.natural := Pcte.natural'LAST;
  status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure append (
  list      : in out  Pcte.attribute_assignments.sequence;
  item      : in      Pcte.attribute_assignment;
  status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure delete (
  list      : in out  Pcte.attribute_assignments.sequence;
  index     : in      Pcte.natural := Pcte.natural'FIRST;
  count     : in      Pcte.positive := Pcte.positive'LAST;
  status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure copy (
  into_list  : in out  Pcte.attribute_assignments.sequence;
  from_list  : in      Pcte.attribute_assignments.sequence;
  into_index : in      Pcte.natural := Pcte.natural'LAST;
  from_index : in      Pcte.natural := Pcte.natural'FIRST;
  count      : in      Pcte.positive := Pcte.positive'LAST;
  status     : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

function length_of (
  list      : Pcte.attribute_assignments.sequence;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
return     Pcte.natural;

```

```

function index_of (
  list      : Pcte.attribute_assignments.sequence;
  item      : Pcte.attribute_assignment;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
return     Pcte.integer;

```

```

function are_equal (
  first      : Pcte.attribute_assignments.sequence;
  second     : Pcte.attribute_assignments.sequence;
  status     : Pcte_error.handle := EXCEPTION_ONLY)
return     Pcte.boolean;

```

```

procedure normalize (
  list      : in out  Pcte.attribute_assignments.sequence;
  status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure discard (
  list      : in out  Pcte.attribute_assignments.sequence;
  status    : in      Pcte_error.handle := EXCEPTION_ONLY);

```

private

implementation-defined

```

    end attribute_assignments;
end Pcte;

```

9 Object management

9.1 Object management datatypes

```
-- See the beginning of the packages Pcte_link, Pcte_object and Pcte_version.
```

9.2 Link operations

```
with Pcte, Pcte_error, Pcte_archive, Pcte_volume;
```

```
package Pcte_link is
```

```
    use Pcte, Pcte.calendar, Pcte_error;
```

```
-- 9.2.1 LINK_CREATE
```

```

procedure create (
    origin      : in    Pcte.object_reference;
    new_link    : in    Pcte.link_reference;
    dest        : in    Pcte.object_reference;
    reverse_key : in    Pcte.actual_key := "";
    status      : in    Pcte_error.handle := EXCEPTION_ONLY);

```

```
-- 9.2.2 LINK_DELETE
```

```

procedure delete (
    origin : in    Pcte.object_reference;
    link   : in    Pcte.link_reference;
    status : in    Pcte_error.handle := EXCEPTION_ONLY);

```

```
-- 9.2.3 LINK_DELETE_ATTRIBUTE
```

```

procedure delete_attribute (
    origin   : in    Pcte.object_reference;
    link     : in    Pcte.link_reference;
    attribute : in    Pcte.attribute_reference;
    status   : in    Pcte_error.handle := EXCEPTION_ONLY);

```

```
-- 9.2.4 LINK_GET_ATTRIBUTE
```

```

-- LINK_GET_ATTRIBUTE is mapped to overloaded versions of the function
-- Pcte_link.get_attribute, one for each possible type of the result. If the type of
-- the result of the particular overloaded function that is used is different from the
-- type of its parameter attribute, then the error VALUE_TYPE_IS_INVALID is
-- raised.

```

```
function get_attribute (  
  origin      : Pcte.object_reference;  
  link        : Pcte.link_reference;  
  attribute    : Pcte.attribute_reference;  
  status      : Pcte_error.handle := EXCEPTION_ONLY)  
return      Pcte.boolean;
```

```
function get_attribute (  
  origin      : Pcte.object_reference;  
  link        : Pcte.link_reference;  
  attribute    : Pcte.attribute_reference;  
  status      : Pcte_error.handle := EXCEPTION_ONLY)  
return      Pcte.integer;
```

```
function get_attribute (  
  origin      : Pcte.object_reference;  
  link        : Pcte.link_reference;  
  attribute    : Pcte.attribute_reference;  
  status      : Pcte_error.handle := EXCEPTION_ONLY)  
return      Pcte.natural;
```

```
function get_attribute (  
  origin      : Pcte.object_reference;  
  link        : Pcte.link_reference;  
  attribute    : Pcte.attribute_reference;  
  status      : Pcte_error.handle := EXCEPTION_ONLY)  
return      Pcte.float;
```

```
function get_attribute (  
  origin      : Pcte.object_reference;  
  link        : Pcte.link_reference;  
  attribute    : Pcte.attribute_reference;  
  status      : Pcte_error.handle := EXCEPTION_ONLY)  
return      Pcte.calendar.time;
```

```
function get_attribute (  
  origin      : Pcte.object_reference;  
  link        : Pcte.link_reference;  
  attribute    : Pcte.attribute_reference;  
  status      : Pcte_error.handle := EXCEPTION_ONLY)  
return      Pcte.enumeral_position;
```

```
function get_attribute (  
  origin      : Pcte.object_reference;  
  link        : Pcte.link_reference;  
  attribute    : Pcte.attribute_reference;  
  status      : Pcte_error.handle := EXCEPTION_ONLY)  
return      Pcte.string;
```

-- 9.2.5 LINK_GET_DESTINATION_VOLUME

```
function get_destination_volume (
    origin      : Pcte.object_reference;
    link        : Pcte.link_reference;
    status      : Pcte_error.handle := EXCEPTION_ONLY)
return       Pcte_volume.volume_info;
```

-- 9.2.6 LINK_GET_KEY

```
function get_key (
    origin      : Pcte.object_reference;
    link        : Pcte.link_reference;
    status      : Pcte_error.handle := EXCEPTION_ONLY)
return       Pcte.actual_key;
```

-- 9.2.7 LINK_GET_REVERSE

```
procedure get_reverse (
    origin      : in      Pcte.object_reference;
    link        : in      Pcte.link_reference;
    reverse_link : in out Pcte.link_reference;
    dest        : in out Pcte.object_reference;
    status      : in      Pcte_error.handle := EXCEPTION_ONLY);
```

-- 9.2.8 LINK_GET_SEVERAL_ATTRIBUTES

-- The effect of assigning VISIBLE_ATTRIBUTE_TYPES to the parameter *attributes* is
 -- achieved by the first overloaded subprogram; the effect of assigning
 -- Attribute_designators to the parameter *attributes* is achieved by the second overloaded
 -- subprogram.

```
procedure get_several_attributes (
    origin : in      Pcte.object_reference;
    link   : in      Pcte.link_reference;
    values : in out Pcte.attribute_assignments.sequence;
    status : in      Pcte_error.handle := EXCEPTION_ONLY);
```

```
procedure get_several_attributes (
    origin : in      Pcte.object_reference;
    link   : in      Pcte.link_reference;
    attributes : in      Pcte.attribute_references.sequence;
    values : in out Pcte.attribute_assignments.sequence;
    status : in      Pcte_error.handle := EXCEPTION_ONLY);
```

-- 9.2.9 LINK_REPLACE

```
procedure replace (
    origin      : in    Pcte.object_reference;
    link        : in    Pcte.link_reference;
    new_origin  : in    Pcte.object_reference;
    new_link    : in    Pcte.link_reference;
    new_reverse_key: in  Pcte.actual_key := "";
    status      : in    Pcte_error.handle := EXCEPTION_ONLY);
```

-- 9.2.10 LINK_RESET_ATTRIBUTE

```
procedure reset_attribute (
    origin  : in    Pcte.object_reference;
    link    : in    Pcte.link_reference;
    attribute : in  Pcte.attribute_reference;
    status   : in  Pcte_error.handle := EXCEPTION_ONLY);
```

-- 9.2.11 LINK_SET_ATTRIBUTE

-- LINK_SET_ATTRIBUTE is mapped to overloaded versions of the procedure
 -- Pcte_link.set_attribute, one for each possible type of the parameter value. If the type
 -- of the parameter value of the particular overloaded procedure that is used is different
 -- from the type of its parameter attribute, then the error VALUE_TYPE_IS_INVALID is
 -- raised.

```
procedure set_attribute (
    origin  : in    Pcte.object_reference;
    link    : in    Pcte.link_reference;
    attribute : in  Pcte.attribute_reference;
    value    : in  Pcte.boolean := FALSE;
    status   : in  Pcte_error.handle := EXCEPTION_ONLY);
```

```
procedure set_attribute (
    origin  : in    Pcte.object_reference;
    link    : in    Pcte.link_reference;
    attribute : in  Pcte.attribute_reference;
    value    : in  Pcte.integer := 0;
    status   : in  Pcte_error.handle := EXCEPTION_ONLY);
```

```
procedure set_attribute (
    origin  : in    Pcte.object_reference;
    link    : in    Pcte.link_reference;
    attribute : in  Pcte.attribute_reference;
    value    : in  Pcte.natural := 0;
    status   : in  Pcte_error.handle := EXCEPTION_ONLY);
```

```
procedure set_attribute (
  origin      : in    Pcte.object_reference;
  link        : in    Pcte.link_reference;
  attribute   : in    Pcte.attribute_reference;
  value       : in    Pcte.float := 0.0;
  status      : in    Pcte_error.handle := EXCEPTION_ONLY);
```

```
procedure set_attribute (
  origin      : in    Pcte.object_reference;
  link        : in    Pcte.link_reference;
  attribute   : in    Pcte.attribute_reference;
  value       : in    Pcte.string := "";
  status      : in    Pcte_error.handle := EXCEPTION_ONLY);
```

```
procedure set_attribute (
  origin      : in    Pcte.object_reference;
  link        : in    Pcte.link_reference;
  attribute   : in    Pcte.attribute_reference;
  value       : in    Pcte.calendar.time := DEFAULT_TIME;
  status      : in    Pcte_error.handle := EXCEPTION_ONLY);
```

```
procedure set_attribute (
  origin      : in    Pcte.object_reference;
  link        : in    Pcte.link_reference;
  attribute   : in    Pcte.attribute_reference;
  value       : in    Pcte.enumeral_position;
  status      : in    Pcte_error.handle := EXCEPTION_ONLY);
```

-- 9.2.12 LINK_SET_SEVERAL_ATTRIBUTES

```
procedure set_several_attributes (
  origin      : in    Pcte.object_reference;
  link        : in    Pcte.link_reference;
  attributes   : in    Pcte.attribute_assignments.sequence;
  status      : in    Pcte_error.handle := EXCEPTION_ONLY);
```

-- 11.2.7 LINK_GET_DESTINATION_ARCHIVE

```
function get_destination_archive (
  origin      : Pcte.object_reference;
  link        : Pcte.link_reference;
  status      : Pcte_error.handle := EXCEPTION_ONLY)
  return      Pcte_archive.archive_identifier;
```

end Pcte_link;

9.3 Object operations

with Pcte, Pcte_error, Pcte_discretionary, Pcte_volume;

package Pcte_object **is**

```

use Pcte, Pcte.calendar, Pcte_error;

type type_ancestry is (EQUAL_TYPE, ANCESTOR_TYPE, DESCENDANT_TYPE,
    UNRELATED_TYPE);

-- Pcte_object.type_ancestry corresponds to the PCTE datatype Type_ancestry.
type link_scope is (INTERNAL_LINKS, EXTERNAL_LINKS, ALL_LINKS);

-- Pcte_object.link_scope corresponds to the PCTE datatype Link_scope.

type link_set_descriptor is record
    origin : Pcte.object_reference;
    links  : Pcte.link_references.sequence;
end record;

-- Pcte_object.link_set_descriptor corresponds to the PCTE datatype Link_set_descriptor.

-- The semantics of the operations of this package are defined in 8.2.8.

package link_set_descriptors is
    type sequence is limited private;

    -- Pcte_object.link_set_descriptor.sequence corresponds to the PCTE datatype
    -- Link_set_descriptors.

    function get (
        list      : Pcte_object.link_set_descriptors.sequence;
        index     : Pcte.natural := Pcte.natural'FIRST;
        status    : Pcte_error.handle := EXCEPTION_ONLY)
        return Pcte_object.link_set_descriptor;

    procedure insert (
        list      : in out Pcte_object.link_set_descriptors.sequence;
        item      : in Pcte_object.link_set_descriptor;
        index     : in Pcte.natural := Pcte.natural'LAST;
        status    : in Pcte_error.handle := EXCEPTION_ONLY);

    procedure replace (
        list      : in out Pcte_object.link_set_descriptors.sequence;
        item      : in Pcte_object.link_set_descriptor;
        index     : in Pcte.natural := Pcte.natural'LAST;
        status    : in Pcte_error.handle := EXCEPTION_ONLY);

    procedure append (
        list      : in out Pcte_object.link_set_descriptors.sequence;
        item      : in Pcte_object.link_set_descriptor;
        status    : in Pcte_error.handle := EXCEPTION_ONLY);

```

```

procedure delete (
  list      : in out  Pcte_object.link_set_descriptors.sequence;
  index     : in      Pcte.natural := Pcte.natural'FIRST;
  count     : in      Pcte.positive := Pcte.positive'LAST;
  status    : in      Pcte_error.handle := EXCEPTION_ONLY);

procedure copy (
  into_list  : in out  Pcte_object.link_set_descriptors.sequence;
  from_list  : in      Pcte_object.link_set_descriptors.sequence;
  into_index : in      Pcte.natural := Pcte.natural'LAST;
  from_index : in      Pcte.natural := Pcte.natural'FIRST;
  count      : in      Pcte.positive := Pcte.positive'LAST;
  status     : in      Pcte_error.handle := EXCEPTION_ONLY);

function length_of (
  list      : Pcte_object.link_set_descriptors.sequence;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
  return    Pcte.natural;

function index_of (
  list      : Pcte_object.link_set_descriptors.sequence;
  item      : Pcte_object.link_set_descriptor;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
  return    Pcte.integer;

function are_equal (
  first     : Pcte_object.link_set_descriptors.sequence;
  second    : Pcte_object.link_set_descriptors.sequence;
  status    : Pcte_error.handle := EXCEPTION_ONLY)
  return    Pcte.boolean;

procedure normalize (
  list      : in out  Pcte_object.link_set_descriptors.sequence;
  status    : in      Pcte_error.handle := EXCEPTION_ONLY);

procedure discard (
  list      : in out  Pcte_object.link_set_descriptors.sequence;
  status    : in      Pcte_error.handle := EXCEPTION_ONLY);

private
  implementation-defined

end link_set_descriptors;

type time_kind is (CURRENT_SYSTEM_TIME, SPECIFIED_TIME);

```

type time_selection (kind : Pcte_object.time_kind := CURRENT_SYSTEM_TIME)
is record

case kind **is**

when CURRENT_SYSTEM_TIME => **null**;

when SPECIFIED_TIME => time : Pcte.calendar.time;

end case;

end record;

-- 9.3.1 OBJECT_CHECK_TYPE

function check_type (

 object : Pcte.object_reference;

 type2 : Pcte.type_reference;

 status : Pcte_error.handle := EXCEPTION_ONLY)

return Pcte_object.type_ancestry;

-- 9.3.2 OBJECT_CONVERT

procedure convert (

 object : **in** Pcte.object_reference;

 pcte_type : **in** Pcte.type_reference;

 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

-- 9.3.3 OBJECT_COPY

procedure copy (

 object : **in** Pcte.object_reference;

 new_origin : **in** Pcte.object_reference;

 new_link : **in** Pcte.link_reference;

 access_mask : **in** Pcte_discretionary.object.atomic_access_rights;

 new_object : **in out** Pcte.object_reference;

 reverse_key : **in** Pcte.actual_key := "";

 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

procedure copy (

 object : **in** Pcte.object_reference;

 new_origin : **in** Pcte.object_reference;

 new_link : **in** Pcte.link_reference;

 on_same_volume_as : **in** Pcte.object_reference;

 access_mask : **in** Pcte_discretionary.object.atomic_access_rights;

 new_object : **in out** Pcte.object_reference;

 reverse_key : **in** Pcte.actual_key := "";

 status : **in** Pcte_error.handle := EXCEPTION_ONLY);

-- 9.3.4 OBJECT_CREATE

```

procedure create (
    pcte_type      : in      Pcte.type_reference;
    new_origin     : in      Pcte.object_reference;
    new_link       : in      Pcte.link_reference;
    access_mask    : in      Pcte_discretionary.object.atomic_access_rights;
    new_object     : in out   Pcte.object_reference;
    reverse_key    : in      Pcte.actual_key := "";
    status         : in      Pcte_error.handle := EXCEPTION_ONLY);

procedure create (
    pcte_type      : in      Pcte.type_reference;
    new_origin     : in      Pcte.object_reference;
    new_link       : in      Pcte.link_reference;
    on_same_volume_as : in   Pcte.object_reference;
    access_mask     : in     Pcte_discretionary.object.atomic_access_rights;
    new_object     : in out   Pcte.object_reference;
    reverse_key     : in     Pcte.actual_key := "";
    status         : in     Pcte_error.handle := EXCEPTION_ONLY);

```

-- 9.3.5 OBJECT_DELETE

```

procedure delete (
    origin : in   Pcte.object_reference;
    link   : in   Pcte.link_reference;
    status : in   Pcte_error.handle := EXCEPTION_ONLY);

```

-- 9.3.6 OBJECT_DELETE_ATTRIBUTE

```

procedure delete_attribute (
    object   : in   Pcte.object_reference;
    attribute : in   Pcte.attribute_reference;
    status   : in   Pcte_error.handle := EXCEPTION_ONLY);

```

-- 9.3.7 OBJECT_GET_ATTRIBUTE

-- OBJECT_GET_ATTRIBUTE is mapped to overloaded versions of the function
 -- Pcte_object.get_attribute, one for each possible type of the result. If the type of the
 -- result of the particular overloaded function that is used is different from the type of its
 -- parameter attribute, then the error VALUE_TYPE_IS_INVALID is raised.

```

function get_attribute (
    object   : Pcte.object_reference;
    attribute : Pcte.attribute_reference;
    status   : Pcte_error.handle := EXCEPTION_ONLY)
return     Pcte.boolean;

```