Standard ECMA-149
4th Edition - December 1997

ECMA

Standardizing Information and Communication Systems

Portable Common Tool Environment (PCTE) - Abstract Specification

VOLUME 1

Standard ECMA-149

4th Edition - December 1997

ECMA

Standardizing Information and Communication Systems

Portable Common Tool Environment (PCTE) - Abstract Specification

VOLUME 1

Brief History

(1) PCTE, Portable Common Tool Environment, is an interface standard. The interface is designed to support program portability by providing machine-independent access to a set of facilities. These facilities, which are described in this standard, are designed particularly to provide an infrastructure for programs which may be part of environments supporting systems engineering projects. Such programs, which are used as aids to systems development, are often referred to as tools.

(2) PCTE has its origin in the European Strategic Programme for Research and Development in Information Technology (ESPRIT) project 32, called "A Basis for a Portable Common Tool Environment". That project produced a specification for a tool interface, an initial implementation, and some tools on that implementation. The interface specifications were produced in the C Language. A number of versions of the specifications were produced, culminating in the fourth edition known as "PCTE Version 1.4". That was in two volumes; volume 2 covered the user interface and volume 1 covered everything else. Subsequently, the Commission of the European Communities (CEC) commissioned Ada versions of the two volumes of the PCTE specification.

(3) The CEC established the PCTE Interface Management Board (PIMB) in 1986 to maintain PCTE and promote its use. Through its subsidiary PCTE Interface Control Group (PICG) PIMB conducted a widespread public review, and published a revision known as PCTE 1.5.

(4) PIMB established an ad hoc task group to consider the form of the standard; this group reported in June 1988, strongly recommending that the standard should comprise an abstract (language-independent) specification and separate dependent bindings to whatever languages were chosen.

(5) In 1986 several nations of the Independent European Programme Group, under Technical Area 13 (IEPG TA-13), embarked on a collaborative programme to enhance PCTE to make it equally suitable for military as for civil use. This project was called PCTE+; the result of the definition phase was an enhanced specification called PCTE+ issue 3, published in October 1988. This consisted of both Ada and C versions of volume 1, volume 2 being the same as PCTE 1.5 volume 2. PCTE+ issue 3 was the basis for the assessment phase, which ended in December 1992. The ECMA PCTE standardization process has benefited greatly from close liaison with the PCTE+ programme; in particular through the availability of PCTE+ documents.

(6) Upon request from the PIMB, ECMA undertook to continue the development of PCTE to bring it into a form suitable for publication as an ECMA Standard. ECMA/TC33 was formed in February 1988 with this objective. Initially it was intended to base ECMA PCTE on PCTE 1.4, but this was soon changed to PCTE+ issue 3. The report of the PIMB task group on the form of the standard was accepted by TC33, and a task group (Task Group for ECMA PCTE, TGEP) was formed in November 1988, charged with producing the Abstract Specification and bindings for Ada and C.

(7) In 1989 attempts were made to standardize the user interface of tools on the basis of PCTE 1.4, volume 2. However it soon became apparent that it would be better for PCTE tools to use emerging general-purpose user interface standards, and the issue of a specific PCTE user interface was considered out of scope.

(8) Following acceptance of the first edition as an ECMA Standard in December 1990 (and of the bindings in 1991), review by international experts led to the production of second editions of all three standards. The second editions were accepted by the General Assembly of June 1993, and were submitted as a draft standard (in 3 parts) to ISO/IEC JTC1 for fast-track processing to international standardization.

(9) During the fast-track processing, which was successfully completed in September 1994, comments from National Bodies resulted in a number of changes to the draft standard. Some further editorial changes were requested by JTC1 ITTF. All these were incorporated in the published international standard, ISO/IEC 13719, with which the third editions of the ECMA standards were aligned.

(10) This fourth edition incorporates the resolutions of all comments received too late for consideration during the fast-track processing, or after, and the contents of Standards ECMA-227 (Extensions for Support of Fine-Grain Objects) and ECMA-255 (Object Orientation Extensions). It is aligned with the second edition of ISO/IEC 13719-1.

Adopted as 4th Edition of Standard ECMA-149 by the General Assembly of December 1997.

Contents

1 Scope **1**

2 Conformance **1**

2.1 Conformance of binding 1

2.2 Conformance of implementation 1

2.3 Conformance of DDL texts and processors 3

3 Normative references **3**

4 Definitions **4**

4.1 Technical terms 4

4.2 Other terms 4

5 Formal notations **5**

6 Overview of PCTE **5**

6.1 PCTE structural architecture 5

6.2 Object management system 5

6.3 Object base 6

6.4 Schema management 6

6.5 Self-representation and predefined SDSs 7

6.6 Object contents 7

6.7 Process execution 7

6.8 Monitoring 7

6.9 Communication between processes 8

6.10 Notification 8

6.11 Concurrency and integrity control 8

6.12 Distribution 9

6.13 Replication 9

6.14 Security 9

6.15 Accounting 10

6.16 Implementation limits 10

6.17 Support of fine-grain objects 10

6.18 Support of object-orientation 11

7 Outline of the Standard 11

8 Foundation 12

8.1 The state 12

8.2 The object base 13
8.2.1 Objects 13
8.2.2 Attributes 14
8.2.3 Links 15

8.3 Types 16
8.3.1 Object types 16
8.3.2 Attribute types 17
8.3.3 Link types 18
8.3.4 Enumeral types 22

8.4 Types in SDS 22
8.4.1 Object types in SDS 24
8.4.2 Attribute types in SDS 24
8.4.3 Link types in SDS 24
8.4.4 Enumeral types in SDS 25

8.5 Types in working schema 25
8.5.1 Object types in working schema 26
8.5.2 Attribute types in working schema 26
8.5.3 Link types in working schema 27
8.5.4 Enumeral types in working schema 27

8.6 Types in global schema 27

8.7 Operations 28
8.7.1 Calling process 28
8.7.2 Direct and indirect effects 28
8.7.3 Errors 30
8.7.4 Operation serializability 31

9 Object management 32

9.1 Object management concepts 32
9.1.1 The basic type "object" 32
9.1.2 The common root 36
9.1.3 Datatypes for object management 36

9.2 Link operations 36

9.3 Object operations 45

9.4 Version operations 59

10 Schema management **66**

10.1 Schema management concepts 66
10.1.1 Schema definition sets and the SDS directory 66
10.1.2 Types 67
10.1.3 Object types 69
10.1.4 Attribute types 69
10.1.5 Link types 71
10.1.6 Enumeral types 72
10.1.7 Datatypes for schema management 72

10.2 SDS update operations 73

10.3 SDS usage operations 101

10.4 Working schema operations 108

11 Volumes, devices, and archives **113**

11.1 Volume, device, and archiving concepts 113
11.1.1 Volumes 113
11.1.2 Administration volumes 114
11.1.3 Devices 114
11.1.4 Archives 115

11.2 Volume, device, and archive operations 116

12 Files, pipes, and devices **124**

12.1 File, pipe, and device concepts 124

12.2 File, pipe, and device operations 127

13 Process execution **135**

13.1 Process execution concepts 135
13.1.1 Static contexts 135
13.1.2 Foreign execution images 136
13.1.3 Execution classes 136
13.1.4 Processes 137
13.1.5 Initial processes 144
13.1.6 Profiling and monitoring concepts 144

13.2 Process execution operations 145

13.3 Security operations 159

13.4 Profiling operations 164

13.5 Monitoring operations 165

14 Message queues **167**

14.1 Message queue concepts 167

14.2 Message queue operations 170

15 Notification 176

15.1 Notification concepts 176
15.1.1 Access events and notifiers 176
15.1.2 Notification messages 177
15.1.3 Time of sending notification messages 178
15.1.4 Range of concerned message queues 178

15.2 Notification operations 178

16 Concurrency and integrity control 180

16.1 Concurrency and integrity control concepts 180
16.1.1 Activities 180
16.1.2 Resources and locks 182
16.1.3 Lock modes 185
16.1.4 Inheritance of locks 187
16.1.5 Establishment and promotion of locks 187
16.1.6 Implied locks 189
16.1.7 Conditions for establishment or promotion of a lock 189
16.1.8 Releasing locks 190
16.1.9 Permanence of updates 191
16.1.10 Tables for locks 192

16.2 Concurrency and integrity control operations 194

17 Replication 200

17.1 Replication concepts 200
17.1.1 Replica sets 200
17.1.2 Replicated objects 201
17.1.3 Selection of an appropriate replica 202
17.1.4 Administration replica set 203

17.2 Replication operations 203

18 Network connection 210

18.1 Network connection concepts 210
18.1.1 Execution sites 210
18.1.2 Workstations 210
18.1.3 Foreign systems 213
18.1.4 Network partitions 214
18.1.5 Accessibility 214
18.1.6 Workstation closedown 216

18.2 Network connection operations 217

18.3 Foreign system operations 222

18.4 Time operations 223

19 Discretionary security 225

19.1 Discretionary security concepts 225
19.1.1 Security groups 225
19.1.2 Access control lists 229
19.1.3 Discretionary access modes 231
19.1.4 Access control lists on object creation 234

19.2 Operations for discretionary access control operation 234

19.3 Discretionary security administration operations 238

20 Mandatory security 243

20.1 Mandatory security concepts 243
20.1.1 Mandatory classes 243
20.1.2 The mandatory class structure 245
20.1.3 Labels and the concept of dominance 246
20.1.4 Mandatory rules for information flow 247
20.1.5 Multi-level security labels 251
20.1.6 Floating security levels 254
20.1.7 Implementation restrictions 256
20.1.8 Built-in policy aspects 256

20.2 Operations for mandatory security operation 258

20.3 Mandatory security administration operations 263

20.4 Mandatory security operations for processes 268

21 Auditing 270

21.1 Auditing concepts 270
21.1.1 Audit files 270
21.1.2 Audit selection criteria 272

21.2 Auditing operations 273

22 Accounting 277

22.1 Accounting concepts 277
22.1.1 Consumers and accountable resources 277
22.1.2 Accounting logs and accounting records 279

22.2 Accounting administration operations 282

22.3 Consumer identity operations 287

23 Common binding features 288

23.1 Mapping of types 288

23.1.1 Mapping of predefined PCTE datatypes 288
23.1.2 Mapping of designators and nominators 290
23.1.3 Mapping of other values 298

23.2 Object reference operations 299

23.3 Link reference operations 301

23.4 Type reference operations 305

24 Implementation limits 307

24.1 Bounds on installation-wide limits 307

24.2 Bounds on workstation-dependent limits 309

24.3 Limit operations 309
24.3.1 Datatypes for limit operations 309

Annex A - VDM Specification Language for the Abstract Specification 311

Annex B - The Data Definition Language (DDL) 317

Annex C - Specification of Errors 327

Annex D - Auditable Events 349

Annex E - The Predefined Schema Definition Sets 357

Annex F - The fine-grain objects module 375

Annex G - The object-orientation module 389

Index of Operations 413

Index of Error Conditions 419

Index of Technical Terms 427

1 Scope

(1) This ECMA Standard specifies PCTE in abstract, programming-language-independent, terms. It specifies the interface supported by any conforming implementation as a set of abstract operation specifications, together with the types of their parameters and results. It is supported by a number of standard *bindings*, i.e. representations of the interface in standard programming languages.

(2) The scope of this ECMA Standard is restricted to a single PCTE installation. It does not specify the means of communication between PCTE installations, nor between a PCTE installation and another system.

(3) A number of features are not completely defined in this ECMA Standard, some freedom being allowed to the implementor. Some of these are *implementation limits*, for which constraints are defined (see clause 24). The other implementation-dependent and implementation-defined features are specified in the appropriate places in this Standard.

(4) 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 systems development, are often referred to as tools.

(5) This ECMA Standard also includes (in annex B) a language standard for the PCTE Data Description Language (DDL), suitable for writing PCTE schema definition sets.

2 Conformance

2.1 Conformance of binding

(1) A binding conforms to this ECMA Standard if and only if:

(2) - it consists of a set of operational interfaces and datatypes, with a mapping from the operations and datatypes of this ECMA Standard;

(3) - each operation of this ECMA Standard is mapped to one or more sequences of one or more operations of the binding (distinct operations need not be mapped to distinct sets of sequences of binding operations);

(4) - each datatype of this ECMA Standard is mapped to one or more datatypes of the binding;

(5) - each named error of this ECMA Standard is mapped to one or more error values (status values, exceptions, or the like) of the binding;

(6) - the conditions of clause 23 on common binding features are satisfied;

(7) - the conditions for conformance of an implementation to the binding are defined, are achievable, and are not in conflict with the conditions in 2.2 below.

2.2 Conformance of implementation

(1) The functionality of PCTE is divided into the following modules:

(2) - The core module consists of the datatypes and operations defined in clauses 8 to 19 (except 13.1.6, 13.4, and 13.5) and 23.

(3) - The mandatory access control module consists of the datatypes and operations defined in clause 20.

(4) - The auditing module consists of the datatypes and operations defined in clause 21.

(5) - The accounting module consists of the datatypes and operations defined in clause 22.

(6) - The profiling module consists of the datatypes defined in 13.1.6 and the operations defined in 13.4.

(7) - The monitoring module consists of the datatype Address defined in 13.1.6 and operations defined in 13.5.

(8) - The fine-grain objects module consists of the following extensions defined in annex F:

(9) . extensions to the semantics of operations to cater for fine-grain objects;

(10) . new operations;

(11) . new error conditions;

(12) . additions to the predefined SDS system.

(13) - The object-orientation module consists of the following extensions defined in annex G:

(14) . additions to the predefined SDSs metasds and system;

(15) . an extension to the semantics of the operation SDS_REMOVE_TYPE to cater for the new classes of type;

(16) . new operations;

(17) . new error conditions.

(18) An implementation of PCTE conforms to this ECMA Standard if and only if it implements the core module.

(19) An implementation of PCTE conforms to this ECMA Standard with mandatory access control level 1 or 2 if it implements the core module and in addition:

(20) - for level 1: the mandatory access control module except the floating security levels features defined in 20.1.6;

(21) - for level 2: the mandatory access control module.

(22) An implementation of PCTE conforms to this ECMA Standard with auditing if and only if it implements the core module and in addition the auditing module.

(23) An implementation of PCTE conforms to this ECMA Standard with accounting if and only if it implements the core module and in addition the accounting module.

(24) An implementation of PCTE conforms to this ECMA Standard with profiling if and only if it implements the core module and in addition the profiling module.

(25) An implementation of PCTE conforms to this ECMA Standard with monitoring if and only if it implements the core module and in addition the monitoring module.

(26) An implementation of PCTE conforms to this ECMA Standard with fine-grain objects if and only if it implements the core module and in addition, implements the fine-grain objects module.

(27) An implementation of PCTE conforms to this ECMA Standard with object-orientation if and only if it implements the core module and in addition the object-orientation module.

(28) By 'an implementation implements a module' is meant that, for the clauses of the module:

(29) - the implementation conforms to a binding of this ECMA Standard which itself conforms to this ECMA Standard and which is itself an ECMA Standard;

(30) - if an operation of this ECMA Standard is mapped to a set of sequences of operations in the binding:

  . case 1: operation_A; operation_B; ... operation_F;

  . case 2: operation_G; operation_H; ...operation_M;

  . etc.

  then in each case the sequence of invocations of the operations of the implementation must have the effect of the original operation of this ECMA Standard;

(31) - the relevant limits on quantities specified in clause 24 are no more restrictive than the values specified there;

(32) - the implementations of the implementation-defined features in this ECMA Standard are all defined.

(33) An implementation of PCTE does not conform to this ECMA Standard if it implements any of the following, whether or not the PCTE entity mentioned is in a module which the implementation implements:

(34) - an operation with same name as a PCTE operation but with different effect;

(35) - an SDS with the same name as a PCTE predefined SDS but with different contents;

(36) - an error condition with the same name as a PCTE error condition but with different meaning.

2.3 Conformance of DDL texts and processors

(1) A DDL definition conforms to this ECMA Standard if it conforms to the syntax and obeys the constraints of the DDL definition in annex B.

(2) A DDL processor conforms to this ECMA Standard if it accepts any conforming DDL definition and processes it in conformance with the meaning of DDL as defined in annex B.

3 Normative references

(1) The following standards contain provisions which, through reference in this text, constitute provisions of this ECMA Standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this ECMA Standard 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.

(2)	ISO/IEC 2022	Information Technology - Character code structure and extension techniques (1994)
(3)	ISO 8601	Data elements and interchange formats - Information interchange - Representation of dates and times (1988)
(4)	ISO 8859-1	Information processing — 8-bit single-byte coded graphic charactersets - Part 1 : Latin alphabet No. 1 (1987)

(5)	ISO/IEC 10646-1	Information Technology - Universal Multiple-Octet Coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane (1993)
(6)	ISO/IEC 11404	Information technology - Programming languages, their environments and system software interfaces - Language-independent datatypes (1996)
(7)	ISO/IEC 13303-1	Information technology - Programming languages, their environments and system software interfaces - Vienna Development Method/Specification language - Part 1 : Basic Language (1995)
(8)	ISO/IEC 14977	Information technology - Programming languages, their environments and system software interfaces - Extended BNF (1996)

4 Definitions

4.1 Technical terms

(1) All technical terms used in this ECMA Standard, other than a few in widespread use, are defined in the text, usually in a formal notation. All identifiers defined in VDM-SL or in DDL (see clause 5) are technical terms; apart from those, a defined technical term is printed in italics at the point of its definition, and only there. For the use of technical terms defined in VDM-SL and DDL see clause A.3 and clause B.9 respectively. All defined technical terms are listed in an index, with references to their definitions.

4.2 Other terms

(1) For the purposes of this ECMA Standard, the following definitions apply.

4.2.1 implementation-defined: Possibly differing between PCTE implementations, but defined for any particular PCTE implementation.

4.2.2 implementation-dependent: Possibly differing between PCTE implementations and not necessarily defined for any particular PCTE implementation.

4.2.3 binding-defined: Possibly differing between language bindings, but defined for any particular language binding.

4.2.4 datatype: The type of a parameter or result of an operation defined in this ECMA Standard, or used to define such a type. Where, as in clause 23, it is necessary to distinguish these types from datatypes defined elsewhere, the term *PCTE datatype* is used.

4.2.5 operation: a name plus a signature that is used in the context of an invocation to trigger the execution of a specific method.

4.2.6 interface: a set of operations; interfaces are a convenient way to group operations so that they can be referred to together, e.g. to define other interfaces by inheritance.

4.2.7 method: the set of actions triggered by an operation.

5 Formal notations

(1) Four formal notations are used in this ECMA Standard.

(2) For datatypes and for operation signatures, a small subset of the *Vienna Development Method Specification Language* or *VDM-SL* is used; it is defined in annex A. This subset of VDM-SL is also used to define some types used for operation parameters and results.

(3) The *Data Definition Language* or *DDL* is used to define types; it is defined in annex B. Where a concept is defined in both VDM-SL and DDL, the same identifier is used.

(4) To define the error conditions detected by operations, a parameterized notation is used; it is defined in annex C.

(5) The BSI syntactic notation (BS 6154 : 1981) is used to define the syntax of VDM-SL and DDL, and in a few other places where the syntax of strings is defined.

6 Overview of PCTE

(1) PCTE is designed to support program portability by providing machine-independent access to a set of facilities. These facilities, which are described in this ECMA Standard, are designed particularly to provide an infrastructure for programs to support systems engineering projects.

(2) The PCTE architecture is described in two dimensions: the *structural architecture* and the *functional architecture*. The structural architecture is described in 6.1, and shows how a PCTE installation is built of a system of communicating workstations and how the software providing the PCTE interfaces is structured. The functional architecture is described in 6.2 onwards, and gives an outline of the functional components of PCTE and the facilities they provide.

6.1 PCTE structural architecture

(1) The preferred structural architecture for a PCTE installation is a set of workstations and associated resources communicating over a network, though other architectures are possible. There is no hierarchy or ordering of workstations within a PCTE installation. If a workstation is part of a PCTE installation then the PCTE installation appears to the workstation's user as a conceptually single machine, although each workstation can act as an autonomous unit. Such a user has access to the total resources of a PCTE installation, subject to the necessary access controls.

(2) The PCTE database (called the *object base*) is partitioned into volumes. Volumes are dynamically allocated to (*mounted on*) particular workstations, and, once mounted, are globally available in that PCTE installation.

(3) The program writer does not need to be aware of the distribution architecture, but the PCTE interfaces do provide all the facilities needed to configure a PCTE installation and control its distribution. The PCTE interfaces appear to the tool writer as available within a PCTE installation irrespective of the tool's physical location within a PCTE installation and independent of any particular network topology.

6.2 Object management system

(1) An aspect of PCTE that is of major importance to the process of constructing and integrating portable tools is the provision of the object base and a set of functions to manipulate the various

objects in the object base. The object base is the repository of the data used by the tools of a PCTE installation, and the *Object Management System* or *OMS* of PCTE provides the functions used to access the object base.

(2) In a general sense, the users and programs of the PCTE installation have the ability to manage entities that are known to, and can be designated in, a particular PCTE installation. These may be files in the traditional sense, or peripherals, interprocess message queues or pipes, or the description of processes themselves or of the static context of a process. Tools supporting user applications establish classes of objects defined by the user: these can represent information items such as project milestones, tasks, and change requests.

6.3 Object base

(1) The basic OMS model is derived from the Entity Relationship data model and defines *objects* and *links* as being the basic items of a PCTE object base.

(2) Objects are entities (in the Entity Relationship sense) which can be designated, and can optionally have:

(3) - *Contents*: a storage of data representing the traditional file concept;

(4) - *Attributes*: primitive values representing specific properties of an object which can be named individually;

(5) - *Links*: representations of associations between objects. Links may have attributes, which may be used to describe properties of the associations or as keys to distinguish between links of the same type from the same object.

(6) Designation of links is the basis for the designation of objects: the principal means for accessing objects in most OMS operations is to navigate the object base by traversing a sequence of links.

6.4 Schema management

(1) Entities used by the user and those used by the system that are represented by objects in the object base can be treated in a uniform manner, and facilities to control their structure, to store and to designate these objects, are provided by PCTE.

(2) The object base of each PCTE installation is governed by a typing mechanism. All entities in the object base are typed and the data must conform to the corresponding type rules. Type rules are defined for objects, for links, and for attributes.

(3) PCTE is designed to allow, but not to require, distributed and devolved management of the object base. To this end the definition of the typing rules which govern an object, a link, or an attribute in the object base may be split up among a number of *schema definition sets* (or *SDSs*). Some properties of an object, a link, or an attribute must be the same in every SDS which contributes to the definition of the typing rules for that object, link, or attribute: these are properties of the *type.* Other properties may differ for different SDSs: these are properties of the *type in SDS.*

(4) Each SDS provides a consistent and self-contained view of the data in the object base. A process, at any one time, views the data in the object base through a *working schema.* A working schema is obtained as a composition of SDSs in an ordered list. The effect of such a composition is to provide a union of all the types contained in the listed SDSs. A uniform naming algorithm, dependent on the ordering of the SDSs, is applied to all the contained types.

(5) The object base of a PCTE installation has a notional *global schema*, composed of all the SDSs. The global schema is not directly represented in the object base, and the concept is used mainly to state certain consistency constraints on the object base as a whole.

(6) Child types of object types can be defined with the effect of implicit inheritance of all properties of their parent types. Additionally, child types can have properties of their own.

6.5 Self-representation and predefined SDSs

(1) Many of the entities in a PCTE installation are represented by objects in the object base. The types of these objects are defined in *predefined SDSs*, which are available in any conforming implementation; for example processes are represented by objects of type "process" which is defined in the predefined SDS 'system'. This property of PCTE is called *self-representation*. In general, in this ECMA Standard, the name of an entity is used also to refer to the object that represents it.

(2) In some cases an object of a type representing some kind of entity requires initializing, or must be created by a particular operation, before it can be used in operations to represent an entity of that kind. Such an object which has been initialized or correctly created is referred to as a *known* entity of that kind (i.e. known to the PCTE installation); any other object of that type is referred to as an *unknown* entity. For example an object of type "process" created by PROCESS_CREATE is a known process, while one created by OBJECT_CREATE is an unknown process.

6.6 Object contents

(1) A set of operations is provided to access the contents of some types of objects (files, pipes, and devices). These operations provide conventional input-output facilities on files and pipes and control of input and output on devices. These contents are not interpreted by PCTE.

(2) Other types of objects (accounting logs and audit files) have contents with structure that is defined by PCTE and for access to which special operations are provided.

6.7 Process execution

(1) PCTE is an interface to support programs. When a program is *run*, this is either the *execution* of the program itself, or the execution of an interpreter which interprets the program. An execution of a program is a *process*. Processes are represented by objects in the object base, so the hierarchy of processes, the environment in which a process runs, the parameters it has been passed, and the various stages of the program execution can be controlled, manipulated and examined.

(2) These facilities can be used also to control processes running on *foreign systems*. A *foreign system* can be a foreign development system, a target system running a real-time operating system, or even a PCTE workstation in another PCTE installation.

6.8 Monitoring

(1) PCTE provides three sets of features to support debugging and monitoring of processes.

(2) - To measure the amount of time spent in selected parts of the code.

(3) - To observe, and modify, the execution of a child process.

(4) - To measure the processor usage of the calling process.

6.9 Communication between processes

(1) PCTE provides a number of different mechanisms for communicating between processes. The principal ones supplied are:

(2) - the objects, links and attributes in the database;

(3) - message queues;

(4) - pipes.

(5) Message queues and pipes are essentially special forms of object. Thus both pipes and message queues are special cases of the general use of the object base for *interprocess communication*.

(6) Pipes and message queues also provide communication between PCTE processes and foreign processes running on foreign systems (if the foreign systems allow it).

6.10 Notification

(1) In PCTE there is a mechanism that allows the designation of objects so that certain types of access result in a message being posted in a message queue which can be accessed by the process requesting the notification.

(2) The notification mechanism allows a process to specify events, corresponding to operations on objects, of which it wants to be notified.

6.11 Concurrency and integrity control

(1) The object base is subject to concurrent access by users, and is liable to underlying system failure.

(2) PCTE provides locking facilities to control the strength of object base concurrency and consistency, ranging from unprotected behaviour, through protected behaviour, to protected atomic and serializable transaction activities. PCTE ensures object base consistency and object base integrity for atomic and serializable transactions.

(3) Each user carrying out a transaction on the object base sees some grouping of operations as an atomic operation which transforms the object base from one consistent state to another. If transactions are run one at a time then each transaction sees the consistent state left by its predecessor. When transactions are run concurrently PCTE ensures that the effect on the object base is as though they were run serially. With a few exceptions, such as messages sent to or received from a message queue, the effect of a sequence of operations performed within a transaction is atomic: either all the operations are performed or none are performed.

(4) Another important aspect of activities arises in composition of programs. A single program carrying out an atomic transaction on the object base can be regarded as performing a single function. More powerful functions can be built up by an outer program invoking a set of other, inner, programs, each of which carries out its own specific function. PCTE provides *nested activities* to allow each inner activity to behave in an atomic way, and at the same time to allow the whole function to be atomic. Thus the outer program can start a transaction, which may be

either committed or aborted, and finally the whole outer transaction is committed or aborted. Each such inner program could itself invoke further nested programs, and so on.

6.12 Distribution

(1) PCTE is based on a community of workstations of possibly differing types connected together by a network. The community is normally seen by the user as a single environment, grouping together the facilities, services and resources of all the different workstations, though in some circumstances a PCTE installation may be temporarily divided into separated partitions, each of which supports useful work.

(2) Objects, including processes, are distributed throughout a PCTE installation. A user is able to disregard both the location of objects on volumes in the network and that of the workstation concerned in executing processes. Alternatively a user may choose to exercise control over the location of objects on volumes and the location of processes. On creation of an object a volume can be specified to indicate its location. Every process executes on a particular workstation and a user can specify which workstation by either static or dynamic means: the static context of a program has an execution class identifying the range of workstations upon which the static context may be executed; the workstation on which a process executes can be specified on invocation.

6.13 Replication

(1) As it is possible that one or more workstations of a PCTE installation become temporarily unavailable, certain installation-wide objects must still be accessible. Replication facilities are available whereby a copy of an object's contents, attributes and links are made to each workstation. Installation-wide objects are predefined as replicated and other objects can be added. This feature is intended for non-volatile, rarely varying, widely consulted objects.

6.14 Security

(1) A PCTE installation has to support many users and many projects. Different users are expected to have different roles within projects and to be authorized to access different objects. The user accesses objects using programs (themselves modelled as static contexts within the object base).

(2) The purpose of security is to prevent the unauthorized disclosure, amendment or deletion of information. Security facilities are provided to support the definition of the different authorizations of users and programs.

(3) Security in PCTE is provided by discretionary and mandatory access controls. Access controls as defined in the security clauses form one aspect of the correct operation of the installation with regard to the integrity of the information held and the correctness of its use. In this regard, the facilities described in the security clauses complement the data modelling facilities of the OMS and schema management, and the transaction and concurrency control facilities.

(4) Each OMS object is associated with *access control lists* which define which types of access to the object are permitted for designated users or programs. Access control lists are expressed in terms of *discretionary access rights* which are explicitly granted or denied to designated individual users, user groups or program groups. Access rights on a particular object are combined in order to determine a process's permission to perform each particular operation on the object.

(5) Mandatory access controls cover both *mandatory confidentiality* and *mandatory integrity*, with distinct controls. Mandatory access controls are additional to discretionary access controls.

(6) Mandatory confidentiality controls prevent the disclosure of information to unauthorized users. They prevent the flow of information to the unauthorized user directly, by controlling read access (*simple confidentiality*), and indirectly, by controlling the flow of information between objects (*confidentiality confinement*).

(7) Mandatory integrity controls prevent unauthorized sources from contributing to the information in an object. They prevent the flow of information from the unauthorized user directly, by controlling write access (*simple integrity*), and indirectly, by controlling the flow of information between objects (*integrity confinement*).

6.15 Accounting

(1) The accounting facilities of PCTE allow the automatic recording of the consumption of selected installation resources by users, groups of users, or groups of programs.

(2) Authorized users may designate selected objects like programs, files, pipes, message queues, devices, workstations, and SDSs as being accountable resources. Access to an accountable resource by a process implies the automatic logging of usage information into the associated accounting log on completion of the operation.

6.16 Implementation limits

(1) PCTE permits the user to examine the implementation-defined limits for the PCTE installation in which a program executes.

(2) Minimal values are defined for limits, so that a program respecting those values is portable to any PCTE installation.

6.17 Support of fine-grain objects

(1) The notion of support of fine-grain objects is mostly concerned with improved performance time for creating and accessing PCTE objects. Object granularity is not dependent on type. It is described in terms of the amount of processing that has to be done to access an object.

(2) To enhance performance, the concept of cluster is introduced. A *cluster* is an object that represents the set of fine-grain objects that share the same values for certain PCTE properties and with some specific restrictions:

(3) - Usage restrictions on concurrency allow them to be cached in the main memory of processes.

(4) - Time attributes of all fine-grain objects residing in a cluster are shared.

(5) - Notification is not applicable to fine-grain objects.

(6) - Security properties are also shared and only checked once at the level of the cluster.

(7) - Auditing has limitations which decreases the controls to be made on fine-grain objects.

(8) - Fine-grain objects are not accountable resources.

(9) - Fine-grain objects have the same replicated state as their cluster.

6.18 Support of object-orientation

(1) One of the prominent characteristics of PCTE is its ability to define any user data model and to use a self-referential approach to describe its metadata. The object-orientation facilities follow a similar approach and describe everything as an extension of the metabase or of the object base.

(2) The data model supporting the object-orientation facilities can be partitioned into three parts: the interface part, the module part, and the method mapping part.

(3) The interface part of this data model is described as an extension of the metasds SDS, while the other two parts are extensions of the system SDS. The reason is that the instances of the interface part are additional information contained in a user SDS, while the instances of the two other parts are user data stored in the object base, as executable programs or loadable modules.

7 Outline of the Standard

(1) Clause 6 gives an informal, non-normative explanation of the concepts of PCTE. Clause 7 gives an overview of the document and of the structure of the definition.

(2) The partly formal, normative definition of PCTE is in clauses 8 to 24 and annexes A to C. It is in two main parts. The first main part is the *foundation* (clause 8) which defines the concept Object and its parts, for example Attribute and Link, and the concepts of the associated typing mechanism, for example Type and Type in SDS. This uses a subset of VDM-SL; see annex A.

(3) The second main part of the definition is the interface definition (clauses 9-22). This defines the other concepts of PCTE, for example Process and Workstation, as specializations of the concept Object (clauses 11-22). This definition is in terms of the typing structure associated with these specializations, that is in terms of the typing concepts of the foundation. A language for the definition of types and types in SDS, called *Data Definition Language* or *DDL*, is defined in annex B.

(4) The concept Object is itself further specialized, i.e. details not necessary for the foundation are added, in clause 9. (The name *Object* is used in both the foundation and the interface definition because it is the same concept although only a few of its details are defined in the foundation.)

(5) Thus the foundation is a relatively simple general model that is specialized in later clauses to provide the PCTE interface definition.

(6) Instances of the PCTE concepts are called *entities* and they are referred to by the names of the underlying concepts, for example instances of Object are called objects. All the entities existing at a time are called the *state* of the PCTE installation. PCTE is defined in terms of the permissible values of the state and the permissible operations on the state. The foundation defines part of the state, namely that part concerned with entities of the foundation concepts; the interface definition defines the rest of the state and all the operations.

(7) The concepts of the typing mechanism cannot be treated as specializations of the concept Object because the definition of PCTE would then be circular. They can however be *represented* by specializations of Object so that tools can determine the current state of the typing mechanism using the operations provided for determining the current state of objects. Operations for manipulating the state of the typing mechanism also manipulate the representing objects automatically and equivalently. The representations and operations of the typing mechanism are defined in clause 10.

(8) The interface is defined by operations grouped according to function. For each group some concepts are defined first in DDL and possibly VDM-SL, as described above. There follow the

operation definitions; a VDM-SL definition of the signature, an informal English description of the normal action of the operation, and a list of the possible error conditions (using an abbreviated notation defined in annex C).

(9) Other ECMA Standards define application programming interfaces to PCTE in terms of specific programming languages by defining the mapping of datatypes, operations, and error conditions of the abstract specification to datatypes, operations, and error conditions respectively of the programming language (see 3.1). Such mapping specifications are called *bindings*. Clause 23 defines a number of features to which all bindings must conform.

(10) Clause 24 defines the limits on the sizes and numbers of various entities which a conforming PCTE implementation must respect. These are given as minima which an implementation must meet or exceed.

(11) Annexes A to C define various notations used in the Abstract Specification. Annex A defines the subset of VDM-SL used for type definitions and operation signatures; annex B defines DDL; and annex C defines the notation for operation error conditions.

(12) Annex D contains a list of auditable events classified by event type.

(13) Annex E is provided for information; it collects the DDL definitions of the types in the predefined schema definition sets.

(14) Annex F is normative and contains the definition of the fine-grain objects module.

(15) Annex G is normative and contains the definition of the object-orientation module.

(16) Clauses 8 to 24 contain commentary (headed NOTE or NOTES) which is not normative and is intended as a help to the reader in understanding the definition.

8 Foundation

8.1 The state

(1)
```
state PCTE_Installation of
    SYSTEM_TIME          : Time
    OBJECT_BASE          : map Object_designator to Object
    PROCESSES            : set of Process
    MESSAGE_QUEUES       : set of Message_queue
    CONTENTS_HANDLES     : map Contents_handle to Current_position
    CURRENT_POSITIONS    : map Current_position to Natural
    WORKSTATIONS         : set of Workstation
end
```

(2)
```
Name = Text
```

(3)
```
Name_sequence = seq of Name
```

(4)
```
Working_schema ::
    VISIBLE_TYPES   : set of Type_in_working_schema
    SDS_NAMES       : Name_sequence
```

(5)
```
Process ::
    PROCESS_OBJECT       : Object_designator
    WORKING_SCHEMA       : Working_schema
    OPEN_CONTENTS        : set of Open_contents
```

(6)
```
Message_queue ::
    QUEUE_OBJECT   : Object_designator
    MESSAGES       : seq of Message
```

(7)

```
Workstation ::
    WORKSTATION_OBJECT      : Object_designator
    AUDIT_CRITERIA          : set of Selection_criterion
```

(8) Instances of the PCTE concepts are called *entities*; they are referred to by the names of the underlying concepts. The state comprises the entities of a PCTE installation that endure from one operation call to another. The effect of an operation call is to modify the state, or to return values derived from the state (and any parameters), or both.

(9) The system time is the date and time of day at any instant, as given by some system clock. For the format of the time see 23.1.1.5. The *current time* for an operation is a value of the system time at some moment between the start and end of the operation.

(10) The object base is a set of objects identified by object designators (see 8.2.1).

(11) A working schema is associated with a process (see clause 13) and consists of a set of types in working schema, derived from a sequence of SDSs. The types in working schema in the working schema of the calling process are called *visible types*. For the creation of a working schema for a process see 13.2.12.

(12) The initial value of the state consists of the following objects:

(13) - at least one workstation, at least one device managed by that workstation, at least one volume mounted on that device, and at least one process running on that workstation (see 18.1.2, 11.1.3, 11.1.1, and 13.1.5);

(14) - the administration replica set, the common root, and the administrative objects (see 17.1.4 and 9.1.2);

(15) - at least one user (see 19.1.1);

(16) - at least the schema definition sets system, metasds, discretionary_security, mandatory_security (if implemented), and accounting (if implemented) (see 10.1);

(17) - the predefined user group ALL_USERS, and the predefined program groups PCTE_AUDIT, PCTE_REPLICATION, PCTE_EXECUTION, PCTE_SECURITY, PCTE_HISTORY, PCTE_CONFIGURATION, and PCTE_SCHEMA_UPDATE (see 19.1.1).

(18) NOTE - It is intended that the system time should be as near as possible the same throughout a PCTE installation.

8.2 The object base

8.2.1 Objects

(1)

```
Object ::
    OBJECT_TYPE             : Object_type_nominator
    ATTRIBUTES              : set of Attribute
    LINKS                   : set of Link
    DIRECT_COMPONENTS       : set of Object
    PREFERRED_LINK_TYPE     : [ Link_type_nominator ]
    PREFERRED_LINK_KEY      : [ Text ]
    CONTENTS                : [ Contents ]
```

(2) Object_designator :: Token

(3) Object_designators = **set of** Object_designator

(4) Contents = Structured_contents **|** Unstructured_contents

(5) Structured_contents = Accounting_log **|** Audit_file

(6) Unstructured_contents = File | Pipe | Device

(7) Object_scope = ATOMIC | COMPOSITE

(8) The object type constrains the properties of the object (see 8.3.1).

(9) No two attributes of an object have the same attribute type. There is a basic set of attributes which all objects have; it is defined in 9.1.1.

(10) The preferred link type and preferred link key, if present, are used as defaults in the identification of a link of the object (see 8.2.3). The preferred link key has the syntax of a key (see 23.1.2.7).

(11) Every direct component of an object is the destination of a composition link of the object, and vice versa.

(12) An *outer object* of an object A is an object of which A is a component.

(13) The *atomic object* associated with an object comprises the links, attributes, preferred link type, preferred link key, and contents of the object. The *atoms* of an object are the atomic objects associated with the object and all its components.

(14) A *component* of an object is a direct component of the object or of a component of the object. An object which is a component of each of two distinct objects, neither of which is a component of the other, is called a *shared component* of those two objects.

(15) An *internal link* of an object is a link of the object or of one of its components for which the destination is either a component of the object or the object itself. An *external link* of an object is a direct or indirect outgoing link of the object which is not an internal link of the object. An object is called the *origin* of each of its links.

(16) An object is specified by an object designator, or by a specialization of object designator defined as follows: if "X" is an object type (that is, it is a descendant of "system-object", see 9.1.1) then 'X_designator' (with capital initial) stands for 'Object_designator' with the condition that the value must designate an object of type "X" or a descendant of "X". For the mapping of object designators to the language bindings, see 23.1.2.2.

(17) An object scope is used to indicate whether the effect of an operation applies to an object (COMPOSITE) or to the atomic object of the object (ATOMIC).

NOTES

(18) 1 An object can be a component of itself. Similarly two objects can be components of each other; in that case there are two distinct objects with the same atoms.

(19) 2 General operations are provided for handling unstructured contents (see clause 12) as a sequence of octets, the meaning of which is not further defined in this ECMA Standard. Specific operations are provided for handling structured contents, which has a defined meaning in each case (see clauses 21 and 22).

(20) 3 When an object is created, so are all its attributes in the global schema. When a new attribute type is applied to the object's type in an SDS, effectively all objects of that type and its descendants gain a new attribute with its initial value. If the application of an attribute type to that object type is removed from all SDSs, the attribute remains on each object of that type until deleted by OBJECT_DELETE_ATTRIBUTE.

8.2.2 Attributes

(1)
```
Attribute ::
    ATTRIBUTE_TYPE      : Attribute_type_nominator
    ATTRIBUTE_VALUE     : Attribute_value
```

(2) Attribute_value = Integer | Natural | Boolean | Time | Float | String

(3) Attribute_designator :: Token

(4) Attribute_designators = **set of** Attribute_designator

(5) Attribute_selection = Attribute_type_nominators | VISIBLE_ATTRIBUTE_TYPES

(6) Attribute_assignments = **map** Attribute_designator **to** Attribute_value

(7) String = **seq of** Octet

(8) The value of an enumeration attribute is represented by its position within the enumeration value type (see 8.3.2).

(9) An attribute is specified as follows:

(10) - for an attribute of an object: an object designator which specifies the object, and an attribute designator which specifies the attribute relative to the object;

(11) - for an attribute of a link: an object designator which specifies the origin of the link, a link designator which specifies the link relative to the object (see 8.2.3), and an attribute designator which specifies the attribute relative to the link.

NOTES

(12) 1 Each attribute in the object base is a key or non-key attribute of a link in the object base or a direct attribute of an object in the object base.

(13) 2 An implementation may impose constraints on the values of attributes (see clause 24). An attribute may take any value of its value type within those constraints; for example, a string attribute may take any string value up to the maximum allowed length, whatever its present value may be.

(14) 3 For the types Integer, Natural, Boolean, Time, Float, and String see 23.1.1.

8.2.3 Links

(1)
```
Link ::
    LINK_TYPE              : Link_type_nominator
    DESTINATION            : [ Object_designator ]
    KEY_ATTRIBUTES         : seq of Attribute
    NON_KEY_ATTRIBUTES     : set of Attribute
    REVERSE                : [ Link_designator ]
```

(2) Link_designator :: Token

(3) Actual_key = **seq1 of** (Text | Natural)

(4) Link_designators = **set of** Link_designator

(5) Link_selection = Link_type_nominators | VISIBLE_LINK_TYPES | ALL_LINK_TYPES

(6) Link_descriptor = Object_designator * Link_designator

(7) Link_descriptors = **set of** Link_descriptor

(8) Link_set_descriptor = Object_designator * Link_designators

(9) Link_set_descriptors = **set of** Link_set_descriptor

(10) Link_scope = INTERNAL_LINKS | EXTERNAL_LINKS | ALL_LINKS

(11) The key attributes and the non-key attributes are together called the *attributes of the link*. No two attributes of a link have the same attribute type.

(12) Two distinct links of the same type from the same object must have different key attributes (i.e. the two sequences of key attribute values must be different).

(13) The reverse link of the reverse link of a link is that link.

(14) A link is said to be *from* its origin and *to* its destination.

(15) A *series of links* from object A to object B is a sequence of 1 or more links L1, L2, ..., Ln such that A is the origin of L1, B is the destination of Ln, and otherwise the destination of each link is the origin of the next in sequence.

(16) A link is specified by an object designator which specifies the origin of the link and a link designator which specifies the link relative to the object. For the mapping of link designators to the language bindings, see 23.1.2.4.

NOTES

(17) 1 Each link in the object base is a link of exactly one object in the object base; i.e. each link has exactly one origin.

(18) 2 When a link is created, so are all its attributes in the global schema. When a new attribute type is applied to the link's type in an SDS, effectively all links of that type gain a new attribute with its initial value. If the application of an attribute type to that link type is removed from all SDSs, the attribute remains on each link of that type until deleted by LINK_DELETE_ATTRIBUTE.

8.3 Types

(1) Type = Object_type | Attribute_type | Link_type | Enumeral_type

(2) Type_nominator = Object_type_nominator | Attribute_type_nominator | Link_type_nominator | Enumeral_type_nominator

(3) Object_type_nominator :: Token

(4) Attribute_type_nominator :: Token

(5) Link_type_nominator :: Token

(6) Enumeral_type_nominator :: Token

(7) Type_nominators = **set of** Type_nominator

(8) Object_type_nominators = **set of** Object_type_nominator

(9) Attribute_type_nominators = **set of** Attribute_type_nominator

(10) Link_type_nominators = **set of** Link_type_nominator

(11) Type_kind = OBJECT_TYPE | ATTRIBUTE_TYPE | LINK_TYPE | ENUMERAL_TYPE

(12) A type is a template defining common basic properties of a set of instances. The *instances* of a type are those whose type nominator identifies that type.

(13) A type is specified by a type nominator, which may be specialized to an object type nominator, an attribute type nominator, a link type nominator, or an enumeral type nominator. A type nominator may be further specialized as follows: if "X" is an object type, attribute type, link type, or enumeral type then 'X_type_nominator' stands for 'Object_type_nominator' etc. with the condition that the value must designate type "X" or a descendant of "X". For the mapping of type nominators to language bindings see 23.1.2.5 and 23.1.2.

8.3.1 Object types

(1)
```
Object_type ::
    TYPE_NOMINATOR    : Object_type_nominator
    CONTENTS_TYPE     : [ Contents_type ]
    PARENT_TYPES      : Object_type_nominators
    CHILD_TYPES       : Object_type_nominators
    represented by object_type
```

(2)
```
Contents_type = FILE_TYPE | PIPE_TYPE | DEVICE_TYPE | AUDIT_FILE_TYPE |
    ACCOUNTING_LOG_TYPE
```

(3) The contents type, if present, specifies the type of contents of instances of the object type. If no contents type is supplied, instances of the object type have no contents.

(4) The parent types define inheritance rules governing the properties of object types in working schema (see 8.5.1). The parent types of an object type, their parent types, and so on, excluding the object type itself, are called the *ancestor types* of the object type.

(5) The child types are the object types which have this object type as parent type. The child types of an object type, their child types, and so on, excluding the object type itself, are called the *descendant types* of the object type.

(6) The parent/child relation between object types forms a directed acyclic graph, with the object type "object" (see 9.1.1) as the root.

8.3.2 Attribute types

(1)
```
Attribute_type ::
    TYPE_NOMINATOR          : Attribute_type_nominator
    VALUE_TYPE_IDENTIFIER   : Value_type_identifier
    INITIAL_VALUE           : [ Attribute_value ]
    DUPLICATION             : Duplication
    represented by attribute_type
```

(2)
```
Value_type_identifier = INTEGER | NATURAL | BOOLEAN | TIME | FLOAT | STRING |
    Enumeration_value_type_identifier
```

(3)
```
Enumeration_value_type_identifier = seq1 of Enumeral_type_nominator
```

(4)
```
Duplication = DUPLICATED | NON_DUPLICATED
```

(5) The value type identifier identifies the *value type* of the instances of the attribute type, i.e. the datatype of their possible attribute values (see table 1). See 23.1.1 for the mapping of values of integers, naturals, Booleans, times, floats, and strings. An enumeration value type identifier is a non-empty sequence of enumeral types.

(6) The initial value, which is a value of the value type, is the initial value of any attribute of this attribute type after creation and before any value has been assigned to it. If no initial value is supplied, the default initial value for the value type is used (see table 1).

(7) If the duplication is DUPLICATED, then every instance of the attribute type is a *duplicable* attribute, i.e. the value of the attribute is copied whenever an object or link with the attribute is copied; if it is NON_DUPLICATED then every instance is a *nonduplicable* attribute, i.e. the value of the copy of the attribute reverts to the initial value.

Table 1 - Value types

Value type identifier	**Value type**	**Default initial value**
INTEGER	Integer	0
NATURAL	Natural	0
BOOLEAN	Boolean	**false**
TIME	Time	1980-01-01T00:00:00Z
FLOAT	Float	0.0
STRING	String	"" (empty string)
Enumeration value type identifier	Enumeral type	1st enumeral type of the enumeration value type identifier

8.3.3 Link types

```
(1)   Link_type ::
          TYPE_NOMINATOR                  : Link_type_nominator
          CATEGORY                        : Category
          LOWER_BOUND, UPPER_BOUND        : [ Natural ]
          EXCLUSIVENESS                   : Exclusiveness
          STABILITY                       : Stability
          DUPLICATION                     : Duplication
          KEY_ATTRIBUTE_TYPES             : Key_types
          REVERSE_LINK_TYPE               : [ Link_type_nominator ]
          represented by link_type
```

(2) Key_types = **seq of** Attribute_type_nominators

(3) Category = COMPOSITION | EXISTENCE | REFERENCE | DESIGNATION | IMPLICIT

(4) Categories = **set of** Category

(5) Exclusiveness = SHARABLE | EXCLUSIVE

(6) Stability = ATOMIC_STABLE | COMPOSITE_STABLE | NON_STABLE

(7) All instances of a link type have the category, exclusiveness, stability, and duplication of the link type.

(8) The lower bound of a link type defines the number below which the number of links of that link type from any instance of an object type with that link type cannot be reduced. If absent, the lower bound is taken as 0. The lower bound is only checked when an attempt is made to delete a link, so that on creation of an object the number of links of a type may be less than the lower bound for that type.

(9) The upper bound of a link type is an optional natural defining the maximal number of links of that link type from any instance of an object type with that link type. If present, it must be greater than 0 and not less than the lower bound. If absent, there is no upper bound.

(10) A link type is said to be *of cardinality one* if its upper bound is 1. A link type of cardinality one has an empty sequence of key attribute types.

(11) A link type is said to be *of cardinality many* if it is not of cardinality one. A link type of cardinality many has a non-empty sequence of key attribute types.

(12) The sequence of key attribute types defines the attribute types of the sequence of key attributes of an instance of the link type. It does not contain any repeated attribute type nominators. A key attribute has value type Natural or String.

(13) The optional reverse link type is the link type which reverses the link type, i.e. whenever a link of this link type exists from object A to object B, a link of the reverse type exists from object B to object A, and vice versa. The reverse link type is not allowed if the category is DESIGNATION, and must be present otherwise.

(14) The term *complementary* is used of pairs of links, each having the other's origin as its destination, which are not reverses of each other.

(15) All link types of category IMPLICIT and cardinality many have lower bound 0, no upper bound, and a single key attribute of the predefined attribute type "system_key". The values of "system_key" attributes are implementation-dependent: each such key value is different from the value of every other "system_key" attribute of a link of the same link type from the same object.

(16) All link types of category EXISTENCE and cardinality many have lower bound 0.

(17) The category identifies certain *properties* of instances of the link type, as follows:

(18) - *relevance to the origin*. For a link with this property:

(19) . The link may be created and deleted explicitly.

(20) . APPEND_LINKS discretionary access right to the origin is required in order to create the link, and WRITE_LINKS discretionary access right to the origin is required in order to delete the link.

(21) . The link cannot be created or deleted if its origin is a stable object.

(22) . The creation and deletion of the link assign the current system time to the last modification time of the origin.

(23) . The link may have non-key attributes.

(24) For a link without the relevance to the origin property:

(25) . The link may only be created and deleted implicitly, i.e. as the reverse of a link with the relevance to the origin property.

(26) . APPEND_IMPLICIT discretionary access right to the origin is required in order to create the link, and WRITE_IMPLICIT discretionary access right to the origin is required in order to delete the link.

(27) . The link can be created or deleted even if its origin is a stable object.

(28) . The creation and deletion of the link have no effect on the last modification time of the origin.

(29) . The link may not have non-key attributes.

(30) - *referential integrity*. For a link with this property:

(31) . If the link exists then so does its destination, i.e. the existence of the link prevents the deletion of its destination.

(32) . The link always has a reverse link with the referential integrity property.

(33) - *existence property*. For a link with this property:

(34) . An object can be created as destination of the link.

(35) . The deletion of the link can imply the deletion of its destination.

(36) - *composition property*. For a link with this property:

(37) . The destination of the link is a component of its origin.

(38) The categories are defined in terms of these properties as follows:

(39) - COMPOSITION: relevance to the origin, referential integrity, existence property, composition property. Links with this category are called *composition links*.

(40) - EXISTENCE: relevance to the origin, referential integrity, existence property. Links with this category are called *existence links*.

(41) - REFERENCE: relevance to the origin, referential integrity. Links with this category are called *reference links*.

(42) - IMPLICIT: referential integrity. Links with this category are called *implicit links*.

(43) - DESIGNATION: relevance to the origin. Links with this category are called *designation links*.

(44) If the stability of a link type is ATOMIC_STABLE, each instance of the link type is an *atomically stabilizing* link, i.e. the destination of the link (excluding its components other than itself) cannot be modified or deleted.

(45) If the stability of a link type is COMPOSITE_STABLE, each instance of the link type is a *compositely stabilizing* link, i.e. the destination of the link (including its components) cannot be modified or deleted.

(46) If the stability of a link type is NON_STABLE, each instance of the link type is a *nonstabilizing* link, i.e. the existence of the link does not prevent the modification or deletion of its destination or its components.

(47) Modification of an object is defined in 9.1.1. A *stable* object is the destination of an atomically or compositely stabilizing link, or a component of the destination of a compositely stabilizing link.

(48) Exclusiveness applies only to composition link types. If it is EXCLUSIVE, each instance of the link type is an *exclusive* composition link, i.e. no other composition link can share the same destination. If it is SHARABLE, each instance of the link type is a *sharable* composition link, i.e. other composition links can share the same destination.

(49) If duplication is DUPLICATED, each instance is a *duplicable* link, i.e. the link is copied whenever its origin is copied; if it is NON_DUPLICATED, each instance is a *nonduplicable* link, i.e. a copy of the object has no copy of the link. An implicit link cannot be duplicable.

(50) A component of an object is a *duplicable component* if it is the destination of at least one duplicable internal composition link whose origin is either the object or a duplicable component of the object.

(51) A link type of category IMPLICIT or DESIGNATION must be nonstabilizing.

(52) The following relations hold between properties of a link type and of its reverse link type:

(53) - if one link type has category IMPLICIT, then the other does not;

(54) - if one link type has the existence property (i.e. has category EXISTENCE or COMPOSITION) then the other does not;

(55) - if one link type has stability ATOMIC_STABLE or COMPOSITE_STABLE then the other has category IMPLICIT.

(56) A link type of category DESIGNATION cannot have a reverse link type.

(57) Links of the following types are termed *usage designation links*, because they are not checked by the normal security rules: "running_process", "in_working_schema_of" , "consumer_process", "user_identity_of", "adopted_user_group_of", "reserved_by", "locked_by", "lock", "opened_by", "mounted_on", and "listened_to". Usage designation links have the following properties:

(58) - creation or deletion implies only a bitwise write access on the origin object from a mandatory security point of view (see 20.1.8.2);

(59) - creation or deletion requires one unspecified discretionary access permission on the origin object;

(60) - creation or deletion is possible for an object on a read-only volume;

(61) - creation or deletion is possible for a copy object as origin;

(62) - creation or deletion does not require the establishment of locks on the links;

(63) - they are not copied by REPLICATED_OBJECT_DUPLICATE;

(64) - they can be implicitly deleted by network failure and workstation closedown;

(65) - creation or deletion does not change the last modification time of the origin object.

(66) Links of the following types are termed *service designation links*, because they indicate that the destination provides a service to the origin (usually a process): "executed_on", "sds_in_working_schema", "consumer_identity", "user_identity", "adopted_user_group", "reserved_message_queue", "open_object", "process_waiting_for", "referenced_object", "adoptable_user_group", "mounted_volume", "is_listener", "notifier", and "executed_static_context". Service designation links have the following properties:

(67) - creation or deletion does not require the establishment of locks on the links;

(68) - they are implicitly deleted by workstation failure;

(69) - for navigation along these links to replicated objects, replication redirection applies to the state of the object base at the time the link was created rather than when it is navigated through.

NOTES

(70) 1 The properties of links of various categories are summarized in table 2.

Table 2 - Properties of link categories

Property	Composition links	Existence links	Reference links	Implicit links	Designation links
relevance to origin	yes	yes	yes	no	yes
referential integrity	yes	yes	yes	yes	no
existence	yes	yes	no	no	no
composition	yes	no	no	no	no
atomic stability	optional	optional	optional	no	no
composite stability	optional	optional	optional	no	no
exclusiveness	optional	no	no	no	no
duplication	optional	optional	optional	no	optional
has a reverse link	yes	yes	yes	yes	no

(71) 2 The reason why the lower bound of an existence link is 0 is that if there existed an existence link type L with a lower bound of 2, for example, and an object X had two outgoing links of type L, it would be impossible to delete either link directly using LINK_DELETE. Indirect deletion of these links by deletion of object X would also be impossible because X would have outgoing existence links. This means that the destinations of these links could never be deleted. This would be an undesirable situation. The same problem does not exist with composition links because a composite object can be deleted in a single operation, OBJECT-DELETE.

8.3.4 Enumeral types

(1)
```
Enumeral_type ::
    TYPE_NOMINATOR    : Enumeral_type_nominator
    represented by enumeral_type
```

(2) An enumeral type is used as a possible value of an enumeration attribute. It has no instances.

8.4 Types in SDS

(1)
```
Type_in_sds  = Object_type_in_sds | Attribute_type_in_sds | Link_type_in_sds |
    Enumeral_type_in_sds
```

(2)
```
Type_in_sds_common_part ::
    ASSOCIATED_TYPE  : Type_nominator
    LOCAL_SDS        : Object_designator
    LOCAL_NAME       : [ Name ]
```

(3)
```
Type_nominator_in_sds = Object_type_nominator_in_sds | Attribute_type_nominator_in_sds |
    Link_type_nominator_in_sds | Enumeral_type_nominator_in_sds
```

(4) `Object_type_nominator_in_sds       :: Token`

(5) `Attribute_type_nominator_in_sds    :: Token`

(6) `Link_type_nominator_in_sds         :: Token`

(7) `Enumeral_type_nominator_in_sds     :: Token`

(8) Type_nominators_in_sds = **set of** Type_nominator_in_sds

(9) Object_type_nominators_in_sds = **set of** Object_type_nominator_in_sds

(10) Attribute_type_nominators_in_sds = **set of** Attribute_type_nominator_in_sds

(11) Link_type_nominators_in_sds = **set of** Link_type_nominator_in_sds

(12) Enumeral_type_nominators_in_sds = **set of** Enumeral_type_nominator_in_sds

(13) Definition_mode_value = CREATE_MODE | DELETE_MODE | READ_MODE | WRITE_MODE | NAVIGATE_MODE

(14) Definition_mode_values = **set of** Definition_mode_value

(15)
```
Definition_modes ::
    USAGE_MODE              : Definition_mode_values
    EXPORT_MODE             : Definition_mode_values
    MAXIMUM_USAGE_MODE      : Definition_mode_values
```

(16) A type in SDS (plural 'types in SDS') is a template defining a set of properties which apply to all instances of its type, in addition to the basic properties of that type. A type in SDS is *associated with* one type; a type is *associated with* one or more types in SDS.

(17) A *schema definition set* (or *SDS*) is an object of type "sds" (see 10.1.1), and is specified by an object designator. A type in SDS *belongs to*, or is *in*, a particular SDS, called its local SDS.

(18) The local name identifies the type in SDS, and hence the associated type, uniquely within the local SDS. The *complete name* of a type in SDS is the name of the SDS, followed by a hyphen '-', followed by the local name of the type in SDS.

(19) The definition modes specify restrictions on the usage of the type in SDS. The usage mode specifies the permitted kinds of access to instances of the type in SDS by a process which has adopted its local SDS in its working schema. The export mode specifies the maximum usage mode of the copy of the type in SDS which is created when the type in SDS is exported to another SDS; it is a subset of the usage mode. The maximum usage mode specifies which definition mode values can be included in the usage mode and export mode; it is set on creation of the type in SDS and cannot be changed. The definition modes of a link and of its reverse must be the same. Enumeral types in SDS do not have definition modes.

(20) The accesses controlled by definition modes are as follows.

(21) - READ_MODE controls reading from attributes by the operations OBJECT_GET_ATTRIBUTE, OBJECT_GET_SEVERAL_ATTRIBUTES, LINK_GET_ATTRIBUTE, and LINK_GET_SEVERAL_ATTRIBUTES.

(22) - WRITE_MODE controls writing to attributes by the operations OBJECT_SET_ATTRIBUTE, OBJECT_SET_SEVERAL_ATTRIBUTES, LINK_SET_ATTRIBUTE, LINK_SET_SEVERAL_ATTRIBUTES, OBJECT_RESET_ATTRIBUTE, and LINK_RESET_ATTRIBUTE.

(23) - CREATE_MODE controls creation of objects and links by the operations OBJECT_CREATE, OBJECT_COPY, OBJECT_CONVERT, VERSION_REVISE, VERSION_SNAPSHOT, DEVICE_CREATE, and PROCESS_CREATE; and creation of links by the operations LINK_CREATE and LINK_REPLACE.

(24) - DELETE_MODE controls deletion of objects and links by the operation OBJECT_DELETE and deletion of links by the operations LINK_DELETE and LINK_REPLACE.

(25) - NAVIGATE_MODE controls the use of link references in pathnames in the evaluation of object references (see 23.1.2.2).

(26) Types in SDS are specialized to object types in SDS, attribute types in SDS, link types in SDS, and enumeral types in SDS; the associated types are object types, attribute types, link types, and enumeral types respectively.

(27) A type in SDS is specified by a type nominator in SDS, which may be specialized to an object type nominator in SDS, an attribute type nominator in SDS, a link type nominator in SDS, or an enumeral type nominator in SDS. A type nominator in SDS may be further specialized as follows: if "X" is an object type, attribute type, link type, or enumeral type then 'X_type_nominator_in_sds' stands for 'Object_type_nominator_in_sds' etc. with the condition that the value must designate a type in SDS associated with type "X" or a descendant of "X". For the mapping of type nominators in SDS to language bindings see 23.1.2.5.

(28) NOTE - The properties of a type and of an associated type in SDS can be specified by means of the Data Definition Language (see annex B).

8.4.1 Object types in SDS

(1)
```
Object_type_in_sds :: Type_in_sds_common_part &&
    DIRECT_ATTRIBUTE_TYPES_IN_SDS        : Attribute_type_nominators_in_sds
    DIRECT_OUTGOING_LINK_TYPES_IN_SDS    : Link_type_nominators_in_sds
    DIRECT_COMPONENT_TYPES_IN_SDS        : Object_type_nominators_in_sds
    DEFINITION_MODES                     : Definition_modes
    represented by object_type_in_sds
```

(2) The only allowed definition mode value for an object type in SDS is CREATE_MODE.

(3) The direct attribute types in SDS must be in the same SDS as the object type in SDS. They participate in the definition of the visible attribute types of object types in working schema; see 8.5.1.

(4) The direct outgoing link types in SDS must be in the same SDS as the object type in SDS. They participate in the definition of the visible link types of object types in working schema; see 8.5.1. The object type in SDS is called the *origin object type in SDS* of each of the direct outgoing link types in SDS.

(5) The direct component types in SDS must be in the same SDS as the object type in SDS. They participate in the definition of the direct component types of object types in working schema; see 8.5.1.

8.4.2 Attribute types in SDS

(1)
```
Attribute_type_in_sds :: Type_in_sds_common_part &&
    DEFINITION_MODES : Definition_modes
    represented by attribute_type_in_sds
```

(2) The only allowed definition mode values for an attribute type in SDS are READ_MODE and WRITE_MODE.

8.4.3 Link types in SDS

(1)
```
Link_type_in_sds :: Type_in_sds_common_part &&
    DESTINATION_OBJECT_TYPES_IN_SDS        : Object_type_nominators_in_sds
    NON_KEY_ATTRIBUTE_TYPES_IN_SDS         : Attribute_type_nominators_in_sds
    DEFINITION_MODES                       : Definition_modes
    represented by link_type_in_sds
```

(2) The only allowed definition mode values for a link type in SDS are CREATE_MODE, DELETE_MODE, and NAVIGATE_MODE.

(3) The destination object types in SDS must be in the same SDS as the link type in SDS. They participate in the definition of the destination object types of link types in working schema; see 8.5.3.

(4) The non-key attribute types in SDS must be in the same SDS as the link type in SDS. They participate in the definition of the visible attribute types of link types in working schema; see 8.5.3.

8.4.4 Enumeral types in SDS

(1)
```
Enumeral_type_in_sds :: Type_in_sds_common_part &&
    IMAGE        : Text
    represented by enumeral_type_in_sds
```

(2) An enumeral type in SDS associates with the enumeral type a string called its image.

8.5 Types in working schema

(1)
```
Type_in_working_schema  = Object_type_in_working_schema |
    Attribute_type_in_working_schema | Link_type_in_working_schema |
    Enumeral_type_in_working_schema
```

(2)
```
Type_in_working_schema_common_part ::
    ASSOCIATED_TYPE                  : Type_nominator
    CONSTITUENT_TYPES_IN_SDS         : seq of (Composite_name * Type_nominator_in_sds)
```

(3)
```
Composite_name ::
    SDS_NAME          : Name
    LOCAL_NAME        : [ Name ]
```

(4) A type in working schema is a template defining common properties for a set of instances of its type. The properties of a type in working schema are derived from the properties of one or more types in SDS (see 8.5.1 to 8.5.4). Types in working schema occur in working schemas, see 8.1. For the construction of working schemas, see 13.2.12.

(5) The constituent types in SDS of a type in working schema must all have the same associated type, which is the type *associated with* the type in working schema.

(6) A type in working schema has several composite names, one for each constituent type in SDS. For each composite name, the SDS name is the name of the local SDS of the corresponding type in SDS, and the local name, if any, is the local name of the type in SDS in its local SDS.

(7) Let C1 and C2 be composite names, and T1 and T2 be type nominators in SDS. Then for any two constituent types in SDS (C1, T1), (C2, T2) of a type in working schema, if the SDS name of C1 precedes the SDS name of C2 in the SDS names of the working schema containing the type in working schema, then (C1, T1) precedes (C2, T2).

(8) Types in working schema are specialized to object types in working schema, attribute types in working schema, link types in working schema, and enumeral types in working schema; their associated types are object types, attribute types, link types, and enumeral types respectively.

(9) The value of a type in SDS cannot be changed while it is part of a type in working schema.

(10) A type in working schema is specified by a type nominator, see 8.3.

8.5.1 Object types in working schema

(1)

```
Object_type_in_working_schema :: Type_in_working_schema_common_part &&
    CHILD_TYPES                  : Object_type_nominators
    PARENT_TYPES                 : Object_type_nominators
    APPLIED_ATTRIBUTE_TYPES      : Attribute_type_nominators
    APPLIED_LINK_TYPES           : Link_type_nominators
    VISIBLE_ATTRIBUTE_TYPES      : Attribute_type_nominators
    VISIBLE_LINK_TYPES           : Link_type_nominators
    DIRECT_COMPONENT_TYPES       : Object_type_nominators
    USAGE_MODES                  : Definition_mode_values
```

(2) The set of constituent types in SDS of the child types is the union of the sets of child types of the constituent types in SDS of the type in working schema.

(3) The set of constituent types in SDS of the parent types is the union of the sets of parent types of constituent types in SDS of the type in working schema.

(4) The applied attribute types are the attribute types in working schema which have a constituent type in SDS of a direct attribute type in SDS of one of the constituent types in SDS of the object type in working schema.

(5) The applied link types are the link types in working schema which have a constituent type in SDS of a direct outgoing link type in SDS of one of the constituent types in SDS of the object type in working schema.

(6) The direct component types are the object types in working schema which have a constituent type in SDS of a direct component type in SDS of one of the constituent types in SDS of the object type in working schema.

(7) The set of visible attribute types is the union of the set of applied attribute types and the sets of the visible attribute types of all the parent types.

(8) The set of visible link types is the union of the set of applied link types and the sets of the visible link types of all the parent types.

(9) The constituent types in SDS must be object types in SDS.

(10) If the type of an object is not visible, the object is considered as an instance of any of its object type's ancestor types which is visible.

(11) The set of usage modes is the union of the sets of definition modes of all constituent types in SDS of the object type in working schema.

8.5.2 Attribute types in working schema

(1)

```
Attribute_type_in_working_schema :: Type_in_working_schema_common_part &&
    USAGE_MODES   : set of Definition_mode_values
```

(2) The constituent types in SDS must be attribute types in SDS.

(3) The set of usage modes is the union of the sets of definition modes of all constituent types in SDS of the attribute type in working schema.

8.5.3 Link types in working schema

(1)
```
Link_type_in_working_schema :: Type_in_working_schema_common_part &&
    DESTINATION_OBJECT_TYPES                : Object_type_nominators
    VISIBLE_DESTINATION_OBJECT_TYPES        : Object_type_nominators
    KEY_ATTRIBUTE_TYPES                     : Key_types
    APPLIED_ATTRIBUTE_TYPES                 : Attribute_type_nominators
    REVERSE                                 : [ Link_type_nominator ]
    USAGE_MODES                             : Definition_mode_values
```

(2) The set of constituent types in SDS of the applied attribute types is the union of the sets of non-key attribute types of the constituent types in SDS of the link type in working schema.

(3) The set of constituent types in SDS of the destination object types is the union of the sets of destination object types of the constituent types in SDS of the link type in working schema.

(4) The constituent types in SDS must be link types in SDS.

(5) The set of visible destination object types is the union of the set of destination object types and the set of visible descendants of the visible destination object types.

(6) The sequence of key attribute types is the same as the sequence of key attribute types of the associated type.

(7) The set of usage modes is the union of the sets of definition modes of all constituent types in SDS of the link type in working schema.

8.5.4 Enumeral types in working schema

(1)
```
Enumeral_type_in_working_schema   ::   Type_in_working_schema_common_part &&
    IMAGE : Text
```

(2) The image of an enumeral type in working schema T1 is the image of the first of its types in SDS which has an image, unless another enumeral type in working schema T2 belonging to the same enumeration attribute type in working schema as T1 but with a lower position already has the same image, in which case T1 has no image.

(3) The constituent types in SDS must be enumeral types in SDS.

8.6 Types in global schema

(1) The *global schema* is the working schema constituted by all the SDSs of a PCTE installation; the order is irrelevant as it affects only the type names, which are of no concern here. A *type in global schema* is a type in working schema in the global schema; it follows that each type is associated with one type in global schema. The global schema is a notional working schema used to state the following consistency rules applying to the whole object base; it is not necessarily the working schema of any process.

(2) An object must be compatible with its associated object type in global schema, i.e.:

(3) - The link types in global schema of the links of the object must be among the visible link types in global schema of the object type in global schema.

(4) - The attribute types in global schema of the attributes of the object must be among the visible attribute types in global schema of the object type in global schema.

(5) - The object types in global schema of the direct components of the object must be among the direct component object types in global schema of the object type in global schema.

(6) - The preferred link type of the object, if present, must be one of the applied link types of the object type in global schema.

(7) - The preferred link key of the object, if present, must have the same value types (String or Natural), in the same order, as the key attribute types of the preferred link type of the object.

(8) A link must be compatible with its associated link type in global schema, i.e.:

(9) - The object type in global schema of the destination of the link must be among the visible destination types of the link type in global schema.

(10) - The key attributes of the link, if any, must have the same value types (String or Natural) in the same order as the key attribute types of the link type in global schema.

(11) - The non-key attributes of the link must be among the applied attribute types of the link type in global schema.

(12) - The link type in global schema of the reverse link, if any, must be the reverse of the link type in global schema.

8.7 Operations

8.7.1 Calling process

(1) The operations defined in clauses 9 to 22 take effect when they are *executed* by a process (see 13.1.4). The process is known as the *calling process* of the operation. The effects on the state of the PCTE installation are global, i.e. can be observed by other processes. Results returned by operations which are designators are local to the calling process.

8.7.2 Direct and indirect effects

(1) The effects of an operation on the state of the PCTE installation comprise direct effects and indirect effects. *Direct effects* are described in the relevant operation descriptions (including the error descriptions). *Indirect effects* are described elsewhere in clauses 9 to 22. The operations of clause 23 do not affect the state.

(2) Indirect effects occur by means of *events*. Events are of several classes, described below. An operation may *raise* an event. Depending on the state of the PCTE installation, the raising of an event may result either in the effect of another operation being different to what it would otherwise be, or in some other change of state.

(3) An operation has a finite non-zero duration, and an event that is raised during the operation may have an effect on that operation.

(4) In general, the raising of an event is not explicitly described in the operation that raises the event, but instead in the part of the interface definition that may be affected by the event. It must nevertheless be understood that the description of an operation may need to be implicitly extended by the description of the raising of events. The processing of an event takes place asynchronously.

(5) There are several different classes of event, as described below.

(6) - *Access event*. This is described in clause 15. Access events are raised by operations which perform certain kinds of access to objects. If an appropriate notifier has been established, then the raising of the event causes an appropriate message to be sent.

(7) - *Lock release event*. A lock release event occurs when a lock is released (see 16.1.8). If some other operation is waiting to acquire a lock on the same resource, then that operation may proceed. If there is more than one such operation then which operation acquires the lock is implementation-dependent. There is no further description of this event in this ECMA Standard.

(8) - *Process timeout event*. This event is raised when the duration of an operation has exceeded the process timeout value for that process. When this event is raised, the error condition OPERATION_HAS_TIMED_OUT holds for that operation, and the operation terminates with that error. This event is described in 13.1.4.

(9) - *Process alarm event*. This event is raised when the time left until alarm has expired. When this event is raised, a message of message type WAKE is sent to the process and the process is resumed. This event and its effect are described in 13.1.4 and 13.2.6.

(10) - *Interrupt operation event*. This is described in 13.2.4. This event is raised by PROCESS_INTERRUPT_OPERATION or by PROCESS_TERMINATE. If the interrupted process is executing an operation or waiting for an event when this error is raised, then the error condition OPERATION_IS_INTERRUPTED holds for that operation and it terminates with that error.

(11) - *Audit event*. These events are described in clause 21. Audit events are raised by operations which carry out object access, and for exploiting audit records, copying audit records, carrying out certain security operations, and at explicit user request (see 21.2.5). If the event type is selected and auditing is enabled (or the event type is always audited) then an audit record is written as described in 21.1.1.

(12) - *Accounting event*. Accounting events are divided into start events and end events. These are raised as a result of certain operations, and if the resource is accountable and accounting is enabled then an accounting record is written (see 22.1.2).

(13) - *Message queue event*. These events are described in 14.1. They are raised by the appearance in a message queue of a message, which may be caused by MESSAGE_SEND_WAIT, MESSAGE_SEND_NO_WAIT, or QUEUE_RESTORE. If there is a handler for the event then that handler is executed.

(14) - *Resource availability event*. These events do not occur as a result of operations, but may occur at any moment. They model changes in the availability of hardware resources. The effect of a resource availability event on the directly affected objects is implementation-dependent. The set of directly affected objects for an event is implementation-defined. The effect on access from other objects is defined as follows for the various resource availability events.

(15) . *Volume failure*: an accessible volume becomes inaccessible. Attempted access to objects on the volume (including replicas in the case of an administration volume) fails with the error OBJECT_IS_INACCESSIBLE. Attempts to commit transactions which have started and which involve objects on the volume also fail.

(16) . *Device failure*: an accessible device becomes inaccessible. Attempted access to the file contents of the device fails. Volume failure is raised for any volume mounted on the device.

(17) . *Network failure*: an accessible workstation becomes inaccessible. There is no distinction between a workstation failing and a network failing so that communication with the workstation is lost. The inaccessible workstation ceases to be in its current network

partition. Associated devices and volumes become inaccessible with consequences as above.

(18) . *Network repair*: a workstation joins a network partition. This has no immediate effect, but the workstation becomes accessible when the other conditions are met.

(19) - *Process termination event*. This event is raised when a process is terminated by PROCESS_TERMINATE, explicitly or implicitly or by normal or abnormal process termination. If a PROCESS_WAIT_FOR_CHILD or PROCESS_WAIT_FOR_ANY_CHILD operation of the parent process is waiting for that process or any sibling process to terminate, then it may proceed. If more than one such operation exists (for different threads) then all may proceed.

(20) - *Data available event*. This event is raised when data is written to a device contents, pipe contents, or message queue. If an operation is waiting on a CONTENTS_READ on a pipe or device which is not non-blocking, and data is written to that pipe or device, then that operation may proceed. If an operation is waiting on a MESSAGE_RECEIVE_WAIT on a message queue, and a message is received of a type included in the set of types specified by that MESSAGE_RECEIVE_WAIT, then the operation may proceed. If more than one operation is waiting on that event, which operation receives the data and ceases to wait is implementation-dependent. AUDIT_FILE_READ and ACCOUNTING_LOG_READ do not wait for data to be available.

(21) - *Data space available event*. This event is raised when space becomes available in a device contents, pipe contents, or message queue. If an operation is waiting on a CONTENTS_WRITE on a pipe or device which is not non-blocking, and data is removed from that pipe or device, then that operation may proceed if sufficient space has been made available. If an operation is waiting on a MESSAGE_SEND_WAIT on a message queue and a message is removed from the queue then the operation may proceed if sufficient space has been made available. If an operation is waiting to write to the audit file or accounting log and the audit file or accounting log becomes available, then the operation proceeds. If more than one operation is waiting on that event for a contents or message queue, which operation writes or sends the data and ceases to wait is implementation-dependent.

(22) - *Security attribute change*. This event is raised by OBJECT_SET_CONFIDENTIALITY_LABEL, OBJECT_SET_INTEGRITY_LABEL, or OBJECT_SET_ACL_ENTRY, or changes to labels as a result of floating. If a CONTENTS_READ, CONTENTS_WRITE, MESSAGE_RECEIVE_WAIT or MESSAGE_SEND_WAIT operation is waiting and a security attribute changes such that the process no longer has permission to access the contents or message queue object in the required mode, then the operation ceases to wait and terminates in an appropriate mandatory or discretionary access mode error (see C.4).

8.7.3 Errors

(1) Execution of an operation may *terminate* after carrying out the *normal behaviour* as described in the main subclause, or may terminate in an *error*.

(2) The list of errors in the subclause **Errors** of each abstract operation definition defines the set of *error conditions* which apply to that operation. Other error conditions, which apply to several operations, are defined in clause 23. The error conditions OPERATION_HAS_TIMED_OUT and OPERATION_IS_INTERRUPTED (see 8.7.2) and SECURITY_POLICY_WOULD_BE_VIOLATED (see 20.1.8) apply to all operations. If an operation terminates in an error then the

associated error condition holds. If any error condition holds then the operation terminates; if none of the error conditions hold, the normal behaviour occurs.

(3) Error conditions are distinguished for the purpose of helping tool writers. Language bindings may add further error conditions. An implementation may not add further error conditions, except as specified in this ECMA Standard.

(4) No precedences are defined in this ECMA Standard between error conditions which hold simultaneously. Implementations which aim for high security must define such a precedence so as to address the problem of covert channels.

(5) Error conditions arising from type mismatches between actual and formal parameters of operations are not explicitly defined in the abstract operation definitions. Language bindings may need to make these error conditions explicit, depending on the strength of type-checking provided by the language. This does not apply to the following cases:

(6) - a check by an operation that an object belongs to a particular subset of instances of a type, e.g. that a security group is not a subgroup, or that an attribute is not applied to a specified object;

(7) - a check by an operation where a specialization of Object_designator is specified and an object reference is supplied (see 23.1.2.2).

(8) If an operation terminates with an error condition, then the operation may have acquired some locks. The locks acquired are implementation-dependent, but in no case may a lock be acquired on a resource (object or link) which is stronger than the lock that would be acquired on that resource if the normal behaviour had occurred. Their duration is determined in the same way as for other locks. No other state changes occur, except that possibly auditing and accounting records are created.

(9) Calls to operations which are part of optional modules which are not implemented by an implementation return with no error and no effect.

8.7.4 Operation serializability

(1) In general, operations are serializable with all other concurrent operations. An operation may be considered to be composed of one or more atomic *actions* which change the state of the PCTE installation. That a set of operations is serializable means that for each operation a single point in time can be determined, lying between the actual time the operation is called and the time of the return from the operation, where all the actions of the operation can be deemed to take place and without any different effect on the state of the PCTE installation to that which actually occurs. This point of time is called the *nominal serialization point*. The following specific exceptions to serializability apply.

(2) - If an operation enters a waiting state then the actions before the operation waits constitute one operation for purposes of serialization and the actions after leaving the waiting state until entering a further waiting state or the end of the operation constitute another operation.

(3) - The values of the "last_access_time", "last_modification_time", "last_change_time", "last_composite_access_time", "last_composite_modification_time", and "last_composite_ change_time" attributes are updated within an operation to a point of time between the start and end of the operation and not necessarily to any nominal serialization point. The time values set on different objects by a single operation are not necessarily the same.

(4) - If PROCESS_INTERRUPT_OPERATION is used on a process between the start of an operation and the nominal serialization point then the operation is interrupted; if it is used after the nominal serialization point then it has no effect.

(5) - Serializability does not apply to PROCESS_SUSPEND for the calling process; nor to WORKSTATION_CHANGE_CONNECTION with the parameter *force* set to **true** and any affected concurrent operations.

(6) - PROCESS_TERMINATE interrupts other ongoing operations (if any) in the same way as PROCESS_INTERRUPT_OPERATION.

NOTES

(7) 1 Serializability is often enforced by locking. However, this is not true for two or more operations running on behalf of the same activity or on behalf of competing unprotected activities. As an example of operation serializability, consider two concurrent invocations of OBJECT_MOVE on the same object, moving it to two different volumes. The result should be that the entire object resides on one or the other volume, rather than some components residing on each volume according to the order in which they were moved.

(8) 2 Evaluation of parameters which are references counts as part of operation execution for serialization.

9 Object management

9.1 Object management concepts

9.1.1 The basic type "object"

(1) **sds** system:

(2) volume_identifier: (**read**) **non_duplicated natural**;

(3) locked_link_name: (**read**) **string**;

(4) lock_identifier: (**read**) **string**;

(5) exact_identifier: (**read**) **non_duplicated string**;

(6) number: **natural**;

(7) name: **string**;

(8) system_key: (**read**) **natural**;

(9)
```
object: with
attribute
    exact_identifier;
    volume_identifier;
    replicated_state: (read) non_duplicated enumeration (NORMAL, MASTER, COPY) :=
        NORMAL;
    last_access_time: (read) non_duplicated time;
    last_modification_time: (read) non_duplicated time;
    last_change_time: (read) non_duplicated time;
    last_composite_access_time: (read) non_duplicated time;
    last_composite_modif_time: (read) non_duplicated time;
    last_composite_change_time: (read) non_duplicated time;
    num_incoming_links: (read) non_duplicated natural;
    num_incoming_composition_links: (read) non_duplicated natural;
    num_incoming_existence_links: (read) non_duplicated natural;
    num_incoming_reference_links: (read) non_duplicated natural;
    num_incoming_stabilizing_links: (read) non_duplicated natural;
    num_outgoing_composition_links: (read) non_duplicated natural;
    num_outgoing_existence_links: (read) non_duplicated natural;
link
    predecessor: (navigate) non_duplicated composite stable existence link
        (predecessor_number: natural) to object reverse successor;
    successor: (navigate) implicit link (system_key) to object reverse predecessor;
    opened_by: (navigate) non_duplicated designation link (number) to process;
    locked_by: (navigate) non_duplicated designation link (lock_identifier) to activity
    with attribute
        locked_link_name;
    end locked_by;
end object;
```

(10)
```
end system;
```

(11) "Object" is the common ancestor type of all objects in the object base.

(12) The exact identifier uniquely identifies the object in the object bases of all PCTE installations. It is composed of a prefix and a suffix separated by ':' (colon). The prefix is the same for all objects created within a PCTE installation. The suffix uniquely identifies the object within the object base of a particular PCTE installation. The exact identifier of an object that has been deleted is never reassigned to an object created later.

(13) The volume identifier identifies the volume on which the object resides, or, for a copy object, on which it is a replica. It uniquely identifies a volume within a PCTE installation.

(14) The replicated state indicates whether the object is normal, a master or a copy (see 17.1). It is MASTER for the master of a replicated object, COPY for a copy of a replicated object, and NORMAL for a non-replicated object. This attribute can be changed only by the operations which manage replicated objects.

(15) The last access time is the date and time of day of the last read access to the contents of the object. It is set to the system time when the object is created and by the following operations (unless the object is on a read-only volume or is a component of an object on a read-only volume):

(16) - QUEUE_RESTORE (*queue, file*) for *file*;

(17) - CONTENTS_READ;

(18) - AUDIT_FILE_READ;

(19) - ACCOUNTING_RECORD_READ;

(20) - CONTENTS_COPY_TO_FOREIGN_SYSTEM (*file*, *foreign_system*, *foreign_parameters*, *foreign_name*) for *file*.

(21) The last modification time is the date and time of day of the last *modification* to the object. It is set to the system time when the object is created and by the following operations:

(22) - LINK_CREATE and LINK_REPLACE for the origin of the created link when the created link is not implicit;

(23) - LINK_DELETE and LINK_REPLACE for the origin of the deleted link when the deleted link is not implicit;

(24) - any operation which results in the creation or deletion of a link which is not implicit, except for usage designation links and "object_on_volume" links;

(25) - OBJECT_CONVERT;

(26) - OBJECT_SET_ATTRIBUTE and OBJECT_SET_SEVERAL_ATTRIBUTES;

(27) - LINK_SET_ATTRIBUTE and LINK_SET_SEVERAL_ATTRIBUTES, for the origins of the links;

(28) - OBJECT_SET_PREFERENCE;

(29) - QUEUE_SAVE (*queue*, *file*) for *file*;

(30) - CONTENTS_WRITE and CONTENTS_TRUNCATE;

(31) - CONTENTS_COPY_FROM_FOREIGN_SYSTEM (*file*, *foreign_system*, *foreign_parameters*, *foreign_name*) for *file*;

(32) - any operation resulting in the creation of an audit record for the audit file;

(33) - any operation resulting in the creation of an accounting record for the accounting log;

(34) The last change time is the date and time of day of the last *change* to the object. It is set by any operation which sets the last modification time, and the following operations:

(35) - creation of an implicit link;

(36) - deletion of an implicit link;

(37) - OBJECT_MOVE for an object which has been moved to another volume;

(38) - operations which change the discretionary access control lists of an object;

(39) - operations which change the mandatory labels of an object;

(40) - operations which change the mandatory label ranges of multi-level secure devices (see 20.1.5);

(41) - PROCESS_SET_CONFIDENTIALITY_LABEL (*process*, *label*) for *process*;

(42) - PROCESS_SET_INTEGRITY_LABEL (*process*, *label*) for *process*.

(43) The last composite access time is the date and time of day of the last read access to the contents of the object or of any component of the object (but is not updated if the object or component is on a read-only volume).

(44) The last composite modif[ication] time is the date and time of day of the last modification to the object or to any component of the object.

(45) The last composite change time is the date and time of day of the last change made to the object or to any component of the object.

(46) An operation which updates the last modification time of an object is said to *atomically modify* the object. An operation which updates the last composite modification time of an object is said to *compositely modify* the object.

(47) The num (number of) incoming links is the number of non-designation links to the object (and is also the number of non-designation links of the object since every non-designation link has a reverse link).

(48) The num (number of) incoming composition links is the number of composition links to the object.

(49) The num (number of) incoming existence links is the number of existence links to the object.

(50) The num (number of) incoming reference links is the number of reference links to the object.

(51) The num (number of) incoming stabilizing links is the number of atomically stabilizing links to the object plus the number of compositely stabilizing links to the object and to its outer objects.

(52) The num (number of) outgoing composition links is the number of composition links of the object.

(53) The num (number of) outgoing existence links is the number of existence links of the object.

(54) The destinations of the "predecessor" links are the immediate predecessor versions of the object; the destinations of the "successor" links are the immediate successor versions of the object. These are used in version control operations; see 9.4. The directed graph of versions created by these links must be acyclic.

(55) The destination of the "opened_by" link is the process that opened the object; see 12.1.

(56) The destination of the "locked_by" link is the activity that has locked the object or a link of the object; see 16.1.2.

(57) There are also attributes defined in the security SDS representing the security properties of the object (see 19.1.2 and 20.1.1); and attributes defined in the accounting SDS representing the accounting properties of the object (see 22.1.1).

(58) The attribute types "number", "name" and "system_key" are predefined. "number" and "name" are used for numeric and string keys, respectively; names are non-empty. "system_key" is the attribute type of system-assigned keys of implicit links (see 8.3.3).

NOTES

(59) 1 The prefix of the object exact identifier is intended to be unique among all PCTE installations, past, present, and future; but the administration of prefix assignment is outside the scope of this ECMA Standard.

(60) 2 The last composite access, modification, and change time may be calculated when required as the most recent of the last access, modification, and change time respectively of the object and its components.

9.1.2 The common root

(1)
```
sds system:
```

(2)
```
common_root: child type of object with
link
    archives: (navigate) existence link to archive_directory reverse archives_of;
    execution_sites: (navigate) existence link to execution_site_directory reverse
        execution_sites_of;
    ground: (protected) existence link to common_root;
    replica_sets: (navigate) existence link to replica_set_directory reverse replica_sets_of;
    volumes: (navigate) existence link to volume_directory reverse volumes_of;
end common_root;
```

(3)
```
end system;
```

(4) The common root has an existence link to each of the *administrative objects* of the PCTE installation: the SDS directory (see 10.1.1), the volume directory (see 11.1.1), the archive directory (see 11.1.4), the replica set directory (see 17.1.1), the execution site directory, (see 18.1.1), the security group directory (see 19.1.1), the mandatory directory (see 20.1.1), and the accounting directory (see 22.1.1).

(5) The "ground" link is to allow deletion of other existence links to the common root (see 9.2.2).

(6) The common root and the administrative objects are predefined replicated objects (see 17.1.4).

9.1.3 Datatypes for object management

(1)
```
Type_ancestry = EQUAL_TYPE | ANCESTOR_TYPE | DESCENDANT_TYPE |
UNRELATED_TYPE
```

(2)
```
Version_relation = ANCESTOR_VSN | DESCENDANT_VSN | SAME_VSN | RELATED_VSN |
UNRELATED_VSN
```

(3) These datatypes are used as parameter and result types of operations in 9.3 and 9.4.

9.2 Link operations

9.2.1 LINK_CREATE

(1)
```
LINK_CREATE (
    origin          : Object_designator,
    new_link        : Link_designator,
    dest            : Object_designator,
    reverse_key     : [ Actual_key ]
)
```

(2) LINK_CREATE creates a new link *link* of *origin* as follows:

(3) - the link type is as specified by the link designator *new_link*;

(4) - the destination is the object *dest*;

(5) - the key attributes are as specified by the link designator *new_link*;

(6) - the non-key attributes are set to their initial values;

(7) - the category, exclusiveness, stability, and duplication are set according to the link type.

(8) If the type of *link* has a reverse link type, LINK_CREATE also creates the reverse link *reverse_link* of *link* and adds it to the links of *dest*:

(9) - the link type of *reverse_link* is the reverse of the link type of *link*;

(10) - the destination of *reverse_link* is *origin*;

(11) - the category, exclusiveness, stability, and duplication of *reverse_link* are set according to the link type.

(12) - the non-key attributes of *reverse_link* are set to their initial values;

(13) - if the type of *reverse_link* is of cardinality many and of category IMPLICIT then the key attribute of *reverse_link* is set to a new system-generated key.

(14) If the type of *reverse_link* is of category COMPOSITION, REFERENCE or EXISTENCE, two cases arise:

(15) - if *dest* has a preferred link type which is the link type of the reverse link, then the key attributes of *reverse_link* are derived from *reverse_key* and from the preferred link key of *dest*, if any, as defined in 23.1.2.7;

(16) - if *dest* has no preferred link type or if the preferred link type of *dest* is not the link type of the reverse link, then the key of the reverse link is set to *reverse_key*.

(17) If *new_link* is a composition link, then any security group that has OWNER granted or denied to *origin* has OWNER or CONTROL_DISCRETIONARY granted or denied respectively to *dest*; similarly if *reverse_link* is a composition link, then any security group that has OWNER granted or denied to *dest* has OWNER or CONTROL_DISCRETIONARY granted or denied respectively to *origin*. This requires the process to have OWNER rights on *dest* or *origin* respectively. See 19.1.2 for details.

(18) Write locks of the default mode are obtained on the new links. A read lock of the default mode is obtained on *origin* if the interpretation of *new_link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7). A read lock of the default mode is obtained on *dest* if the interpretation of *reverse_key* implies the evaluation of any '+' or '++' key attribute values.

(19) A write lock of the default mode is obtained on *dest* and each of its components if the OWNER discretionary access right is granted or denied for one or more groups to *origin*, and different OWNER discretionary access rights exist for one or more of those same groups to *dest*.

Errors

(20) ACCESS_ERRORS (*origin*, ATOMIC, MODIFY, APPEND_LINKS)

(21) If *reverse_link* is implicit:
ACCESS_ERRORS (*dest*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(22) If *reverse_link* is not implicit:
ACCESS_ERRORS (*dest*, ATOMIC, MODIFY, APPEND_LINKS)

(23) If *new_link* is atomically stabilizing:
ACCESS_ERRORS (*dest*, ATOMIC, CHANGE, STABILIZE)

(24) If *new_link* is compositely stabilizing:
ACCESS_ERRORS (*dest*, COMPOSITE, CHANGE, STABILIZE)

(25) CATEGORY_IS_BAD (*origin*, *new_link*, (COMPOSITION, EXISTENCE , REFERENCE, DESIGNATION))

(26) COMPONENT_ADDITION_ERRORS (*dest*, *new_link*)

(27) COMPONENT_ADDITION_ERRORS (*origin*, *reverse_link*)

(28) DESTINATION_OBJECT_TYPE_IS_INVALID (*origin*, *new_link*, *dest*)

(29) LINK_EXISTS (*origin*, *new_link*)

(30) If *link* is atomically or compositely stabilizing:
OBJECT_CANNOT_BE_STABILIZED (*dest*)

(31) If *link* is compositely stabilizing:
OBJECT_CANNOT_BE_STABILIZED (component of *dest*)

(32) REVERSE_KEY_IS_BAD (*origin*, *new_link, dest*, *reverse_key*)

(33) REVERSE_KEY_IS_NOT_SUPPLIED (*origin*, *new_link, dest*)

(34) REVERSE_KEY_IS_SUPPLIED (*reverse_key*)

(35) REVERSE_LINK_EXISTS (*origin*, *new_link, dest, reverse_key)*

(36) UPPER_BOUND_WOULD_BE_VIOLATED (*dest*, *reverse_link*)

(37) UPPER_BOUND_WOULD_BE_VIOLATED (*origin*, *new_link*)

(38) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*origin*, *new_link*, CREATE_MODE)

9.2.2 LINK_DELETE

(1)
```
LINK_DELETE (
     origin  : Object_designator,
     link    : Link_designator
)
```

(2) LINK_DELETE deletes the link specified by *origin* and *link*.

(3) Let *dest* be the destination of *link*, and *reverse_link* be the reverse link of *link* (if any). LINK_DELETE deletes *link* from the links of *origin* and deletes *reverse_link* (if any) from the links of *dest*.

(4) *dest* is deleted from the object base if *link* is the last composition or existence link to *dest*.

(5) *origin* is deleted from the object base if *reverse_link* is the last composition or existence link to *origin*.

(6) For each deleted object the "object_on_volume" link from the volume on which the deleted object was residing to the deleted object is also deleted.

(7) If either deleted object is opened by one or more processes (see 12.1), the deletion of its contents is postponed until all processes have closed the contents. An operation using a contents handle to access its contents is not affected by the deletion until the contents handle is closed.

(8) Write locks of the default mode are obtained on the deleted objects (if any) and on the deleted links except the "object_on_volume" link. A read lock of the default mode is obtained on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(9) ACCESS_ERRORS (*origin*, ATOMIC, MODIFY, WRITE_LINKS)

(10) If *link* is atomically stabilizing:
ACCESS_ERRORS (*dest*, ATOMIC, CHANGE, STABILIZE)

(11) If *link* is compositely stabilizing:
ACCESS_ERRORS (*dest*, COMPOSITE, CHANGE, STABILIZE)

(12) If *reverse_link* is implicit:
ACCESS_ERRORS (*dest*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(13) If *reverse_link* is not implicit:
ACCESS_ERRORS (*dest*, ATOMIC, MODIFY, WRITE_LINKS)

(14) If *link* is the last composition or existence link to *dest*:
ACCESS_ERRORS (*dest*, ATOMIC, MODIFY, DELETE)

(15) If *reverse_link* is the last composition or existence link to *origin*:
ACCESS_ERRORS (*origin*, ATOMIC, MODIFY, DELETE)

(16) For each origin X of an implicit link to a deleted object:
ACCESS_ERRORS (X, ATOMIC, CHANGE, WRITE_IMPLICIT)

(17) For each compositely stabilizing link L of a deleted object:
ACCESS_ERRORS (destination of L, COMPOSITE, CHANGE, STABILIZE)

(18) CATEGORY_IS_BAD (*origin*, *link*, (COMPOSITION, EXISTENCE, REFERENCE, DESIGNATION))

(19) If *link* is not a designation link:
DESTINATION_OBJECT_TYPE_IS_INVALID (*origin*, *link*, *dest*)

(20) LOWER_BOUND_WOULD_BE_VIOLATED (*origin*, *link*)

(21) LOWER_BOUND_WOULD_BE_VIOLATED (*dest*, *reverse_link*)

(22) If *reverse_link* is the last existence or composition link to *origin*:
OBJECT_HAS_LINKS_PREVENTING_DELETION (*origin*)

(23) If *link* is the last existence or composition link to *dest*:
OBJECT_HAS_LINKS_PREVENTING_DELETION (*dest*)

(24) If *link* is the last composition or existence link to *dest*:
OBJECT_IS_IN_USE_FOR_DELETE (*dest*)

(25) If *reverse_link* is the last composition or existence link to *origin*:
OBJECT_IS_IN_USE_FOR_DELETE (*origin*)

(26) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, DELETE_MODE)

9.2.3 LINK_DELETE_ATTRIBUTE

(1)
```
LINK_DELETE_ATTRIBUTE (
    origin       : Object_designator,
    link         : Link_designator,
    attribute    : Attribute_designator
)
```

(2) LINK_DELETE_ATTRIBUTE deletes the non-key attribute *attribute* of the link *link* of the object *origin*, if the "attribute_type" object representing the attribute type of *attribute* is no longer in the object base.

(3) A write lock of the default mode is obtained on *link*. A read lock of the default mode is obtained on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(4) ACCESS_ERRORS (*origin*, ATOMIC, MODIFY, WRITE_LINKS)

(5) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(6) REFERENCED_OBJECT_IS_NOT_MUTABLE (key of *link*)

(7) NOTE - It is the responsibility of the user to ensure that the attribute type is no longer in the object base.

9.2.4 LINK_GET_ATTRIBUTE

(1)
```
LINK_GET_ATTRIBUTE (
    origin      : Object_designator,
    link        : Link_designator,
    attribute   : Attribute_designator
)
    result      : Attribute_value
```

(2) LINK_GET_ATTRIBUTE returns the value of the non-key attribute *attribute* of the link *link* of the object *origin*.

(3) A read lock of the default mode is obtained on *link*. A read lock of the default mode is obtained on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(4) ACCESS_ERRORS (*origin*, ATOMIC, READ, READ_LINKS)

(5) USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, *attribute*, READ_MODE)

9.2.5 LINK_GET_DESTINATION_VOLUME

(1)
```
LINK_GET_DESTINATION_VOLUME (
    origin          : Object_designator,
    link            : Link_designator
)
    destination     : Volume_info
```

(2) LINK_GET_DESTINATION_VOLUME returns the volume identifier *volume_identifier* and the volume accessibility *mounted* of the volume on which the destination *dest* of the link *link* of the object *origin* resides. The returned value of *mounted* is as follows:

(3) - ACCESSIBLE if the volume on which *dest* resides is mounted and is accessible in the network partition that contains the calling procedure's workstation. In this case, *volume_identifier* is the volume identifier of the volume on which *dest* resides.

(4) - INACCESSIBLE if the PCTE implementation is able to determine on which volume the object resides, and that volume is not accessible (either because the volume is not mounted or because the volume is mounted in a network partition which does not contain the calling process's workstation). In this case, *volume_identifier* is the volume identifier of the volume on which the object resides.

(5) - UNKNOWN if the PCTE implementation is unable to determine on which volume the object resides. In this case, *volume_identifier* is the volume identifier of a volume which is not currently accessible.

(6) The situations in which UNKNOWN is returned rather than INACCESSIBLE, and vice versa, are implementation-defined, as is the general meaning of the volume identifier returned for UNKNOWN. In any particular situation, the choice is implementation-dependent.

(7) Read locks of the default mode are obtained on *link*, and on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(8) ACCESS_ERRORS (*origin*, ATOMIC, READ, READ_LINKS)

(9) LINK_DESTINATION_DOES_NOT_EXIST (*link*)

(10) OBJECT_IS_ARCHIVED (destination of *link*)

(11) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, NAVIGATE_MODE)

(12) NOTE - Some implementations may be able to guarantee that only ACCESSIBLE or INACCESSIBLE is returned. For implementations that return UNKNOWN, the volume identifier returned should be that of the volume which should be made accessible prior to repeating the call. The destination object may reside on this volume, or it may contain implementation-dependent details of the volume on which the object resides, or of a further volume to be made accessible. Although the operation may require access to other volumes, no error condition is raised as a result.

9.2.6 LINK_GET_KEY

(1)
```
LINK_GET_KEY (
    origin  : Object_designator,
    link    : Link_designator
)
    key     : [ Actual_key ]
```

(2) LINK_GET_KEY returns in *key* the complete sequence of key attribute values (if any) of the link *link* of the object *origin*.

(3) Read locks of default mode are obtained on *link*, and on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(4) ACCESS_ERRORS (*origin*, ATOMIC, READ, READ_LINKS)

(5) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, NAVIGATE_MODE)

9.2.7 LINK_GET_REVERSE

(1)
```
LINK_GET_REVERSE (
    origin          : Object_designator,
    link            : Link_designator
)
    reverse_link    : [ Link_designator ],
    dest            : Object_designator
```

(2) LINK_GET_REVERSE returns in *reverse_link* the reverse link (if there is one) of the link *link* of the object *origin*, and in *dest* the destination of *link*. If *link* has no reverse link, no value is returned in *reverse_link*.

(3) Read locks of the default mode are obtained on *link*, and on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(4) ACCESS_ERRORS (*origin*, ATOMIC, READ, READ_LINKS)

(5) ACCESS_ERRORS (destination of *link*, ATOMIC, READ, READ_LINKS)

(6) LINK_DESTINATION_DOES_NOT_EXIST (*link*)

(7) LINK_NAME_IS_TOO_LONG_IN_CURRENT_WORKING_SCHEMA

(8) REFERENCE_CANNOT_BE_ALLOCATED

(9) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, NAVIGATE_MODE)

9.2.8 LINK_GET_SEVERAL_ATTRIBUTES

(1)
```
LINK_GET_SEVERAL_ATTRIBUTES (
    origin      : Object_designator,
    link        : Link_designator,
    attributes  : Attribute_selection
)
    values      : Attribute_assignments
```

(2) LINK_GET_SEVERAL_ATTRIBUTES returns in *values* a set of attribute assignments of the link *link* of the object *origin*.

(3) The returned set of attributes is determined by *attributes*:

(4) - a set of attribute designators: the set of non-key attributes, as for LINK_GET_ATTRIBUTE (*origin*, *link*, A) for each attribute A of *attributes*;

(5) - VISIBLE_ATTRIBUTE_TYPES: all non-key attributes of *link* visible in the working schema of the calling process and with usage mode including READ_MODE.

(6) Read locks of the default mode are obtained on *link*, and on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(7) ACCESS_ERRORS (*origin*, ATOMIC, READ, READ_LINKS)

(8) If *attributes* is not VISIBLE_ATTRIBUTE_TYPES:
USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, each element of *attributes*, READ_MODE)

9.2.9 LINK_REPLACE

(1)
```
LINK_REPLACE (
    origin            : Object_designator,
    link              : Link_designator,
    new_origin        : Object_designator,
    new_link          : Link_designator,
    new_reverse_key   : [ Actual_key ]
)
```

(2) LINK_REPLACE replaces the composition or existence link *link* of the object *origin* by a new composition or existence link as specified by *new_link* from *new_origin*, with the same destination *dest*.

(3) The new link from *new_origin* to *dest* is created and *link* is deleted in the same way as the following sequence of operations, ignoring any temporary violation of link bounds or composition exclusivity on *dest*:

(4) `LINK_CREATE (new_origin, new_link, dest, new_reverse_key);`

(5) `LINK_DELETE (origin, link)`

(6) Write locks of the default mode are obtained on the links to be deleted and on the new links. A read lock of the default mode is obtained on *new_origin* if the interpretation of *link* or *new_link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7). A read lock of the

default mode is obtained on *dest* if the interpretation of *new_reverse_key* implies the evaluation of any '+' or '++' key attribute values.

(7) A write lock of the default mode is obtained on *dest* and each of its components if the OWNER discretionary access right is granted or denied for one or more groups to *new_origin*, and different OWNER discretionary access rights exist for one or more of those same groups to *dest*.

Errors

(2) ACCESS_ERRORS (*new_origin*, ATOMIC, MODIFY, APPEND_LINKS)

(3) ACCESS_ERRORS (*origin*, ATOMIC, MODIFY, WRITE_LINKS)

(4) If the reverse link of *new_link* is implicit:
ACCESS_ERRORS (*dest*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(5) If the reverse link of *new_link* is not implicit:
ACCESS_ERRORS (*dest*, ATOMIC, MODIFY, APPEND_LINKS)

(6) If the reverse link of *link* is implicit:
ACCESS_ERRORS (*dest*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(7) If the reverse link of *link* is not implicit:
ACCESS_ERRORS (*dest*, ATOMIC, MODIFY, WRITE_LINKS)

(8) If *new_link* is atomically stabilizing and *link* is not, or vice versa:
ACCESS_ERRORS (*dest*, ATOMIC, CHANGE, STABILIZE)

(9) If *new_link* is compositely stabilizing and *link* is not, or vice versa:
ACCESS_ERRORS (*dest*, COMPOSITE, CHANGE, STABILIZE)

(10) CATEGORY_IS_BAD (*new_origin*, *new_link*, (COMPOSITION, EXISTENCE))

(11) CATEGORY_IS_BAD (*origin*, *link*, (COMPOSITION, EXISTENCE))

(12) COMPONENT_ADDITION_ERRORS (*dest*, *new_link*)

(13) DESTINATION_OBJECT_TYPE_IS_INVALID (*new_origin*, *new_link*, *dest*.)

(14) DESTINATION_OBJECT_TYPE_IS_INVALID (*origin*, *link*, *dest*)

(15) If *new_link* is of category COMPOSITION, and *new_origin* has OWNER granted or denied:
LINK_EXISTS (*new_origin*, *new_link*)

(16) LINK_EXISTS (*dest*, *new_reverse_key*)

(17) LOWER_BOUND_WOULD_BE_VIOLATED (*origin*, *link*)

(18) OBJECT_CANNOT_BE_STABILIZED (*dest*)

(19) REVERSE_KEY_IS_BAD (*new_origin*, *new_link*, *new_reverse_key*)

(20) REVERSE_KEY_IS_NOT_SUPPLIED (*new_origin*, *new_link*, *dest*)

(21) UPPER_BOUND_WOULD_BE_VIOLATED (*dest*, reverse link of *new_link*)

(22) UPPER_BOUND_WOULD_BE_VIOLATED (*new_origin*, *new_link*)

(23) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, DELETE_MODE)

(24) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*new_origin*, *new_link*, CREATE_MODE)

9.2.10 LINK_RESET_ATTRIBUTE

(1)
```
LINK_RESET_ATTRIBUTE (
    origin      : Object_designator,
    link        : Link_designator,
    attribute   : Attribute_designator
)
```

(2) LINK_RESET_ATTRIBUTE resets the non-key attribute *attribute* of the link *link* of the object *origin* to its initial value.

(3) A write lock of the default mode is obtained on *link*. A read lock of the default mode is obtained on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(4) ACCESS_ERRORS (*object*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(5) KEY_UPDATE_IS_FORBIDDEN (*attribute*)

(6) REFERENCED_OBJECT_IS_NOT_MUTABLE (key of *link*)

(7) USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, *attribute*, WRITE_MODE)

9.2.11 LINK_SET_ATTRIBUTE

(1)
```
LINK_SET_ATTRIBUTE (
    origin      : Object_designator,
    link        : Link_designator,
    attribute   : Attribute_designator,
    value       : Attribute_value
)
```

(2) LINK_SET_ATTRIBUTE assigns the value *value* to the non-key attribute *attribute* of the link *link* of the object *origin*.

(3) A write lock of the default mode is obtained on *link*. A read lock of the default mode is obtained on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(4) ACCESS_ERRORS (*origin*, ATOMIC, MODIFY, WRITE_LINKS)

(5) ENUMERATION_VALUE_IS_OUT_OF_RANGE (*value*, values of *attribute*)

(6) KEY_UPDATE_IS_FORBIDDEN (*origin*, *link*, *attribute*)

(7) If *link* is a "referenced_object" link:
REFERENCED_OBJECT_IS_NOT_MUTABLE (key of *link*)

(8) USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, *attribute*, WRITE_MODE)

(9) VALUE_LIMIT_ERRORS (*value*)

(10) The following implementation-dependent error may be raised:
VALUE_TYPE_IS_INVALID (*value*, *origin*, *link*, *attribute*)

9.2.12 LINK_SET_SEVERAL_ATTRIBUTES

(1)
```
LINK_SET_SEVERAL_ATTRIBUTES (
    origin      : Object_designator,
    link        : Link_designator,
    attributes  : Attribute_assignments
)
```

(2) For each element A in the domain of *attributes*, LINK_SET_SEVERAL_ATTRIBUTES sets the value of the attribute A of the link *link* of the object *origin* to the value *attributes* (A), in the same way as:

(3)
```
LINK_SET_ATTRIBUTE (origin, link, A, attributes (A))
```

(4) A write lock of the default mode is obtained on *link*. A read lock of the default mode is obtained on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(5) ACCESS_ERRORS (*origin*, ATOMIC, MODIFY, WRITE_LINKS)

(6) For each element A in the domain of *attributes*:
ENUMERATION_VALUE_IS_OUT_OF_RANGE (*attribute*(A), values of A)
KEY_UPDATE_IS_FORBIDDEN (*origin*, *link*, A)
USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, A, WRITE_MODE)
VALUE_LIMIT_ERRORS (*attributes* (A))

(7) The following implementation-dependent error may be raised for each element A in the domain of *attributes*:
VALUE_TYPE_IS_INVALID (*attributes* (A), *origin*, *link*, A)

9.3 Object operations

9.3.1 OBJECT_CHECK_TYPE

(1)
```
OBJECT_CHECK_TYPE(
    object      : Object_designator,
    type2       : Object_type_nominator
)
    relation    : Type_ancestry
```

(2) OBJECT_CHECK_TYPE compares the object type *type1* of the object *object* against the object type *type2*, and returns in *relation* a value defined as follows:

(3) - EQUAL_TYPE if *type1* is the same as *type2*;

(4) - ANCESTOR_TYPE if *type1* is an ancestor of *type2*;

(5) - DESCENDANT_TYPE if *type1* is a descendant type of *type2* in the working schema of the calling process;

(6) - UNRELATED_TYPE in all other cases.

(7) The visibility of the type of *object* does not affect the result of the operation.

(8) A read lock of the default mode is obtained on *object*.

Errors

(9) ACCESS_ERRORS (*object*, ATOMIC, READ)

(10) OBJECT_TYPE_IS_UNKNOWN (*type2*)

(11) VOLUME_IS_INACCESSIBLE (*object*, ATOMIC)

9.3.2 OBJECT_CONVERT

(1)
```
OBJECT_CONVERT(
    object  : Object_designator,
    type    : Object_type_nominator
)
```

(2) OBJECT_CONVERT changes the object type of the object *object* to the descendant object type *type*.

(3) The operation has no effect if the current type of *object* is already *type*.

(4) A write lock of the default mode is obtained on *object*.

Errors

(5) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, CONTROL_OBJECT)

(6) OBJECT_IS_NOT_CONVERTIBLE (*object*)

(7) OBJECT_IS_STABLE (*object*)

(8) OBJECT_TYPE_IS_INVALID (*type*)

(9) OBJECT_TYPE_IS_UNKNOWN (*type*)

(10) TYPE_IS_NOT_DESCENDANT (object type of *object*, *type*)

(11) If *object* is not of type *type*:
USAGE_MODE_ON_OBJECT_TYPE_WOULD_BE_VIOLATED (current type of object, *type*)

9.3.3 OBJECT_COPY

(1)
```
OBJECT_COPY (
    object              : Object_designator,
    new_origin          : Object_designator,
    new_link            : Link_designator,
    reverse_key         : [ Actual_key ],
    on_same_volume_as   : [ Object_designator ],
    access_mask         : Atomic_access_rights
)
    new_object          : Object_designator
```

(2) OBJECT_COPY creates an object *new_object* as a *copy* of *object*. More precisely:

(3) - If *object* is a file, an accounting log, or an audit file, then the contents of *new_object* is the same as the contents of *object*; if *object* is a pipe then the contents of *new_object* is empty.

(4) - For each duplicable attribute X of *object*, there is an attribute of *new_object* which is a copy of X.

(5) - For each duplicable direct component X of *object*, there is a direct component of *new_object* which is a copy of X.

(6) - For each duplicable internal link A of *object* whose destination is a duplicable component, there is an internal link B of *new_object* such that the destination of B is the copy of the destination of A and all other properties of B are the same as for A.

(7) - For each duplicable external link A of *object*, there is a corresponding external link of *new_object* which is a copy of A.

(8) - For each non-duplicable attribute of *object,* there exists a corresponding attribute of *new_object* whose value is either set to the initial value of the attribute type or, for the following predefined attributes: the exact identifier, volume identifier, replicated state, contents type, last access time, last modification time, last change time, last composite access time, last composite modification time, last composite change time, number of incoming links, number of incoming composition links, number of incoming existence links, number of incoming reference links, number of incoming stabilizing links, number of outgoing composition links, number of outgoing existence links, atomic ACL, composite ACL, confidentiality label, and integrity label, is set to a value corresponding to the newly created object. In particular, since the attribute "replicated_state" is not copied, the copy of a replicated object is not replicated.

(9) - For each non-duplicable non-key attribute of a copied link, there exists a corresponding non-key attribute of the copied link whose value is set to the initial value of the attribute type.

(10) A *copy* of an attribute A is an attribute which has the same attribute type, attribute value, and attribute properties as A.

(11) A *copy* of a link A is a link which has the same link type, key attributes, duplicable non-key attributes, link properties, and destination as A.

(12) A *copy* of a component X of an object A is a component Y of a copy B of A such that Y is a copy of X as an object, and the composition link from B to Y is a copy of the composition link from X to A

(13) OBJECT_COPY creates a link *link* of the object *new_origin*, as specified by *new_link*, and with *new_object* as destination, and its reverse link *reverse_link* with origin *new_object* and key derived from *reverse_key* as described in 23.1.2.7.

(14) *access_mask* is used in conjunction with the default atomic ACL and default object owner of the calling process to specify the atomic ACL and the composite ACL of *new_object* and its components. See 19.1.4 for more details.

(15) If *new_link* is a composition link, then any security group that has OWNER granted or denied to *new_origin* has OWNER granted or denied respectively to *dest*; similarly if *reverse_link* is a composition link, then any security group that has OWNER granted or denied to *dest* has OWNER granted or denied respectively to *new_origin*.

(16) If *new_link* is a composition link, then any security group that has OWNER granted or denied to *origin* has OWNER granted or denied respectively to *dest*; similarly if *reverse_link* is a composition link, then any security group that has OWNER granted or denied to *dest* has OWNER granted or denied respectively to *origin*.

(17) *new_object* has the same integrity and confidentiality label as *object* and each component of *new_object* has the same integrity and confidentiality labels as the corresponding component of *object*.

(18) If *on_same_volume_as* is supplied, *new_object* resides on the same volume as the object *on_same_volume_as*. Otherwise, *new_object* resides on the same volume as *object* and each component of *new_object* resides on the same volume as its corresponding component in *object*.

(19) An "object_on_volume" link is created from the volume on which *new_object* resides to *new_object*, and similarly for all its components. Each created link is keyed by the exact identifier of its destination object.

(20) Read locks of the default mode are obtained on *object* and on all its components; write locks of the default mode are obtained on the new objects and links except the new "object_on_volume" links. A read lock of the default mode is obtained on *new_object* if the interpretation of the link *new_link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

(21) If *object* is an accounting log, its contents is preserved.

Errors

(22) ACCESS_ERRORS (*new_origin*, ATOMIC, MODIFY, APPEND_LINKS)

(23) ACCESS_ERRORS (*object*, COMPOSITE, READ, READ_LINKS)

(24) ACCESS_ERRORS (*object*, COMPOSITE, READ, READ_ATTRIBUTES)

(25) ACCESS_ERRORS (*object*, COMPOSITE, READ, READ_CONTENTS)

(26) ACCESS_ERRORS (*on_same_volume_as*, ATOMIC, SYSTEM_ACCESS)

(27) If *new_link* is compositely stabilizing:
    ACCESS_ERRORS (*new_object*, COMPOSITE, CHANGE, STABILIZE)

(28) If *reverse_link* is compositely stabilizing:
    ACCESS_ERRORS (*new_origin*, COMPOSITE, CHANGE, STABILIZE)

(29) For each destination X of a duplicable external link L of a duplicated component:

(30)   ACCESS_ERRORS (X, ATOMIC, CHANGE, APPEND_IMPLICIT)

(31)   If L is atomically stabilizing:
        ACCESS_ERRORS (X, ATOMIC, CHANGE, STABILIZE)

(32)   If L is compositely stabilizing:
        ACCESS_ERRORS (X, COMPOSITE, CHANGE, STABILIZE)

(33) CATEGORY_IS_BAD (*new_origin*, *new_link*, (COMPOSITION, EXISTENCE))

(34) If *object* is a component of itself, and *new_link* is a composition link:
    COMPONENT_ADDITION_ERRORS (*new_object*, *new_link*)

(35) CONTROL_WOULD_NOT_BE_GRANTED (*new_object*)

(36) DESTINATION_OBJECT_TYPE_IS_INVALID (*new_origin*, *new_link*, *new_object*)

(37) EXTERNAL_LINK_IS_BAD (*object*, COMPOSITE)

(38) EXTERNAL_LINK_IS_NOT_DUPLICABLE (*object)*

(39) LABEL_IS_OUTSIDE_RANGE (*new_object*, volume on which *on_same_volume_as* resides)

(40) LINK_EXISTS (*new_origin*, *new_link*)

(41) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL (*new_object*)

(42) If *object* is not a component of itself, *new_link* is a composition link, and *new_origin* has OWNER granted or denied:
    OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_object*)

(43) REFERENCE_CANNOT_BE_ALLOCATED

(44) REVERSE_KEY_IS_BAD (*new_origin*, *new_link*, *new_object*, *reverse_key*)

(45) REVERSE_KEY_IS_NOT_SUPPLIED (*new_origin*, *new_link*, *new_object*)

(46) REVERSE_KEY_IS_SUPPLIED (*reverse_key*)

(47) REVERSE_LINK_EXISTS (*new_origin*, *new_link*, *new_object*, *reverse_key*)

(48) TYPE_OF_OBJECT_IS_INVALID (*object*, COMPOSITE)

(49) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*new_origin, new_link,* CREATE)

(50) USAGE_MODE_ON_OBJECT_TYPE_WOULD_BE_VIOLATED ("object", type of *object*)

(51) VOLUME_IS_FULL (volume on which *on_same_volume_as* resides)

(52) NOTE - Key values of reverse links (which must be implicit of cardinality many) are system-generated, because they must be different from key values of other links of the same types from the same origins.

9.3.4 OBJECT_CREATE

(1)
```
OBJECT_CREATE (
    type                : Object_type_nominator,
    new_origin          : Object_designator,
    new_link            : Link_designator,
    reverse_key         : [ Actual_key ],
    on_same_volume_as   : [ Object_designator ],
    access_mask         : Atomic_access_rights
)
    new_object          : Object_designator
```

(2) OBJECT_CREATE creates an object *new_object* as follows:

(3) - the object type of *new_object* is *type*;

(4) - the contents of *new_object* is empty;

(5) - the value of each attribute of *new_object* is the initial value of its attribute type, except for some predefined attributes, set as defined below.

(6) A composition or existence link, as specified by *new_link*, is created from *new_origin* to *new_object*, together with its reverse link *reverse_link*, with key *reverse_key*.

(7) *access_mask* is used in conjunction with the default atomic ACL and default object owner of the calling process to specify the atomic ACL and the composite ACL of *new_object*. See 19.1.4 for more details.

(8) If *new_link* is a composition link, then any security group that has OWNER granted or denied to *new_origin* has OWNER granted or denied respectively to *new_object*; similarly if *reverse_link* is a composition link, then any security group that has OWNER granted or denied to *new_object* has OWNER granted or denied respectively to *new_origin*.

(9) The confidentiality label of *new_object* is set to the current confidentiality context of the calling process and the integrity label of *new_object* is set to the current integrity context of the calling process.

(10) If *on_same_volume_as* is supplied, *new_object* resides on the same volume as the object *on_same_volume_as*. Otherwise, *new_object* resides on the same volume as *new_origin*.

(11) An "object_on_volume" link is created from the volume on which *new_object* resides to *new_object*. Each created link is keyed by the exact identifier of its destination object.

(12) Write locks of the default mode are obtained on *new_object* and on *new_link*. A read lock of the default mode is obtained on *new_origin* if the interpretation of *new_link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(13) ACCESS_ERRORS (*new_origin*, ATOMIC, MODIFY, APPEND_LINKS)

(14) If *new_link* is atomically stabilizing:
ACCESS_ERRORS (*new_object*, ATOMIC, CHANGE, STABILIZE)

(15) If *new_link* is compositely stabilizing:
ACCESS_ERRORS (*new_object*, COMPOSITE, CHANGE, STABILIZE)

(16) If *reverse_link* is compositely stabilizing:
ACCESS_ERRORS (*new_origin*, COMPOSITE, CHANGE, STABILIZE)

(17) ACCESS_ERRORS (*on_same_volume_as*, ATOMIC, SYSTEM_ACCESS)

(18) CATEGORY_IS_BAD (*new_origin*, *new_link*, (COMPOSITION, EXISTENCE)

(19) CONTROL_WOULD_NOT_BE_GRANTED (*new_object*)

(20) DESTINATION_OBJECT_TYPE_IS_INVALID (*new_origin*, *new_link*, *new_object*)

(21) LABEL_IS_OUTSIDE_RANGE (*new_object*, volume on which *on_same_volume_as* resides)

(22) LINK_EXISTS (*new_origin*, *new_link*)

(23) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL (*new_object*)

(24) OBJECT_TYPE_IS_INVALID (*type*)

(25) OBJECT_TYPE_IS_UNKNOWN (*type*)

(26) If *new_link* is a composition link and *new_origin* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_object*)

(27) If *reverse_link* is a composition link and *new_object* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_object*)

(28) REFERENCE_CANNOT_BE_ALLOCATED

(29) REVERSE_KEY_IS_NOT_SUPPLIED (*new_origin*, *new_link*, *new_object*)

(30) REVERSE_KEY_IS_BAD (*new_origin*, *new_link*, *new_object*, *reverse_key*)

(31) REVERSE_KEY_IS_SUPPLIED (*reverse_key*)

(32) UPPER_BOUND_WOULD_BE_VIOLATED (*new_origin*, *new_link*)

(33) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*new_origin*, *new_link*, CREATE_MODE)

(34) USAGE_MODE_ON_OBJECT_TYPE_WOULD_BE_VIOLATED ("object", *type*)

9.3.5 OBJECT_DELETE

(1)
```
OBJECT_DELETE (
    origin  : Object_designator,
    link    : Link_designator
)
```

(2) OBJECT_DELETE deletes the composition or existence link *link* of the object *origin*, its reverse link *reverse_link*, and possibly its destination *dest*. More precisely:

(3) - the link *link* of the object *origin* is deleted; the reverse link *reverse_link* of *link* is deleted;

(4) - if *link* is the last existence or composition link to *dest*, then *dest* is deleted.

(5) To *delete* an object X entails the deletion of all components of X except components Y for which there is an incoming external link with the existence property to Y or to an enclosing

object of Y, and the deletion of all links from and to those deleted objects (except designation links to them).

(6) Non-implicit links of components which are not deleted are not affected.

(7) If *origin* or any of its components is opened by one or more processes (see 12.1), the deletion of its contents is postponed until all processes have closed the contents: i.e. the object is no longer accessible but an operation using a contents handle to access its contents is not affected by the deletion until the contents handle is closed.

(8) For each deleted object, if any, the "object_on_volume" link from the volume on which the deleted object resided to the deleted object is also deleted.

(9) Write locks of the default kind are obtained on the deleted objects, if any, and on the deleted links except the deleted "object_on_volume" links; and a read lock of the default mode is obtained on *origin* if the interpretation of *link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

Errors

(10) If the conditions hold for the deletion of *dest*:
  ACCESS_ERRORS (*dest* and its deleted components, ATOMIC, MODIFY, DELETE)

(11) ACCESS_ERRORS (*origin*, ATOMIC, MODIFY, WRITE_LINKS)

(12) If *reverse_link* is not implicit:
  ACCESS_ERRORS (*dest*, ATOMIC, MODIFY, WRITE_LINKS)

(13) If *reverse_link* is implicit:
  ACCESS_ERRORS (*dest*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(14) For each origin X of an implicit incoming external link to a deleted object:
  ACCESS_ERRORS (X, ATOMIC, CHANGE, WRITE_IMPLICIT)

(15) For each atomically stabilizing external link L of a deleted object:
  ACCESS_ERRORS (destination of L, ATOMIC, CHANGE, STABILIZE)

(16) For each compositely stabilizing external link L of a deleted object:
  ACCESS_ERRORS (destination of L, COMPOSITE, CHANGE, STABILIZE)

(17) CATEGORY_IS_BAD (*origin*, *link*, (EXISTENCE, COMPOSITION))

(18) DESTINATION_OBJECT_TYPE_IS_INVALID (*origin*, *link*, *dest*)

(19) LOWER_BOUND_WOULD_BE_VIOLATED (*origin*, *link*)

(20) OBJECT_HAS_EXTERNAL_LINKS_PREVENTING_DELETION (*dest*)

(21) OBJECT_HAS_INTERNAL_LINKS_PREVENTING_DELETION (*dest*)

(22) OBJECT_IS_IN_USE_FOR_DELETE (*dest*)

(23) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*origin*, *link*, DELETE_MODE)

(24) NOTE - OBJECT_DELETE works exactly like LINK_DELETE if *dest* has no direct components.

9.3.6 OBJECT_DELETE_ATTRIBUTE

(1)
```
OBJECT_DELETE_ATTRIBUTE (
    object      : Object_designator,
    attribute   : Attribute_designator
)
```

(2) OBJECT_DELETE_ATTRIBUTE removes the attribute *attribute* from the attributes of the object *object*, if the "attribute_type" object representing the attribute type of *attribute* is no longer in the object base.

(3) A write lock of the default mode is obtained on *object*.

Errors

(4) ACCESS_ERRORS (*object*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(5) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(6) NOTE - It is the responsibility of the user to ensure that the attribute type is no longer in the object base.

9.3.7 OBJECT_GET_ATTRIBUTE

(1)
```
OBJECT_GET_ATTRIBUTE (
    object      : Object_designator,
    attribute   : Attribute_designator
)
    value       : Attribute_value
```

(2) OBJECT_GET_ATTRIBUTE returns the value *value* of the attribute *attribute* of the object *object*.

(3) A read lock of the default mode is obtained on *object*. If *attribute* is a predefined composite time, a read lock of the default mode is also obtained on all components of *object*.

Errors

(4) ACCESS_ERRORS (*object*, ATOMIC, READ, READ_ATTRIBUTES)

(5) If *attribute* is the last composite access time, last composite modification time, or last composite change time:
ACCESS_ERRORS (*object*, COMPOSITE, READ, READ_ATTRIBUTES)

(6) USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*object*, *attribute*, READ_MODE)

9.3.8 OBJECT_GET_PREFERENCE

(1)
```
OBJECT_GET_PREFERENCE (
    object  : Object_designator
)
    key     : [ Text ],
    type    : [ Link_type_nominator ]
```

(2) OBJECT_GET_PREFERENCE returns the preferred link key *key* and preferred link type *type*, if any, of the object *object*.

(3) A read lock of the default mode is obtained on *object*.

Errors

(4) ACCESS_ERRORS (*object*, ATOMIC, READ, READ_ATTRIBUTES)

9.3.9 OBJECT_GET_SEVERAL_ATTRIBUTES

(1)
```
OBJECT_GET_SEVERAL_ATTRIBUTES (
    object      : Object_designator,
    attributes  : Attribute_selection
)
    values      : Attribute_assignments
```

(2) OBJECT_GET_SEVERAL_ATTRIBUTES returns a set of attribute assignments *values* of the object *object*.

(3) The returned set of attributes is determined by *attributes*:

(4) - a set of attribute designators: the set of attributes, as for OBJECT_GET_ATTRIBUTE (*object*, A) for each attribute A of *attributes*;

(5) - VISIBLE_ATTRIBUTE_TYPES: all attributes of *object* visible in the working schema of the calling process and with usage mode including READ_MODE.

(6) A read lock of the default mode is obtained on *object*. If any of the attributes is a predefined composite time, a read lock of the default mode is also obtained on all components of *object*.

Errors

(7) ACCESS_ERRORS (*object*, ATOMIC, READ, READ_ATTRIBUTES)

(8) If *attributes* contains one or more of last composite access time, last composite modification time, or last composite change time, or if *attributes* is VISIBLE_ATTRIBUTE_TYPES and one or more of those attributes are visible in the working schema of the calling process with usage mode READ_MODE:
ACCESS_ERRORS (*object*, COMPOSITE, READ, READ_ATTRIBUTES)

(9) USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*object*, an element of *attributes*, READ_MODE)

9.3.10 OBJECT_GET_TYPE

(1)
```
OBJECT_GET_TYPE (
    object  : Object_designator
)
    type    : Object_type_nominator
```

(2) OBJECT_GET_TYPE returns the type *type* of the object *object*; i.e. the actual type of *object*, whether or not it is visible.

(3) A read lock of the default mode is obtained on *object*.

Errors

(4) CONFIDENTIALITY_WOULD_BE_VIOLATED (*object*)

(5) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (*object*)

(6) OBJECT_IS_ARCHIVED (*object*)

(7) VOLUME_IS_INACCESSIBLE (*object*, ATOMIC)

9.3.11 OBJECT_IS_COMPONENT

(1)
```
OBJECT_IS_COMPONENT (
    object1    : Object_designator,
    object2    : Object_designator
)
    value      : Boolean
```

(2) OBJECT_IS_COMPONENT tests if *object1* is a component of *object2*.

(3) If *object1* is a component of *object2*, *value* is **true**, otherwise it is **false**.

(4) Read locks of the default mode are obtained on *object1* and on *object2*, and on accessed components of *object2*.

Errors

(5) ACCESS_ERRORS (*object2*, COMPONENTS, READ, READ_LINKS)

9.3.12 OBJECT_LIST_LINKS

(1)
```
OBJECT_LIST_LINKS (
    origin     : Object_designator,
    extent     : Link_scope,
    scope      : Object_scope,
    categories : [ Categories ],
    visibility : Link_selection
)
    links      : Link_set_descriptors
```

(2) OBJECT_LIST_LINKS returns in *links* a set of links of the object *origin* and possibly of its components determined by *extent*, *scope*, *categories*, and *visibility*.

(3) *extent* affects the returned set of links as follows:

(4) - INTERNAL_LINKS: only internal links are returned.

(5) - EXTERNAL_LINKS: only external links are returned.

(6) - ALL_LINKS: both internal and external links are returned.

(7) In the lists of links returned, designation links to deleted objects appear only when *extent* is ALL_LINKS or EXTERNAL_LINKS.

(8) *scope* affects the returned set of links as follows:

(9) - ATOMIC: only links of *origin* are returned.

(10) - COMPOSITE: links of *origin* and of all components of *origin* are returned.

(11) *categories* may be omitted if *visibility* is a set of link type nominators, and is ignored in that case if supplied. In other cases only link types with category in the set *categories* are returned.

(12) *visibility* restricts the returned set of links as follows:

(13) - VISIBLE_LINK_TYPES: only links with link type which is visible in the calling process's working schema are returned;

(14) - ALL_LINK_TYPES: all links are returned;

(15) - a set of link type nominators: only links with link type identified by an element of the set are returned.

(16) A read lock of the default mode is obtained on *origin*, and if *scope* is COMPOSITE read locks of the default mode are obtained on all its components.

Errors

(17) ACCESS_ERRORS (*origin*, *scope*, READ, READ_LINKS)

(18) LINK_NAME_IS_TOO_LONG_IN_CURRENT_WORKING_SCHEMA

(19) REFERENCE_CANNOT_BE_ALLOCATED

NOTES

(20) 1 When *scope* is ATOMIC or *origin* has no components, the object designator in each returned link designates the object *origin*. In other cases, the designated objects may include the object *origin* and its components.

(21) 2 Each object designator returned in a link set descriptor of *links* can be used with each link designator of that link set descriptor to retrieve information about the volume on which the object resides, by means of LINK_GET_DESTINATION_VOLUME.

(22) 3 If *scope* is COMPOSITE, links of *origin* and of all components of *origin* are returned. It is possible that the origins of some of the returned links are not accessible from *origin* through paths of visible links.

(23) 4 OBJECT_LIST_LINKS does not prevent a process from seeing data structures which are inconsistent with the visibility restrictions of its working schema.

9.3.13 OBJECT_LIST_VOLUMES

(1)
```
OBJECT_LIST_VOLUMES(
    object      : Object_designator
)
    volumes     : Volume_infos
```

(2) OBJECT_LIST_VOLUMES returns the set of volume identifiers of volumes holding components of *object* (except for any components to which there are composition links from components on unmounted volumes), with an indication of the mounted state of each volume.

(3) A read lock of the default mode is obtained on *object*.

Errors

(4) ACCESS_ERRORS (*object*, COMPOSITE, READ, READ_LINKS)

(5) OBJECT_IS_ARCHIVED (component of *object*)

(6) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (component of *object*, direct outgoing composition link of that component, NAVIGATE_MODE)

(7) NOTE - The note of 9.2.5 applies for each component of *object*.

9.3.14 OBJECT_MOVE

(1)
```
OBJECT_MOVE (
    object              : Object_designator,
    on_same_volume_as   : Object_designator,
    scope               : Object_scope
)
```

(2) OBJECT_MOVE moves *object* to the volume *volume* on which *on_same_volume_as* resides.

(3) If *scope* is ATOMIC:

(4) - If *object* already resides on *volume*, it is not affected.

(5) - Otherwise, *object* is moved to *volume*.

(6) If *scope* is COMPOSITE:

(7) - Each of *object* and the components of *object* which already reside on *volume* are not affected.

(8) - All other of *object* and the components of *object* are moved to *volume*, and the space previously occupied by those components is freed.

(9) The effect of *moving* an object A to a volume V is as follows.

(10) - The attributes and links of A are unchanged, except for the predefined attributes "volume_identifier" which is set to the volume identifier of V, and "last_change_time" and "last_composite_change_time", which are set to the current system time.

(11) - For *object* (if moved) and each moved component, the "object_on_volume" link to it from the volume on which the component was previously residing is deleted, and a new "object_on_volume" link is created to it from *volume*. The created link is keyed by the exact identifier of its destination object.

(12) A write lock of the default mode is obtained on each moved object. An implementation may set a write lock of the default mode on each link to a moved object (except the "object_on_volume" links) if the link is modified by the operation.

Errors

(13) If *scope* is ATOMIC:
ACCESS_ERRORS (*object*, ATOMIC, CHANGE, CONTROL_OBJECT)

(14) If *scope* is COMPOSITE:
ACCESS_ERRORS (*object*, COMPOSITE, READ, READ_LINKS)
ACCESS_ERRORS (*object* and its moved components, ATOMIC, CHANGE, CONTROL_OBJECT)

(15) ACCESS_ERRORS (*on_same_volume_as*, ATOMIC, SYSTEM_ACCESS)

(16) If *object* or some of its components are moved:
OBJECT_IS_IN_USE_FOR_MOVE (*object*)
OBJECT_IS_INACCESSIBLY_ARCHIVED (*object*, *scope*)
OBJECT_IS_LOCKED (*object*, *scope*)
OBJECT_IS_NOT_MOVABLE (*object*, *scope*)
OBJECT_IS_REPLICATED (*object*, *scope*)
TYPE_OF_OBJECT_IS_INVALID (*object*, *scope*)
VOLUME_IS_FULL (volume of *on_same_volume_as*)

(17) The following implementation-dependent errors may be raised for any object X with a link to *object*:
OBJECT_IS_INACCESSIBLY_ARCHIVED (X)
VOLUME_IS_INACCESSIBLE (volume on which X resides)
VOLUME_IS_READ_ONLY (volume on which X resides)

9.3.15 OBJECT_RESET_ATTRIBUTE

(1)
```
OBJECT_RESET_ATTRIBUTE (
    object      : Object_designator,
    attribute   : Attribute_designator
)
```

(2) OBJECT_RESET_ATTRIBUTE resets the attribute *attribute* of the object *object* to its initial value.

(3) A write lock of the default mode is obtained on *object*.

Errors

(4) ACCESS_ERRORS (*object*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(5) USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*object*, *attribute*, WRITE_MODE)

9.3.16 OBJECT_SET_ATTRIBUTE

(1)
```
OBJECT_SET_ATTRIBUTE (
    object      : Object_designator,
    attribute   : Attribute_designator,
    value       : Attribute_value
)
```

(2) OBJECT_SET_ATTRIBUTE assigns the value *value* to the attribute *attribute* of the object *object*.

(3) A write lock of the default mode is obtained on *object*.

Errors

(4) ACCESS_ERRORS (*object*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(5) ENUMERATION_VALUE_IS_OUT_OF_RANGE (*value*, values of *attribute*)

(6) USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*object*, *attribute*, WRITE_MODE)

(7) VALUE_LIMIT_ERRORS (*value*)

(8) The following implementation-dependent error may be raised:
VALUE_TYPE_IS_INVALID (*value*, *object*, *attribute*)

9.3.17 OBJECT_SET_PREFERENCE

(1)
```
OBJECT_SET_PREFERENCE (
    object  : Object_designator,
    type    : [ Link_type_nominator ],
    key     : [ Text ]
)
```

(2) OBJECT_SET_PREFERENCE sets the preferred link type of the object *object* to the link type *type* (if supplied), and preferred link key of *object* to *key* (if supplied).

(3) If both *type* and *key* are supplied, the preferred link type of *object* is set to *type* and the preferred link key of *object* is set to *key*.

(4) If *type* is supplied and *key* is not, the preferred link type of *object* is set to *type* and the preferred link key of *object* is unset.

(5) If *type* is not supplied and *key* is supplied, then the preferred link type of *object* must already be set (else the error condition PREFERRED_LINK_TYPE_IS_UNSET is raised); the preferred link key is set to *key*.

(6) If *key* and *type* are not supplied, the preferred link type and preferred link key of *object* are unset.

(7) A write lock of the default mode is obtained on *object*.

Errors

(8) ACCESS_ERRORS (*object*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(9) CARDINALITY_IS_INVALID (*type*)

(10) LIMIT_WOULD_BE_EXCEEDED (MAX_KEY_SIZE)

(11) LIMIT_WOULD_BE_EXCEEDED (MAX_KEY_VALUE)

(12) LINK_TYPE_IS_UNKNOWN (*type*)

(13) PREFERRED_LINK_KEY_IS_BAD (*key*, *type* or preferred link type of *object* if *type* is not supplied)

(14) If *type* is not supplied and *key* is supplied:
PREFERRED_LINK_TYPE_IS_UNSET (*object*)

9.3.18 OBJECT_SET_SEVERAL_ATTRIBUTES

(1)
```
OBJECT_SET_SEVERAL_ATTRIBUTES (
    object      : Object_designator,
    attributes  : Attribute_assignments
)
```

(2) For each element A of the domain of *attributes*, OBJECT_SET_SEVERAL_ATTRIBUTES sets the value of the attribute A of *object* to *attributes* (A) in the same way as:

(3)
```
OBJECT_SET_ATTRIBUTE (object, A, attributes (A)).
```

(4) A write lock of the default mode is obtained on *object*.

Errors

(5) ACCESS_ERRORS (*object*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(6) For each element A of the domain of *attributes*
ENUMERATION_VALUE_IS_OUT_OF_RANGE (*attributes*(A), values of A)
USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*object*, A, WRITE_MODE)
VALUE_LIMIT_ERRORS (*attributes* (A))

(7) The following implementation-dependent error may be raised for each element A of the domain of *attributes*:
VALUE_TYPE_IS_INVALID (*attributes* (A), *object*, A)

9.3.19 OBJECT_SET_TIME_ATTRIBUTES

(1)
```
OBJECT_SET_TIME_ATTRIBUTES(
    object             : Object_designator,
    last_access        : [ Time ],
    last_modification  : [ Time ],
    scope              : Object_scope
)
```

(2) OBJECT_SET_TIME_ATTRIBUTES sets the time attributes of the object *object* as follows.

(3) If *scope* is ATOMIC:

(4) - the last access time of *object* is set to *last_access* if supplied, otherwise to the current system time;

(5) - the last modification time of *object* is set to *last_modification* if supplied, otherwise to the current system time;

(6) If *scope* is COMPOSITE:

(7) - the last composite access time of *object*, and the last access time of each component of *object*, are set to *last_access* if supplied, otherwise to the current system time;

(8) - the last composite modification time of *object*, and the last modification time of each component of *object*, are set to *last_modification* if supplied, otherwise to the current system time;

(9) A write lock of the default mode is obtained on *object*.

Errors

(10) ACCESS_ERRORS (*object*, *scope*, MODIFY, WRITE_ATTRIBUTES)

(11) If *last_access* or *last_modification* is supplied:
PRIVILEGE_IS_NOT_GRANTED (PCTE_HISTORY)

(12) LIMIT_WOULD_BE_EXCEEDED (MAX_TIME_ATTRIBUTE, MIN_TIME_ATTRIBUTE)

9.3.20 VOLUME_LIST_OBJECTS

(1)
```
VOLUME_LIST_OBJECTS (
    volume      : Volume_designator,
    types       : Object_type_nominators
)
    objects     : Object_designators
```

(2) VOLUME_LIST_OBJECTS returns in *objects* a set of object designators determined by *types*.

(3) An object designator is returned in *objects* for each object which resides on *volume*, whose type in working schema is an element of *types*.

(4) A read lock of the default mode is obtained on *volume*.

Errors

(5) ACCESS_ERRORS (*volume*, ATOMIC, READ, READ_LINKS)

(6) REFERENCE_CANNOT_BE_ALLOCATED

9.4 Version operations

9.4.1 VERSION_ADD_PREDECESSOR

(1)
```
VERSION_ADD_PREDECESSOR (
    version           : Object_designator,
    new_predecessor   : Object_designator
)
```

(2) VERSION_ADD_PREDECESSOR adds *new_predecessor* as a predecessor of *version* in a graph of versions, by creating a "predecessor" link with key the next available natural value from *version* to *new_predecessor*.

(3) Write locks of the default mode are obtained on the new links.

Errors

(4) ACCESS_ERRORS (*new_predecessor*, ATOMIC, CHANGE, STABILIZE)

(5) ACCESS_ERRORS (*version*, ATOMIC, MODIFY, APPEND_LINKS)

(6) MASTER_IS_INACCESSIBLE (some object of the graph of security groups, ATOMIC)

(7) OBJECT_CANNOT_BE_STABILIZED (component of *version*)

(8) PRIVILEGE_IS_NOT_GRANTED (PCTE_HISTORY)

(9) VERSION_GRAPH_IS_INVALID (*version*, *new_predecessor*)

9.4.2 VERSION_IS_CHANGED

(1)
```
VERSION_IS_CHANGED (
    version        : Object_designator,
    predecessor    : Natural
)
    changed        : Boolean
```

(2) VERSION_IS_CHANGED tests whether *version* has been changed since being created as a new version of its predecessor *predecessor* by comparing the values of the last composite modification time for *version* and *predecessor*. If it has been changed, i.e. the last composite modification times are different, then *changed* is **true**, otherwise it is **false**.

(3) Read locks of the default mode are obtained on *version*, on all components of *version*, and on *predecessor*.

Errors

(4) ACCESS_ERRORS (*predecessor*, COMPONENTS, READ, (READ_LINKS, READ_ATTRIBUTES, READ_CONTENTS))

(5) ACCESS_ERRORS (*version*, COMPONENTS, READ, (READ_LINKS, READ_ATTRIBUTES, READ_CONTENTS))

(6) ACCESS_ERRORS (*version*, COMPOSITE, CHANGE)

(7) LINK_DOES_NOT_EXIST (*version*, "predecessor" link with key *predecessor*)

9.4.3 VERSION_REMOVE

(1)
```
VERSION_REMOVE (
    version     : Object_designator
)
```

(2) VERSION_REMOVE removes *version* from its graph of versions.

(3) For X as *version* and each component of *version*:

(4) - If X has external successors and predecessors then a "predecessor" link is created from each successor of X to each predecessor of X, and all the "predecessor" links to and from X are deleted.

(5) - If X has only external successors: then all the "predecessor" links to X are deleted.

(6) - If X has only external predecessors then all the "predecessor" links from X are deleted.

(7) Write locks of the default mode are obtained on the deleted links and on the created links.

Errors

(8) ACCESS_ERRORS (*version*, COMPOSITE, MODIFY, (WRITE_LINKS, WRITE_IMPLICIT))

(9) For each component X of *version* which has more than one successor:
ACCESS_ERRORS (predecessor of X, COMPOSITE, CHANGE, STABILIZE)

(10) ACCESS_ERRORS (predecessor of *version*, ATOMIC, CHANGE, (WRITE_IMPLICIT, APPEND_IMPLICIT))

(11) ACCESS_ERRORS (successor of *version*, ATOMIC, CHANGE, (WRITE_LINKS, APPEND_LINKS))

(12) ACCESS_ERRORS (*version* and its deleted components, ATOMIC, MODIFY, DELETE)

(13) OBJECT_HAS_EXTERNAL_LINKS_PREVENTING_DELETION (*version*)

(14) OBJECT_HAS_INTERNAL_LINKS_PREVENTING_DELETION (*version*)

(15) OBJECT_IS_IN_USE_FOR_DELETE (*version*)

(16) PRIVILEGE_IS_NOT_GRANTED (PCTE_HISTORY)

(17) VERSION_IS_REQUIRED (*version*, COMPOSITE)

9.4.4 VERSION_REMOVE_PREDECESSOR

(1)
```
VERSION_REMOVE_PREDECESSOR (
    version        : Object_designator,
    predecessor    : Object_designator
)
```

(2) VERSION_REMOVE_PREDECESSOR removes the object *predecessor* as a predecessor of *version* in the graph of versions, by deleting the "predecessor" link from *version* to *predecessor* and its reverse "successor" link.

(3) Write locks of the default mode are obtained on the deleted links.

Errors

(4) ACCESS_ERRORS (*predecessor*, ATOMIC, CHANGE, STABILIZE)

(5) If *predecessor* is to be deleted:
ACCESS_ERRORS (*predecessor* and its deleted components, ATOMIC, MODIFY, DELETE)

(6) ACCESS_ERRORS (*predecessor*, COMPOSITE, CHANGE)

(7) ACCESS_ERRORS (*version*, ATOMIC, MODIFY, WRITE_LINKS)

(8) If there is no "predecessor" link between *version* and *predecessor*:
LINK_DOES_NOT_EXIST (*version*, "predecessor" link)

(9) PRIVILEGE_IS_NOT_GRANTED (PCTE_HISTORY)

9.4.5 VERSION_REVISE

(1)
```
VERSION_REVISE (
    version            : Object_designator,
    new_origin         : Object_designator,
    new_link           : Link_designator,
    on_same_volume_as  : [ Object_designator ],
    access_mask        : Atomic_access_rights
)
    new_version        : Object_designator
```

(2) VERSION_REVISE creates a new updatable version (a *revision*) of *version*. That is:

(3) - A copy *new_version* of *version* is created in the same way as

  OBJECT_COPY (*version*, *new_origin*, *new_link*, **nil**, *on_same_volume_as*, *access_mask*)

  where *reverse_key* is not supplied.

(4) - "predecessor" links with key 1 (one) are created from *new_version* and each of its components to *version* and each of its corresponding components. As a consequence, *new_version* and each of its components becomes a new successor of *version* and each of its corresponding components (i.e. a reverse "successor" link with a system-generated key is created).

(5) Since the "predecessor" links are compositely stabilizing, *version* and its components are made stable.

(6) *access_mask* is used in conjunction with the default atomic ACL and default object owner of the calling process to specify the atomic ACL and the composite ACL of *new_version*. *new_version* and its components have the same integrity and confidentiality labels as the objects they have been copied from.

(7) If *on_same_volume_as* is supplied, the *new_version* resides on the same volume as *on_same_volume_as*. Otherwise, *new_version* resides on the same volume as *version* and each component of *new_version* resides on the same volume as the corresponding component of *version*.

(8) If a replicated component is revised, its revision is not replicated.

(9) Read locks of the default mode are obtained on all components of *version* and write locks of the default mode are obtained on the new links and components of *new_version*.

(10) A read lock of the default mode is obtained on *new_origin* if the interpretation of *new_link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

(11) If *version* is an accounting log, its contents is preserved.

Errors

(12) ACCESS_ERRORS (*new_origin*, ATOMIC, MODIFY, APPEND_LINKS)

(13) ACCESS_ERRORS (*version*, COMPOSITE, CHANGE, APPEND_IMPLICIT)

(14) ACCESS_ERRORS (*version*, COMPOSITE, READ, (READ_LINKS, READ_ATTRIBUTES, READ_CONTENTS))

(15) ACCESS_ERRORS (*version*, COMPOSITE, CHANGE, STABILIZE)

(16) If the reverse of *new_link* is compositely stabilizing:
  ACCESS_ERRORS (*new_origin*, COMPOSITE, CHANGE, STABILIZE)

(17) ACCESS_ERRORS (*on_same_volume_as*, ATOMIC, SYSTEM_ACCESS)

(18) For each destination X of a duplicable external link L of a duplicated component:

(19) ACCESS_ERRORS (X, ATOMIC, CHANGE, APPEND_IMPLICIT)

(20) If L is atomically stabilizing:
ACCESS_ERRORS (X, ATOMIC, CHANGE, STABILIZE)

(21) If L is compositely stabilizing:
ACCESS_ERRORS (X, COMPOSITE, CHANGE, STABILIZE)

(22) CATEGORY_IS_BAD (*new_origin*, *new_link*, (COMPOSITION, EXISTENCE))

(23) CATEGORY_IS_BAD (destination of *new_link*, reverse of *new_link*, IMPLICIT)

(24) If *version* is a component of itself, and *new_link* is a composition link:
COMPONENT_ADDITION_ERRORS (*new_version*, *new_link*)

(25) CONTROL_WOULD_NOT_BE_GRANTED (*new_version*)

(26) EXTERNAL_LINK_IS_BAD (*version*, COMPOSITE)

(27) LINK_EXISTS (*new_origin*, *new_link*)

(28) OBJECT_CANNOT_BE_STABILIZED (component of *version*)

(29) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL (*new_object*)

(30) If *version* is not a component of itself, *new_link* is a composition link and *new_origin* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_version*)

(31) REFERENCE_CANNOT_BE_ALLOCATED

(32) TYPE_OF_OBJECT_IS_INVALID (*version*, COMPOSITE)

(33) UPPER_BOUND_WOULD_BE_VIOLATED (*new_origin*, *new_link*)

(34) USAGE_MODE_ON_OBJECT_TYPE_WOULD_BE_VIOLATED ("object", type of *version*)

(35) VALUE_LIMIT_ERRORS (*reverse_key*)

(36) VOLUME_IS_FULL (volume on which *on_same_volume_as* resides)

9.4.6 VERSION_SNAPSHOT

(1)
```
VERSION_SNAPSHOT (
    version                : Object_designator,
    new_link_and_origin    : [ Link_descriptor ],
    on_same_volume_as      : [ Object_designator ],
    access_mask            : Atomic_access_rights
)
    new_version            : Object_designator
```

(2) VERSION_SNAPSHOT creates a new stable version (a *snapshot*) of *version*. That is, if *new_origin* and *new_link* are the object designator and link designator respectively of *new_link_and_origin*:

(3) - A copy *new_version* of *version* is created in the same way as

OBJECT_COPY (*version*, *new_origin*, *new_link*, **nil**, *on_same_volume_as*, *access_mask*).

where *reverse_link* is not supplied; except that if *new_link_and_origin* is not supplied, then no new link of *new_origin* is created

(4) - The set of predecessors of each component of *new_version* is the set of predecessors of its corresponding component of *version*. Then a "predecessor" link with key 1 (one) is created

in such a way that each component of *new_version* becomes the first predecessor of its corresponding component of *version*.

(5) All the granted write and append discretionary access rights are suppressed (i.e. set to UNDEFINED) for all the components of *new_version*.

(6) The components of *version* are still updatable. *access_mask* is used in conjunction with the default atomic ACL and default object owner of the calling process to specify the atomic ACL and the composite ACL of the created objects.

(7) The components of *new_version* are stabilized.

(8) The created objects have the same integrity and confidentiality labels as the objects they have been copied from.

(9) If *on_same_volume_as* is supplied, the *new_version* resides on the same volume as *on_same_volume_as*. Otherwise, *new_version* resides on the same volume as *version* and each component of *new_version* resides on the same volume as the corresponding component of *version*.

(10) If a component of *version* is replicated, its snapshot is not replicated.

(11) The predecessor links are created even if their origins are stable.

(12) Read locks of the default mode are obtained on all components of *version* to be copied and write locks of the default mode are obtained on the new links and components of *new_version*. A read lock of the default mode is obtained on *new_origin* if the interpretation of *new_link* implies the evaluation of any '+' or '++' key attribute values (see 23.1.2.7).

(13) If *version* is an accounting log, its contents is preserved.

Errors

(14) ACCESS_ERRORS (*new_origin*, ATOMIC, MODIFY, APPEND_LINKS)

(15) ACCESS_ERRORS (*version*, COMPOSITE, MODIFY, APPEND_LINKS)

(16) If *version* or any component of *version* already has a predecessor:
ACCESS_ERRORS (*version*, COMPOSITE, MODIFY, WRITE_LINKS)

(17) ACCESS_ERRORS (*version*, COMPOSITE, READ, (READ_LINKS, READ_ATTRIBUTES, READ_CONTENTS))

(18) ACCESS_ERRORS (*new_version*, COMPOSITE, CHANGE, STABILIZE)

(19) If *version* has a predecessor, than for each predecessor X of each component of *version*:
ACCESS_ERRORS (X, ATOMIC, CHANGE, (APPEND_IMPLICIT, WRITE_IMPLICIT))

(20) If *new_link* is provided and its reverse is compositely stabilizing:
ACCESS_ERRORS (*new_origin*, COMPOSITE, CHANGE, STABILIZE)

(21) ACCESS_ERRORS (*on_same_volume_as*, ATOMIC, SYSTEM_ACCESS)

(22) For each destination X of a duplicable external link L of a duplicated component:

(23) ACCESS_ERRORS (X, ATOMIC, CHANGE, APPEND_IMPLICIT)

(24) If L is atomically stabilizing:
ACCESS_ERRORS (X, ATOMIC, CHANGE, STABILIZE)

(25) If L is compositely stabilizing:
ACCESS_ERRORS (X, COMPOSITE, CHANGE, STABILIZE)

(26) CATEGORY_IS_BAD (*new_origin*, *new_link*, (COMPOSITION, EXISTENCE, REFERENCE, DESIGNATION))

(27) If *new_link* has a reverse link:
CATEGORY_IS_BAD (destination of *new_link*, reverse of *new_link*, IMPLICIT)

(28) If *version* is a component of itself, and *new_link* is a composition link:
COMPONENT_ADDITION_ERRORS (*new_version*, *new_link*)

(29) CONTROL_WOULD_NOT_BE_GRANTED (*new_version*)

(30) EXTERNAL_LINK_IS_BAD (*version*, COMPOSITE)

(31) REFERENCE_CANNOT_BE_ALLOCATED

(32) LINK_EXISTS (*new_origin*, *new_link*)

(33) OBJECT_CANNOT_BE_STABILIZED (component of *version*)

(34) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL (*new_object*)

(35) If *version* is not a component of itself, *new_link* is a composition link and *new_origin* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_version*)

(36) TYPE_OF_OBJECT_IS_INVALID (*version*, COMPOSITE)

(37) UPPER_BOUND_WOULD_BE_VIOLATED (*new_origin*, *new_link*)

(38) USAGE_MODE_ON_OBJECT_TYPE_WOULD_BE_VIOLATED ("object", type of *version*)

(39) VALUE_LIMIT_ERRORS (*reverse_key*)

(40) VOLUME_IS_FULL (volume on which *on_same_volume_as* resides)

9.4.7 VERSION_TEST_ANCESTRY

(1)
```
VERSION_TEST_ANCESTRY (
    version1    : Object_designator,
    version2    : Object_designator
)
    ancestry    : Version_relation
```

(2) VERSION_TEST_ANCESTRY tests the ancestry of the objects *version1* and *version2* in their graphs of versions. That is, it returns in *ancestry*:

(3) - ANCESTOR_VSN if there exists a series of "predecessor" links from *version2* to *version1;*

(4) - DESCENDANT_VSN if there exists a series of "predecessor" links from *version1* to *version2;*

(5) - SAME_VSN if *version1* is the same object as *version2;*

(6) - RELATED_VSN if there exist an object X which is neither *version1* nor *version2*, a series of "predecessor" links from *version1* to X, and a series of "predecessor" links from *version2* to X;

(7) - UNRELATED_VSN otherwise.

(8) Read locks of the default mode are obtained on *version1* and on *version2* and on all the origins and destinations of the links in the series of links.

Errors

(9) ACCESS_ERRORS (*version1*, ATOMIC, READ, READ_LINKS)

(10) ACCESS_ERRORS (element of version graph of *version1*, ATOMIC, READ, READ_LINKS)

(11) ACCESS_ERRORS (*version2*, ATOMIC, READ, READ_LINKS)

(12) ACCESS_ERRORS (element of version graph of *version2*, ATOMIC, READ, READ_LINKS)

9.4.8 VERSION_TEST_DESCENT

(1)
```
VERSION_TEST_DESCENT (
    version1    : Object_designator,
    version2    : Object_designator
)
    descent     : Version_relation
```

(2) VERSION_TEST_DESCENT tests the descent of the objects *version1* and *version2* in their graphs of versions. That is, it returns:

(3) - ANCESTOR_VSN if there exists a series of "successor" links from *version1* to *version2*;

(4) - DESCENDANT_VSN if there exists a series of "successor" links from *version2* to *version1*;

(5) - SAME_VSN if *version1* is the same object as *version2*;

(6) - RELATED_VSN if there exist an object X which is neither *version1* nor *version2*, a series of "successor" links from *version1* to X, and a series of "successor" links from *version2* to X.

(7) - UNRELATED_VSN otherwise.

(8) Read locks of the default mode are obtained on *version1* and on *version2* and on the all the origins and destinations of the links in the series of links.

Errors

(9) ACCESS_ERRORS (*version1*, ATOMIC, READ, READ_LINKS)

(10) ACCESS_ERRORS (element of version graph of *version1*, ATOMIC, READ, READ_LINKS)

(11) ACCESS_ERRORS (*version2*, ATOMIC, READ, READ_LINKS)

(12) ACCESS_ERRORS (element of version graph of *version2*, ATOMIC, READ, READ_LINKS)

10 Schema management

10.1 Schema management concepts

10.1.1 Schema definition sets and the SDS directory

(1) **sds** metasds:

(2) **import object type** system-object, system-process, system-common_root;

(3) **import attribute type** system-number, system-system_key;

(4) type_identifier: (**read**) **string**;

(5)
```
sds_directory: child type of object with
link
    known_sds: (navigate) non_duplicated existence link (sds_name: string) to sds;
    schemas_of: (navigate) implicit link to common_root reverse schemas;
end sds_directory;
```

(6)
```
sds: child type of object with
link
    named_definition: (navigate) reference link (local_name: string) to type_in_sds
        reverse named_in_sds;
    in_working_schema_of: (navigate) non_duplicated designation link (number) to
        process;
component
    definition: (navigate) exclusive composition link (type_identifier) to type_in_sds
        reverse in_sds;
end sds;
```

(7)
```
extend object type common_root with
link
    schemas: (navigate) existence link to sds_directory reverse schemas_of;
end common_root;
```

(8)
```
end metasds;
```

(9) The SDS directory is an administrative object (see 9.1.2).

(10) The "sds" components of the SDS directory represent the known SDSs of the PCTE installation (see 8.4):

(11) - The "sds_name" key of the "known_sds" link from the SDS directory represents the SDS name of the SDS.

(12) - The definition components of an SDS represent the types in SDS of the SDS (see 8.3). The key of the "named_definition" link is the local name of the type in SDS. The destinations of the "named_definition" links are a subset of the SDS object components. The destinations of the "in_working_schema" links are the known processes which are neither ready nor terminated and have included the SDS in their working schemas.

(13) The destinations of the "in_working_schema" links are the known non-terminated processes which have included the SDS in their working schemas (see 8.5). An SDS and its types in SDS cannot be modified by using the operations defined in 10.2 while the SDS is included in the working schema of such a process.

(14) NOTE - The predefined SDSs 'system', 'metasds', 'discretionary_security', 'mandatory_security', 'auditing', and 'accounting' are protected against modification, and so they cannot be extended directly. However, the predefined types can be imported into other SDSs and then extended, thus achieving the same effect. See 20.1.8.1.

10.1.2 Types

(1)
```
sds metasds:
```

(2)
```
type: (protected) child type of object with
attribute
    type_identifier;
link
    has_type_in_sds: (navigate) implicit link (system_key) to type_in_sds reverse of_type;
end type;
```

(3)
```
type_in_sds: (protected) child type of object with
attribute
    annotation: string;
    creation_or_importation_time: (read) time;
link
    in_sds: (navigate) implicit link to sds reverse definition;
    of_type: (navigate) existence link to type reverse has_type_in_sds;
    named_in_sds: (navigate) implicit link to sds reverse named_definition;
end type_in_sds;
```

(4)
```
usage_mode: (read) natural;
```

(5)
```
export_mode: (read) natural;
```

(6)
```
maximum_usage_mode: (read) natural;
```

(7)
```
end metasds;
```

(8) A "type" object represents a type (see 8.3):

(9) - The "type_identifier" attribute represents the type identifier.

(10) - The destinations of the "has_type_in_sds" links represent types in SDS associated with this type.

(11) Further attribute types and link types are particular to the object types "object_type" (see 10.1.3), "attribute_type" (see 10.1.4), "link_type" (see 10.1.5), and "enumeral_type" (see 10.1.6).

(12) A "type_in_sds" object represents a type in SDS (see 8.4):

(13) - The destination of the "in_sds" link represents the SDS to which the type in SDS belongs.

(14) - The destination of the "named_in_sds" link represents the SDS in which the type in SDS has a local name. It is the same as the destination of the "in_sds" link.

(15) - The destination of the "of_type" link represents the type associated with the type in SDS.

(16) - The usage mode, export mode, and maximum usage mode represent the definition modes of the type in SDS. A set of definition mode values is represented as the sum of the powers of 2 representing its elements as follows:

. CREATE	: 1
. DELETE	: 2
. READ	: 4
. WRITE	: 8
. NAVIGATE	: 16

(17) - The annotation is the complete name of the type when it is created, but may be changed by the user.

(18) - The creation or importation time is the system time when the type in SDS was created or imported into the SDS.

(19) Further attribute types and link types are particular to the object types "object_type_in_sds" (see 10.1.3), "attribute_type_in_sds" (see 10.1.4), "link_type_in_sds" (see 10.1.5), and "enumeration_type_in_sds" (see 10.1.6).

10.1.3 Object types

(1)
```
sds metasds:
```

(2)
```
object_type: (protected) child type of type with
attribute
    contents_type: (read) enumeration (FILE_TYPE, PIPE_TYPE, DEVICE_TYPE,
        AUDIT_FILE_TYPE, ACCOUNTING_LOG_TYPE, NO_CONTENTS_TYPE) :=
        NO_CONTENTS_TYPE;
link
    parent_type: (navigate) reference link (number) to object_type reverse child_type;
    child_type: (navigate) implicit link (system_key) to object_type reverse parent_type;
end object_type;
```

(3)
```
object_type_in_sds: (protected) child type of type_in_sds with
attribute
    usage_mode;
    export_mode;
    maximum_usage_mode;
link
    in_attribute_set: (navigate) reference link (number) to attribute_type_in_sds reverse
        is_attribute_of;
    in_link_set: (navigate) reference link (number) to link_type_in_sds reverse is_link_of;
    is_destination_of: (navigate) reference link (number) to link_type_in_sds reverse
        in_destination_set;
end object_type_in_sds;
```

(4)
```
end metasds;
```

(5) An "object_type" object represents an object type (see 8.3.1):

(6) - The destinations of the "parent_type" links represent the parent types of the object type.

(7) - The destinations of the "child_type" links represent the child types of the object type.

(8) An "object_type_in_sds" object represents an object type in SDS (see 8.4.1):

(9) - The destinations of the "in_attribute_set" links represent the direct attribute types in SDS of the object type in SDS.

(10) - The destinations of the "in_link_set" links represent the direct outgoing link types in SDS of the object type in SDS. The component object types of the object type in SDS are represented by the destinations of the "in_destination_set" links of the destinations of the "in_link_set" links with category COMPOSITION.

(11) - The destinations of the "is_destination_of" links represent the link types in SDS of which this object type is a destination object type.

10.1.4 Attribute types

(1)
```
sds metasds:
```

(2)
```
duplication: (read) enumeration (DUPLICATED, NON_DUPLICATED):= DUPLICATED;
```

(3)
```
key_attribute_of: (navigate) implicit link (system_key) to link_type reverse key_attribute;
```

(4)
```
attribute_type: (protected) child type of type with
attribute
    duplication;
end attribute_type;
```

(5)
```
string_attribute_type: (protected) child type of attribute_type with
attribute
    string_initial_value: (read) string;
link
    key_attribute_of;
end string_attribute_type;
```

(6)
```
integer_attribute_type: (protected) child type of attribute_type with
attribute
    integer_initial_value: (read) integer;
end integer_attribute_type;
```

(7)
```
natural_attribute_type: (protected) child type of attribute_type with
attribute
    natural_initial_value: (read) natural;
link
    key_attribute_of;
end natural_attribute_type;
```

(8)
```
float_attribute_type: (protected) child type of attribute_type with
attribute
    float_initial_value: (read) float;
end float_attribute_type;
```

(9)
```
boolean_attribute_type: (protected) child type of attribute_type with
attribute
    boolean_initial_value: (read) boolean;
end boolean_attribute_type;
```

(10)
```
time_attribute_type: (protected) child type of attribute_type with
attribute
    time_initial_value: (read) time;
end time_attribute_type;
```

(11)
```
enumeration_attribute_type: (protected) child type of attribute_type with
attribute
    initial_value_position: (read) natural;
component
    enumeral: (navigate) composition link [1 .. ] (position: natural) to enumeral_type
        reverse enumeral_of;
end enumeration_attribute_type;
```

(12)
```
attribute_type_in_sds: (protected) child type of type_in_sds with
attribute
    usage_mode;
    export_mode;
    maximum_usage_mode;
link
    is_attribute_of: (navigate) reference link (number) to object_type_in_sds,
        link_type_in_sds reverse in_attribute_set;
end attribute_type_in_sds;
```

(13)
```
end metasds;
```

(14) "Attribute_type" objects represent attribute types (see 8.3.2). They are divided into child types according to value type.

(15) - The initial value attribute represents the initial value of the attribute type. For string, integer, natural, float, boolean, and time attribute types, it is an actual value of the value type. For an enumeration attribute type, it is a non-negative integer defining the position of the initial value within the enumeration type.

(16) - For a natural or a string attribute type, the destinations of the "key_attribute_of" links represent the link types for which this attribute type is a key attribute type.

(17) - For an enumeration attribute type, the destinations of the "enumeral" links represent the enumeral types of the value type. The "position" key attribute represents the ordering of the enumeral types: it must take successive values 0, 1, 2, 3, ...

(18) An "attribute_type_in_sds" object represents an attribute type in SDS (see 8.4.2):

(19) - The destinations of the "in_attribute_set" links represent the object types in SDS and link types in SDS for which the attribute type is a direct attribute type.

10.1.5 Link types

(1)
```
sds metasds:
```

(2)
```
link_type : (protected) child type of type with
attribute
    category: (read) enumeration (COMPOSITION, EXISTENCE, REFERENCE, IMPLICIT,
        DESIGNATION) := COMPOSITION;
    lower_bound: (read) natural := 0;
    upper_bound: (read) natural := MAX_NATURAL_ATTRIBUTE;
    stability: (read) enumeration (ATOMIC_STABLE, COMPOSITE_STABLE, NON_STABLE)
        := NON_STABLE;
    exclusiveness: (read) enumeration (SHARABLE, EXCLUSIVE) := SHARABLE;
    duplication;
link
    reverse: (navigate) reference link to link_type;
    key_attribute: (navigate) reference link (key_number: natural) to string_attribute_type,
        natural_attribute_type reverse key_attribute_of;
end link_type;
```

(3)
```
link_type_in_sds: child type of type_in_sds with
attribute
    usage_mode;
    export_mode;
    maximum_usage_mode;
link
    in_attribute_set;
    is_link_of: (navigate) reference link (number) to object_type_in_sds reverse
        in_link_set;
    in_destination_set: (navigate) reference link (number) to object_type_in_sds reverse
        is_destination_of;
end link_type_in_sds;
```

(4)
```
end metasds;
```

(5) A "link_type" object represents a link type (see 8.3.3):

(6) - The "category" attribute represents the category of the link type.

(7) - The "lower_bound" and "upper_bound" attributes represent the lower bound and upper bound, respectively, of the link type. For MAX_NATURAL_ATTRIBUTE see 24.1.

(8) - The "stability" attribute represents the stability of the link type.

(9) - The "exclusiveness" attribute represents the exclusiveness of the link type.

(10) - The "duplication" attribute represents the duplication property of the link type.

(11) - The destination of the "reverse" link represents the reverse link type of the link type.

(12) - The destinations of the "key_attribute" links represent the key attribute types of the link type.

(13) A "link_type_in_sds" object represents a link type in SDS (see 8.4.3):

(14) - The destinations of the "in_attribute_set" links represent the non-key attribute types of the link type in SDS.

(15) - The destinations of the "is_link_of" links represent the origin object types in SDS of the link type in SDS.

(16) - The destinations of the "in_destination_set" links represent the destination object types in SDS of the link type in SDS.

10.1.6 Enumeral types

(1)
```
sds metasds:
```

(2)
```
enumeral_type: (protected) child type of type with
link
    enumeral_of: (navigate) implicit link (system_key) to enumeration_attribute_type
        reverse enumeral;
end enumeral_type;
```

(3)
```
enumeral_type_in_sds: (protected) child type of type_in_sds with
attribute
    image: (read) string;
end enumeral_type_in_sds;
```

(4)
```
end metasds;
```

(5) An "enumeral_type" object represents an enumeral type (see 8.3.4).

(6) The destinations of the "enumeral_of" links represent the attribute types of which the enumeral type is a possible value.

(7) An "enumeral_type_in_sds" object represents an enumeral type in SDS (see 8.4.4); the "image" attribute represents the image of the enumeral type in SDS.

10.1.7 Datatypes for schema management

(1)
```
Enumeration_values = seq1 of Enumeral_type_nominator_in_sds
```

(2)
```
Key_types_in_sds = seq of Attribute_type_nominator_in_sds
```

(3)
```
Attribute_scan_kind = OBJECT | OBJECT_ALL | LINK_KEY | LINK_NON_KEY
```

(4)
```
Link_scan_kind = ORIGIN | ORIGIN_ALL | DESTINATION | DESTINATION_ALL | KEY |
NON_KEY
```

(5)
```
Object_scan_kind = CHILD | DESCENDANT | PARENT | ANCESTOR | ATTRIBUTE |
ATTRIBUTE_ALL | LINK_ORIGIN | LINK_ORIGIN_ALL | LINK_DESTINATION |
LINK_DESTINATION_ALL
```

(6) These datatypes are used as parameter and result types of operations defined in 10.2, 10.3, and 10.4.

10.2 SDS update operations

10.2.1 SDS_ADD_DESTINATION

(1)
```
SDS_ADD_DESTINATION (
    sds          : Sds_designator,
    link_type    : Link_type_nominator_in_sds,
    object_type  : Object_type_nominator_in_sds
)
```

(2) SDS_ADD_DESTINATION extends the set of destination object types of the link type in SDS *link_type_in_sds* associated with the link type *link_type* in the SDS *sds* to include the object type in SDS *object_type_in_sds* associated with the object type *object_type* in *sds*.

(3) If *link_type* has a reverse link type *reverse*, then *reverse* is applied to *object_type*.

(4) An "in_destination_set" link from *link_type_in_sds* to *object_type_in_sds* and its reverse "is_destination_of" link are created.

(5) If *link_type* has a reverse link type *reverse* then an "in_link_set" link from *object_type_in_sds* to the "link_type_in_sds" object *reverse_link_type_in_sds* associated with *reverse* in *sds*, and its reverse "is_link_of" link, are created.

(6) Write locks of the default mode are obtained on the created links.

Errors

(7) ACCESS_ERRORS (*object_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(8) ACCESS_ERRORS (*link_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(9) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(10) ACCESS_ERRORS (*reverse_link_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(11) OBJECT_TYPE_IS_ALREADY_IN_DESTINATION_SET (*link_type*, *object_type*)

(12) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(13) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(14) SDS_IS_PREDEFINED (*sds*)

(15) SDS_IS_UNKNOWN (*sds*)

(16) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *link_type*)

(17) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *object_type*)

10.2.2 SDS_APPLY_ATTRIBUTE_TYPE

(1)
```
SDS_APPLY_ATTRIBUTE_TYPE (
    sds             : Sds_designator,
    attribute_type  : Attribute_type_nominator_in_sds,
    type            : Object_type_nominator_in_sds | Link_type_nominator_in_sds
)
```

(2) SDS_APPLY_ATTRIBUTE_TYPE extends the object type or link type *type* by the application of the attribute type *attribute_type* in the SDS *sds*.

(3) An "in_attribute_set" link and its reverse "is_attribute_of" link are created between the type in SDS *type_in_sds* associated with *type* in *sds* and the attribute type in SDS *attribute_type_in_sds* associated with *attribute_type* in *sds*.

(4) Write locks of the default mode are obtained on the created links.

Errors

(5) ACCESS_ERRORS (*type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*attribute_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(7) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(8) KEY_ATTRIBUTE_TYPE_APPLY_IS_FORBIDDEN (*attribute_type*)

(9) LINK_TYPE_CATEGORY_IS_BAD (*link_type*, (COMPOSITION, EXISTENCE, REFERENCE, DESIGNATION))

(10) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(11) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(12) SDS_IS_PREDEFINED (*sds*)

(13) SDS_IS_UNKNOWN (*sds*)

(14) TYPE_CANNOT_BE_APPLIED_TO_LINK_TYPE (*type*, *attribute_type*)

(15) TYPE_IS_ALREADY_APPLIED (*sds*, *attribute_type*, *type*)

(16) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *attribute_type*)

(17) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.2.3 SDS_APPLY_LINK_TYPE

(1)
```
SDS_APPLY_LINK_TYPE (
    sds             : Sds_designator,
    link_type       : Link_type_nominator_in_sds,
    object_type     : Object_type_nominator_in_sds
)
```

(2) SDS_APPLY_LINK_TYPE extends the object type *object_type* in the SDS *sds* by the application of the link type *link_type*. If *link_type* has a reverse link type *reverse* then the destination set of *reverse* is extended to include the object type *object_type*.

(3) An "in_link_set" link from the object type in SDS *object_type_in_sds* associated with *object_type* in *sds* to the link type in SDS *link_type_in_sds* associated with *link_type* in *sds*, and its reverse "is_link_of" link, are created. If *link_type* has a reverse link type *reverse*, an "in_destination_set" link from the link type in SDS associated with *reverse* in *sds* to *object_type_in_sds*, and its reverse "is_destination_of" link, are created.

(4) Write locks of the default mode are obtained on the created links.

Errors

(5) ACCESS_ERRORS (*link_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*object_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(7) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(8) ACCESS_ERRORS (reverse of *link_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(9) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(10) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(11) SDS_IS_PREDEFINED (*sds*)

(12) SDS_IS_UNKNOWN (*sds*)

(13) TYPE_IS_ALREADY_APPLIED (*sds*, *link_type*, *object_type*)

(14) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *object_type*)

(15) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *link_type*)

10.2.4 SDS_CREATE_BOOLEAN_ATTRIBUTE_TYPE

(1)
```
SDS_CREATE_BOOLEAN_ATTRIBUTE_TYPE (
    sds             : Sds_designator,
    local_name      : [ Name ],
    initial_value   : [ Boolean ],
    duplication     : Duplication
)
    new_type        : Attribute_type_nominator_in_sds
```

(2) SDS_CREATE_BOOLEAN_ATTRIBUTE_TYPE creates a new boolean attribute type *new_type*, and its associated attribute type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(4) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(5) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation. The duplication of *new_type* is set to *duplication*. The boolean initial value of *new_type* is set to *initial_value* if supplied, and otherwise to **false**.

(6) The three definition mode attributes of *new_type_in_sds* are set to 12, representing READ_MODE and WRITE_MODE, and its creation or importation time is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(7) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(8) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(9) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

Errors

(10) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(11) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(12) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_type_in_sds*)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(14) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(15) SDS_IS_PREDEFINED (*sds*)

(16) SDS_IS_UNKNOWN (*sds*)

(17) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(18) TYPE_NAME_IS_INVALID (*local_name*)

10.2.5 SDS_CREATE_DESIGNATION_LINK_TYPE

(1)
```
SDS_CREATE_DESIGNATION_LINK_TYPE (
    sds             : Sds_designator,
    local_name      : [ Name ],
    lower_bound     : [ Natural ],
    upper_bound     : [ Natural ],
    duplication     : Duplication,
    key_types       : Key_types_in_sds
)
    new_type        : Link_type_nominator_in_sds
```

(2) SDS_CREATE_DESIGNATION_LINK_TYPE creates a new designation link type *new_type* and its associated link type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(4) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(5) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation.

(6) The three definition mode attributes of *new_type_in_sds* are set to 19, representing CREATE_MODE, DELETE_MODE, and NAVIGATE_MODE, and its creation or importation time is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(7) The lower bound, upper bound (if provided), and duplication of *new_type* are set from the parameters of the same names. If *new_type* is of cardinality many, for each attribute type of *key_types*, a "key_attribute" link is created from *new_type* to that attribute type. The keys of these links correspond to the order of the key attribute types in *key_types* starting at 0 and incremented by 1. The category of *new_type* is set to DESIGNATION.

(8) The sets of origin object types in SDS and destination object types in SDS of *new_type* are initially empty.

(9) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(10) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(11) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

Errors

(12) ACCESS_ERRORS (element of *key_types*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(13) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(14) KEY_TYPE_IS_BAD (element of *key_types*)

(15) KEY_TYPES_ARE_MULTIPLE (*key_types*)

(16) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(17) LINK_TYPE_PROPERTIES_ARE_INCONSISTENT (DESIGNATION, *lower_bound*, *upper_bound*, SHARABLE, NON_STABLE, *duplication*)

(18) LINK_TYPE_PROPERTIES_AND_KEY_TYPES_ARE_INCONSISTENT (DESIGNATION, *lower_bound*, *upper_bound*, SHARABLE, NON_STABLE, *duplication*, *key_types*)

(19) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_type_in_sds*)

(20) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(21) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(22) SDS_IS_PREDEFINED (*sds*)

(23) SDS_IS_UNKNOWN (*sds*)

(24) TYPE_IS_UNKNOWN_IN_SDS (*sds*, element of *key_types*)

(25) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(26) TYPE_NAME_IS_INVALID (*local_name*)

10.2.6 SDS_CREATE_ENUMERAL_TYPE

(1)
```
SDS_CREATE_ENUMERAL_TYPE (
    sds            : Sds_designator,
    local_name     : [ Name ]
)
    new_type       : Enumeral_type_nominator_in_sds
```

(2) SDS_CREATE_ENUMERAL_TYPE creates a new enumeral type *new_type* and its associated enumeral type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(4) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(5) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation.

(6) The creation or importation time of *new_type_in_sds* is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string. The image of *new_type_in_sds* is set to the empty string.

(7) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(8) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(9) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

Errors

(10) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(11) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(12) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_type_in_sds*)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(14) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(15) SDS_IS_PREDEFINED (*sds*)

(16) SDS_IS_UNKNOWN (*sds*)

(17) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(18) TYPE_NAME_IS_INVALID (*local_name*)

10.2.7 SDS_CREATE_ENUMERATION_ATTRIBUTE_TYPE

(1)
```
SDS_CREATE_ENUMERATION_ATTRIBUTE_TYPE (
    sds             : Sds_designator,
    local_name      : [ Name ],
    values          : Enumeration_values,
    duplication     : Duplication,
    initial_value   : [ Natural ]
)
    new_type        : Attribute_type_nominator_in_sds
```

(2) SDS_CREATE_ENUMERATION_ATTRIBUTE_TYPE creates a new enumeration attribute type *new_type*, and its associated attribute type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(4) "enumeral" links are created from *new_type* to the enumeral types specified in *values*. The "position" key attributes of the links are set according to the sequential order of the type nominators in *value*, starting at 0 and incremented by 1 for each enumeral type. The reverse "enumeral_of" links are also created.

(5) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(6) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation. The duplication of *new_type* is set to *duplication*. The initial value position of *new_type* is set to *initial_value* if supplied, and otherwise to 0.

(7) The three definition mode attributes of *new_type_in_sds* are set to 12, representing READ_MODE and WRITE_MODE, and its creation or importation time is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(8) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(9) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(10) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

(11) A write lock of the default mode is obtained on any enumeral type specified in *values* if the OWNER discretionary access right is granted for a group to *new_type* (the default object owner group), and a different OWNER discretionary access right exists for the same group to that enumeral type.

Errors

(12) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(13) ACCESS_ERRORS (element of *values*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(14) For each enumeral type in SDS E associated with an element of *values*:
ACCESS_ERRORS (E, ATOMIC, READ, READ_ATTRIBUTES)

(15) COMPONENT_ADDITION_ERRORS (enumeral type, "enumeral" link)

(16) ENUMERAL_TYPES_ARE_MULTIPLE (*values*)

(17) ENUMERATION_ATTRIBUTE_WOULD_HAVE_NO_ENUMERAL_TYPES (*values*)

(18) ENUMERATION_VALUE_IS_OUT_OF_RANGE (*initial_value*, *values*)

(19) IMAGE_IS_DUPLICATED (*values*, *sds*)

(20) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(21) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION
(*new_type_in_sds*)

(22) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(23) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(24) SDS_IS_PREDEFINED (*sds*)

(25) SDS_IS_UNKNOWN (*sds*)

(26) TYPE_IS_UNKNOWN_IN_SDS(*sds*, element of *values*)

(27) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(28) TYPE_NAME_IS_INVALID (*local_name*)

10.2.8 SDS_CREATE_FLOAT_ATTRIBUTE_TYPE

(1)
```
SDS_CREATE_FLOAT_ATTRIBUTE_TYPE (
    sds             : Sds_designator,
    local_name      : [ Name ],
    initial_value   : [ Float ],
    duplication     : Duplication
)
    new_type        : Attribute_type_nominator_in_sds
```

(2) SDS_CREATE_FLOAT_ATTRIBUTE_TYPE creates a new float attribute type *new_type*, and its associated attribute type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(4) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(5) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation. The duplication of *new_type* is set to *duplication*. The float initial value of *new_type* is set to *initial_value* if supplied, and otherwise to 0.0 (zero).

(6) The three definition mode attributes of *new_type_in_sds* are set to 12, representing READ_MODE and WRITE_MODE, and its creation or importation time is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(7) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(8) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(9) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

Errors

(10) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(11) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(12) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_type_in_sds*)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(14) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(15) SDS_IS_PREDEFINED (*sds*)

(16) SDS_IS_UNKNOWN (*sds*)

(17) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(18) TYPE_NAME_IS_INVALID (*local_name*)

(19) VALUE_LIMIT_ERRORS (*initial_value*)

10.2.9 SDS_CREATE_INTEGER_ATTRIBUTE_TYPE

(1)
```
SDS_CREATE_INTEGER_ATTRIBUTE_TYPE (
    sds             : Sds_designator,
    local_name      : [ Name ],
    initial_value   : [ Integer ],
    duplication     : Duplication
)
    new_type        : Attribute_type_nominator_in_sds
```

(2) SDS_CREATE_INTEGER_ATTRIBUTE_TYPE creates a new integer attribute type *new_type*, and its associated attribute type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(4) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(5) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation. The duplication of *new_type* is set to *duplication*. The integer initial value of *new_type* is set to *initial_value* if supplied, and otherwise to 0 (zero).

(6) The three definition mode attributes of *new_type_in_sds* are set to 12, representing READ_MODE and WRITE_MODE, and its creation or importation time is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(7) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(8) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(9) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

Errors

(10) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(11) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(12) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION
(*new_type_in_sds*)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(14) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(15) SDS_IS_UNKNOWN (*sds*)

(16) SDS_IS_PREDEFINED (*sds*)

(17) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(18) TYPE_NAME_IS_INVALID (*local_name*)

(19) VALUE_LIMIT_ERRORS (*initial_value*)

10.2.10 SDS_CREATE_NATURAL_ATTRIBUTE_TYPE

(1)
```
SDS_CREATE_NATURAL_ATTRIBUTE_TYPE (
    sds             : Sds_designator,
    local_name      : [ Name ],
    initial_value   : [ Natural ],
    duplication     : Duplication
)
    new_type        : Attribute_type_nominator_in_sds
```

(2) SDS_CREATE_NATURAL_ATTRIBUTE_TYPE creates a new natural attribute type *new_type*, and its associated attribute type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(4) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(5) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation. The duplication of *new_type* is set to *duplication*. The natural initial value of *new_type* is set to *initial_value* if supplied, and otherwise to 0 (zero).

(6) The three definition mode attributes of *new_type_in_sds* are set to 12, representing READ_MODE and WRITE_MODE, and its creation or importation time is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(7) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(8) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(9) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

Errors

(10) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(11) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(12) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_type_in_sds*)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(14) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(15) SDS_IS_PREDEFINED (*sds*)

(16) SDS_IS_UNKNOWN (*sds*)

(17) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(18) TYPE_NAME_IS_INVALID (*local_name*)

(19) VALUE_LIMIT_ERRORS (*initial_value*)

10.2.11 SDS_CREATE_OBJECT_TYPE

(1)
```
SDS_CREATE_OBJECT_TYPE (
    sds           : Sds_designator,
    local_name    : [ Name ],
    parents       : Object_type_nominators_in_sds
)
    new_type      : Object_type_nominator_in_sds
```

(2) SDS_CREATE_OBJECT_TYPE creates a new object type *new_type* and its associated object type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The contents type of *new_type* is determined as follows. If one or more of the object types in *parents* has a contents type (if more than one, they must all have the same contents type) then *new_type* has that contents type; otherwise *new_type* has no contents type.

(4) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation.

(5) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(6) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link. "parent_type" links are created from *new_type* to each of *parents*, together with their reverse "child_type" links.

(7) The three definition mode attributes of *new_type_in_sds* are set to 1, representing CREATE_MODE, and its creation or importation time is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(8) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(9) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(10) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

Errors

(11) ACCESS_ERRORS (element of *parents*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(12) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(13) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(14) OBJECT_TYPE_WOULD_HAVE_NO_PARENT_TYPE (*parents*)

(15) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_type_in_sds*)

(16) PARENT_BASIC_TYPES_ARE_MULTIPLE (*parents*)

(17) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(18) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(19) SDS_IS_PREDEFINED (*sds*)

(20) SDS_IS_UNKNOWN (*sds*)

(21) TYPE_IS_UNKNOWN_IN_SDS (*sds*, element of *parents*)

(22) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(23) TYPE_NAME_IS_INVALID (*local_name*)

10.2.12 SDS_CREATE_RELATIONSHIP_TYPE

(1)
```
SDS_CREATE_RELATIONSHIP_TYPE (
    sds                        : Sds_designator,
    forward_local_name         : [ Name ],
    forward_category           : Category,
    forward_lower_bound        : Natural,
    forward_upper_bound        : [ Natural ],
    forward_exclusiveness      : Exclusiveness,
    forward_stability          : Stability,
    forward_duplication        : Duplication,
    forward_key_types          : [ Key_types_in_sds ],
    reverse_local_name         : [ Name ],
    reverse_category           : Category,
    reverse_lower_bound        : Natural,
    reverse_upper_bound        : [ Natural ],
    reverse_exclusiveness      : Exclusiveness,
    reverse_stability          : Stability,
    reverse_duplication        : Duplication,
    reverse_key_types          : [ Key_types_in_sds ]
)
    new_forward_type           : Link_type_nominator_in_sds,
    new_reverse_type           : Link_type_nominator_in_sds
```

(2) SDS_CREATE_RELATIONSHIP_TYPE creates two new non-designation link types *new_forward_type* and *new_reverse_type*, and their associated types in SDS *new_forward_type_in_sds* and *new_reverse_type_in_sds* in the SDS *sds*. The new link types are the reverse of each other.

(3) The operation creates "definition" links from *sds* to *new_forward_type_in_sds* and *new_reverse_type_in_sds*, and their reverse "type_in_sds" links; the keys of the "definition" links are the system-assigned type identifiers of *new_forward_type* and *new_reverse_type*. The operation also creates "of_type" links from *new_forward_type_in_sds* to *new_forward_type* and from *new_reverse_type_in_sds* to *new_reverse_type*, and their reverse "has_type_in_sds" links.

(4) If *forward_local_name* is supplied, a "named_definition" link is created from *sds* to *new_forward_type_in_sds* with *forward_local_name* as key. If *reverse_local_name* is supplied, a "named_definition" link is created from *sds* to *new_reverse_type_in_sds* with *reverse_local_name* as key, together with its reverse "named_in_sds" link.

(5) The type identifiers of *new_forward_type* and *new_reverse_type* are set to implementation-dependent values which identify the types within the PCTE installation.

(6) Two "reverse" links are created between *new_forward_type* and *new_reverse_type*, one in each direction.

(7) The three definition mode attributes of *new_forward_type_in_sds* and *new_reverse_type_in_sds* are set to 19, representing CREATE_MODE, DELETE_MODE, and NAVIGATE_MODE, and their creation or importation times are set to the system time. If *forward_local_name* is supplied, the annotation of *new_forward_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string. If *reverse_local_name* is supplied, the annotation of *new_reverse_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(8) The category, lower and upper bounds, exclusiveness, stability, and duplication of *new_forward_type* are set from *forward_category*, *forward_lower_bound*, *forward_upper_ bound*, *forward_exclusiveness*, *forward_stability*, and *forward_duplication*, respectively; the

category, lower and upper bounds, exclusiveness, stability, and duplication of *new_reverse_type* are set from *reverse_category*, *reverse_lower_bound*, *reverse_upper_bound*, *reverse_exclusiveness*, *reverse_stability*, and *reverse_duplication*, respectively. Furthermore, if for either link type the cardinality is MANY, for each attribute type of *forward_key_types* or *reverse_key_types*, a "key_attribute" link is created from *new_forward_type* or *new_reverse_type* respectively to that attribute type. The keys of these links correspond to the order of the key attribute types in *forward_key_types* or *reverse_key_types* starting at 0 and incremented by 1.

(9) The new objects reside on the same volume as *sds*. Their access control lists are built using the default access control list of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(10) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(11) Write locks of the default mode are obtained on the created objects and links (except the new "object_on_volume" links).

Errors

(12) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(13) ACCESS_ERRORS (element of *forward_key_types*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(14) ACCESS_ERRORS (element of *reverse_key_types*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(15) KEY_TYPE_IS_BAD (element of *forward_key_types*)

(16) KEY_TYPE_IS_BAD (element of *reverse_key_types*)

(17) KEY_TYPES_ARE_MULTIPLE (*forward_key_types*)

(18) KEY_TYPES_ARE_MULTIPLE (*reverse_key_types*)

(19) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(20) LINK_TYPE_CATEGORY_IS_BAD (*forward_link_type*, (COMPOSITION, EXISTENCE, REFERENCE, IMPLICIT))

(21) LINK_TYPE_CATEGORY_IS_BAD (*reverse_link_type*, (COMPOSITION, EXISTENCE, REFERENCE, IMPLICIT))

(22) LINK_TYPE_PROPERTIES_ARE_INCONSISTENT (*forward_category*, *forward_lower_bound*, *forward_upper_bound*, *forward_exclusiveness*, *forward_stability*, *forward_duplication*)

(23) LINK_TYPE_PROPERTIES_ARE_INCONSISTENT (*reverse_category*, *reverse_lower_bound*, *reverse_upper_bound*, *reverse_exclusiveness*, *reverse_stability*, *reverse_duplication*)

(24) LINK_TYPE_PROPERTIES_AND_KEY_TYPES_ARE_INCONSISTENT (*forward_category*, *forward_lower_bound*, *forward_upper_bound*, *forward_exclusiveness*, *forward_stability*, *forward_duplication*, *forward_key_types*)

(25) LINK_TYPE_PROPERTIES_AND_KEY_TYPES_ARE_INCONSISTENT (*reverse_category*, *reverse_lower_bound*, *reverse_upper_bound*, *reverse_exclusiveness*, *reverse_stability*, *reverse_duplication*, *reverse_key_types*)

(26) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (type in sds of *forward_link_type*)
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (type in sds of *reverse_link_type*)

(27) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(28) RELATIONSHIP_TYPE_PROPERTIES_ARE_INCONSISTENT (*forward_category*, *forward_lower_bound*, *forward_upper_bound*, *forward_exclusiveness*, *forward_stability*, *forward_duplication*, *reverse_category*, *reverse_lower_bound*, *reverse_upper_bound*, *reverse_exclusiveness*, *reverse_stability*, *reverse_duplication*)

(29) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(30) SDS_IS_PREDEFINED (*sds*)

(31) SDS_IS_UNKNOWN (*sds*)

(32) TYPE_IS_UNKNOWN_IN_SDS (*sds*, element of *forward_key_types*)

(33) TYPE_IS_UNKNOWN_IN_SDS (*sds*, element of *reverse_key_types*)

(34) TYPE_NAME_IN_SDS_IS_DUPLICATE (*forward_local_name*)

(35) TYPE_NAME_IN_SDS_IS_DUPLICATE (*reverse_local_name*)

(36) TYPE_NAME_IS_INVALID (*forward_local_name*)

(37) TYPE_NAME_IS_INVALID (*reverse_local_name*)

10.2.13 SDS_CREATE_STRING_ATTRIBUTE_TYPE

(1)
```
SDS_CREATE_STRING_ATTRIBUTE_TYPE (
    sds             : Sds_designator,
    local_name      : [ Name ],
    initial_value   : [ String ],
    duplication     : Duplication
)
    new_type        : Attribute_type_nominator_in_sds
```

(2) SDS_CREATE_STRING_ATTRIBUTE_TYPE creates a new string attribute type *new_type*, and its associated attribute type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(4) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(5) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation. The duplication of *new_type* is set to *duplication*. The string initial value of *new_type* is set to *initial_value* if supplied, and otherwise to the empty string.

(6) The three definition mode attributes of *new_type_in_sds* are set to 12, representing READ_MODE and WRITE_MODE, and its creation or importation time is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(7) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process, and their confidentiality

labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(8) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(9) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

Errors

(10) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(11) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(12) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION
(*new_type_in_sds*)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(14) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(15) SDS_IS_PREDEFINED (*sds*)

(16) SDS_IS_UNKNOWN (*sds*)

(17) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(18) TYPE_NAME_IS_INVALID (*local_name*)

(19) VALUE_LIMIT_ERRORS *initial_value*)

10.2.14 SDS_CREATE_TIME_ATTRIBUTE_TYPE

(1)
```
SDS_CREATE_TIME_ATTRIBUTE_TYPE (
    sds            : Sds_designator,
    local_name     : [ Name ],
    initial_value  : [ Time ],
    duplication    : Duplication
)
    new_type       : Attribute_type_nominator_in_sds
```

(2) SDS_CREATE_TIME_ATTRIBUTE_TYPE creates a new time attribute type *new_type*, and its associated attribute type in SDS *new_type_in_sds* in the SDS *sds*.

(3) The operation creates a "definition" link from *sds* to *new_type_in_sds*; the key of the link is the system-assigned type identifier of *new_type*. The operation also creates an "of_type" link from *new_type_in_sds* to *new_type*.

(4) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_type_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(5) The type identifier of *new_type* is set to an implementation-defined value which identifies the type within the PCTE installation. The duplication of *new_type* is set to *duplication*. The time initial value of *new_type* is set to *initial_value* if supplied, and otherwise to 1980-01-01T00:00:00Z.

(6) The three definition mode attributes of *new_type_in_sds* are set to 12, representing READ_MODE and WRITE_MODE, and its creation or importation time is set to the system time. If *local_name* is supplied, the annotation of *new_type_in_sds* is set to the complete name of the created type; otherwise it is set to the empty string.

(7) The new objects reside on the same volume as *sds*. Their access control lists are built using the default atomic ACL and the default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(8) For each created object, an "object_on_volume" link is created from the volume on which the object resides to the object. The key of the link is the exact identifier of the object.

(9) Write locks of the default mode are obtained on the created objects and links except the new "object_on_volume" links.

Errors

(10) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(11) ATTRIBUTE_VALUE_LIMIT_WOULD_BE_EXCEEDED (*initial_value*)

(12) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(13) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_type_in_sds*)

(14) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(15) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(16) SDS_IS_PREDEFINED (*sds*)

(17) SDS_IS_UNKNOWN (*sds*)

(18) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(19) TYPE_NAME_IS_INVALID (*local_name*)

(20) VALUE_LIMIT_ERRORS (*initial_value*)

10.2.15 SDS_GET_NAME

(1)
```
SDS_GET_NAME (
    sds     : Sds_designator
)
    name    : Name
```

(2) SDS_GET_NAME returns the name of the SDS *sds*.

(3) The returned name *name* is the key of the "known_sds" link from the SDS directory to *sds*.

(4) A read lock of the default mode is obtained on that link.

Errors

(5) ACCESS_ERRORS (the SDS directory, ATOMIC, READ, READ_LINKS)

(6) SDS_IS_PREDEFINED (*sds*)

(7) SDS_IS_UNKNOWN (*sds*)

10.2.16 SDS_IMPORT_ATTRIBUTE_TYPE

(1)
```
SDS_IMPORT_ATTRIBUTE_TYPE (
    to_sds          : Sds_designator,
    from_sds        : Sds_designator,
    type            : Attribute_type_nominator_in_sds,
    local_name      : [ Name ]
)
```

(2) SDS_IMPORT_ATTRIBUTE_TYPE imports the attribute type *type* from the SDS *from_sds* to the SDS *to_sds*, along with the associated enumeral types if *type* is an enumeration attribute type.

(3) If *type* is an enumeration attribute type, all its enumeral types are implicitly imported, if not already in *to_sds*. The implicitly imported enumeral types have the same images as in *from_sds*, but do not have local names in *to_sds*.

(4) The operation creates an attribute type in SDS *type_in_sds* in *to_sds* associated with *type*. If *type* is an enumeration attribute type, it also creates an enumeral type in SDS in *sds* associated with each enumeral type of *type* (unless one already exists). For each of the created types in SDS a "definition" link is created from *to_sds* whose key is the type identifier of the associated type.

(5) An "of_type" link from each new type in SDS to its associated type and its reverse "has_type_in_sds" link are created.

(6) If *local_name* is supplied, or if *type* has a local name in *from_sds*, a "named_definition" link from *to_sds* to *type_in_sds* and its reverse "named_in_sds" link are created. The key of the "named_definition" link is *local_name* if supplied, otherwise the local name of *type* in *from_sds*.

(7) Each of the three definition mode attributes of *type_in_sds* is set to the export mode of the corresponding type in SDS in *from_sds*.

(8) The creation or importation time of each new type in SDS is set to the system time.

(9) The annotation of each new type in SDS is the same as the annotation of the corresponding type in SDS in *from_sds*.

(10) The new types in SDS reside on the same volume as *to_sds*. Their access control lists are built using the default atomic ACL and default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(11) An "object_on_volume" link is created to each new type in SDS from the volume on which it resides. The key of the link is the exact identifier of the new type in SDS.

(12) Read locks of the default mode are obtained on the types in SDS in *from_sds*. Write locks of the default mode are obtained on the new types in SDS and links (except the new "object_on_volume" links).

Errors

(13) ACCESS_ERRORS (*from_sds*, ATOMIC, READ, NAVIGATE)

(14) ACCESS_ERRORS (*to_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(15) ACCESS_ERRORS (the type in SDS associated with *type* in *from_sds*, ATOMIC, READ, EXPLOIT_SCHEMA)

(16) ACCESS_ERRORS (the type in SDS associated with *type* in *from_sds*, ATOMIC, READ, READ_ATTRIBUTES)

(17) ACCESS_ERRORS (the "type" object associated with *type*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(18) If *type* is an enumeration attribute type, for each enumeral type in SDS S associated with *type* in *from_sds*:
ACCESS_ERRORS (S, ATOMIC, READ, EXPLOIT_SCHEMA)

(19) If *type* is an enumeration attribute type, for each enumeral type E associated with *type* not already present in *to_sds*:
ACCESS_ERRORS (E, ATOMIC, CHANGE, APPEND_IMPLICIT)

(20) If *type* is an enumeration attribute type:
IMAGE_IS_DUPLICATED (enumeral types of *type*, *to_sds*)

(21) If *to_sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*type_in_sds*)

(22) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(23) SDS_IS_IN_A_WORKING_SCHEMA (*to_sds*)

(24) SDS_IS_PREDEFINED (*sds*)

(25) SDS_IS_UNKNOWN (*to_sds*)

(26) SDS_IS_UNKNOWN (*from_sds*)

(27) TYPE_IS_ALREADY_KNOWN_IN_SDS (*type*, *to_sds*)

(28) If *local_name* is supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, *local_name*)

(29) If *local_name* is not supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, local name of *type* in *from_sds*)

(30) TYPE_IS_UNKNOWN_IN_SDS (*from_sds*, *type*)

(31) TYPE_NAME_IS_INVALID (*local_name*)

10.2.17 SDS_IMPORT_ENUMERAL_TYPE

(1)
```
SDS_IMPORT_ENUMERAL_TYPE (
    to_sds        : Sds_designator,
    from_sds      : Sds_designator,
    type          : Enumeral_type_nominator_in_sds,
    local_name    : [ Name ]
)
```

(2) SDS_IMPORT_ENUMERAL_TYPE imports the enumeral type *type* from the SDS *from_sds* to the SDS *to_sds*.

(3) The operation creates an enumeral type in SDS *type_in_sds* in *to_sds* associated with *type*. A "definition" link is created from *to_sds* to *type_in_sds* whose key is the type identifier of *type*, together with its reverse "in_sds" link.

(4) An "of_type" link is created from *type_in_sds* to *type*, together with its reverse "has_type_in_sds" link.

(5) If *local_name* is supplied, or if *type* has a local name in *from_sds*, a "named_definition" link is created from *to_sds* to *type_in_sds*, together with its reverse "named_in_sds" link. The key of the "named_definition" link is *local_name* if supplied, otherwise the local name of *type* in *from_sds*.

(6) The creation or importation time of *type_in_sds* is set to the system time.

(7) The annotation of *type_in_sds* is the same as the annotation of the corresponding type in SDS in *from_sds*.

(8) *type_in_sds* resides on the same volume as *to_sds*. Its access control lists are built using the default atomic ACL and the default object owner of the calling process, and its confidentiality label and integrity label is set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(9) An "object_on_volume" link is created to *type_in_sds* from the volume on which it resides. The key of the link is the exact identifier of *type_in_sds*.

(10) Read locks of the default mode are obtained on the type in SDS in *from_sds*. Write locks of the default mode are obtained on *type_in_sds* and the created links (except the new "object_on_volume" link).

Errors

(11) ACCESS_ERRORS (*from_sds*, ATOMIC, READ, NAVIGATE)

(12) ACCESS_ERRORS (*to_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(13) ACCESS_ERRORS (the type in SDS associated with *type* in *from_sds*, ATOMIC, READ, EXPLOIT_SCHEMA)

(14) ACCESS_ERRORS (the type in SDS associated with *type* in *from_sds*, ATOMIC, READ, READ_ATTRIBUTES)

(15) ACCESS_ERRORS (the "type" object associated with *type*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(16) If *to_sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*type_in_sds*)

(17) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(18) SDS_IS_IN_A_WORKING_SCHEMA (*to_sds*)

(19) SDS_IS_PREDEFINED (*to_sds*)

(20) SDS_IS_UNKNOWN (*to_sds*)

(21) SDS_IS_UNKNOWN (*from_sds*)

(22) TYPE_IS_ALREADY_KNOWN_IN_SDS (*type*, *to_sds*)

(23) TYPE_IS_UNKNOWN_IN_SDS (*from_sds*, *type*)

(24) If *local_name* is supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, *local_name*)

(25) If *local_name* is not supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, local name of *type* in *from_sds*)

(26) TYPE_NAME_IS_INVALID (*local_name*)

10.2.18 SDS_IMPORT_LINK_TYPE

(1)
```
SDS_IMPORT_LINK_TYPE (
    to_sds          : Sds_designator,
    from_sds        : Sds_designator,
    type            : Link_type_nominator_in_sds,
    local_name      : [ Name ]
)
```

(2) SDS_IMPORT_LINK_TYPE imports the link type *type* from the SDS *from_sds* to the SDS *to_sds*.

(3) All the key attribute types of *type*, and its reverse link type with all its key attributes, are implicitly imported, if not already in *to_sds*. The imported link types have the same key attributes as in *from_sds*. The link and the key attribute types implicitly imported do not have local names assigned to them within *to_sds*.

(4) The importation of a type (either explicitly or implicitly) results in the creation of a type in SDS in *to_sds* associated with the imported type, with a "definition" link from *from_sds* whose key is the type identifier of the imported type. An "of_type" link from the new type in SDS to the imported type and its reverse "has_type_in_sds" link are created.

(5) If *local_name* is supplied or if *type* has a name in *from_sds*, a "named_definition" link is created from *to_sds* to the new link type in SDS associated with *type*, together with its reverse "named_in_sds" link. The key of the "named_definition" link is *local_name* if supplied, otherwise it is the local name of *type* in the SDS *from_sds*.

(6) Each of the three definition mode attributes of each new type in SDS is set to the export mode of the corresponding type in SDS in *from_sds*.

(7) The creation or importation time of each new type in SDS is set to the system time.

(8) The annotation of each new type in SDS is the same as the annotation of the corresponding type in SDS in *from_sds*.

(9) The new types in SDS reside on the same volume as *to_sds.* Their access control lists are built using the default atomic ACL and default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(10) An "object_on_volume" link is created to each new type in SDS from the volume on which it resides. The key of the link is the exact identifier of the new type in SDS.

(11) Read locks of the default mode are obtained on the types in SDS in *from_sds*. Write locks of the default mode are obtained on the new types in SDS and links (except the new "object_on_volume" links).

Errors

(12) ACCESS_ERRORS (*from_sds*, ATOMIC, READ, NAVIGATE)

(13) ACCESS_ERRORS (*to_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(14) ACCESS_ERRORS (type in SDS associated with *type* in *from_sds*, ATOMIC, READ, EXPLOIT_SCHEMA)

(15) ACCESS_ERRORS (the type in SDS associated with *type* in *from_sds,* ATOMIC, READ, READ_ATTRIBUTES)

(16) For each attribute type in SDS A associated with a key attribute type of *type*:
ACCESS_ERRORS (A, ATOMIC, READ, EXPLOIT_SCHEMA)

(17) ACCESS_ERRORS (the "type" object associated with *type,* ATOMIC, CHANGE, APPEND_IMPLICIT)

(18) For each key attribute type K of *type* not already present in *to_sds*:
ACCESS_ERRORS (K, ATOMIC, CHANGE, APPEND_IMPLICIT)

(19) If *type* has a reverse link type R not already present in *to_sds*:

(20) ACCESS_ERRORS (R, ATOMIC, CHANGE, APPEND_IMPLICIT)

(21) For each key attribute type K1 of R not already present in *to_sds*:
ACCESS_ERRORS (K1, ATOMIC, CHANGE, APPEND_IMPLICIT)

(22) If *to_sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*type_in_sds*)

(23) If *to_sds* has OWNER granted or denied and *link_type* has a reverse link type:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (reverse of *type_in_sds*)

(24) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(25) SDS_IS_IN_A_WORKING_SCHEMA (*to_sds*)

(26) SDS_IS_PREDEFINED (*to_sds*)

(27) SDS_IS_UNKNOWN (*to_sds*)

(28) SDS_IS_UNKNOWN (*from_sds*)

(29) TYPE_IS_ALREADY_KNOWN_IN_SDS (*type*, *to_sds*)

(30) If *local_name* is supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, *local_name*)

(31) If *local_name* is not supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, local name of *type* in *from_sds*)

(32) TYPE_NAME_IS_INVALID (*local_name*)

(33) TYPE_IS_UNKNOWN_IN_SDS (*from_sds*, *type*)

10.2.19 SDS_IMPORT_OBJECT_TYPE

(1)
```
SDS_IMPORT_OBJECT_TYPE (
    to_sds         : Sds_designator,
    from_sds       : Sds_designator,
    type           : Object_type_nominator_in_sds,
    local_name     : [ Name ]
)
```

(2) SDS_IMPORT_OBJECT_TYPE imports the object type *type* from the SDS *from_sds* to the SDS *to_sds*.

(3) The importation of an object type implies the implicit importation of all its ancestor types if not already in *to_sds*. The attribute and link types applied to the explicitly or implicitly imported types are not imported, nor is the notion of their application. The object types implicitly imported do not have a local name assigned to them within *to_sds*

(4) The importation of an object type (either explicit or implicit) results in the creation of an object type in SDS in *to_sds* with a "definition" link from *to_sds* whose key is the type identifier of the imported type. An "of_type" link from the new object type in SDS to the imported type and its reverse "has_type_in_sds" link are created.

(5) If *local_name* is supplied or if the imported *type* has a name in the originating SDS, a "named_definition" link is created from *to_sds* to the new object type in SDS associated with *link_type*, together with its reverse "named_in_sds" link. The key of the "named_definition" link is *local_name* if supplied, otherwise it is the local name of *type* in *from_sds*.

(6) Each of the three definition mode attributes of each new type in SDS is set to the export mode of the corresponding type in SDS in *from_sds*.

(7) The creation or importation time of each new type in SDS is set to the system time.

(8) The annotation of each new type in SDS is the same as the annotation of the corresponding type in SDS in *from_sds*.

(9) The new types in SDS reside on the same volume as *to_sds*. Their access control lists are built using the default atomic ACL and default object owner of the calling process, and their confidentiality labels and integrity labels are set to be equal to the current confidentiality context and integrity context, respectively, of the calling process.

(10) An "object_on_volume" link is created to each new type in SDS from the volume on which it resides. The key of the link is the exact identifier of the new type in SDS.

(11) Read locks of the default mode are obtained on the types in SDS in *from_sds*. Write locks of the default mode are obtained on the new types in SDS and links (except the new "object_on_volume" links).

Errors

(12) ACCESS_ERRORS (*from_sds*, ATOMIC, READ, NAVIGATE)

(13) ACCESS_ERRORS (*to_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(14) ACCESS_ERRORS (the type in SDS associated with /type/ in /from_sds/, ATOMIC, READ, READ_ATTRIBUTES)

(15) For each object type in SDS S associated with the ancestor types of *type*:
ACCESS_ERRORS (S, ATOMIC, READ, EXPLOIT_SCHEMA)

(16) ACCESS_ERRORS (object type in SDS associated with *type* in *from_sds*, ATOMIC, READ, EXPLOIT_SCHEMA)

(17) ACCESS_ERRORS (the "type" object associated with *type*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(18) For each ancestor object type A of *type* not already present in *to_sds*:
ACCESS_ERRORS(A, ATOMIC, CHANGE, APPEND_IMPLICIT)

(19) If *to_sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*type_in_sds*)

(20) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(21) SDS_IS_IN_A_WORKING_SCHEMA (*to_sds*)

(22) SDS_IS_PREDEFINED (*to_sds*)

(23) SDS_IS_UNKNOWN (*to_sds*)

(24) SDS_IS_UNKNOWN (*from_sds*)

(25) TYPE_IS_ALREADY_KNOWN_IN_SDS (*type*, *to_sds*)

(26) If *local_name* is supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, *local_name*)

(27) If *local_name* is not supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, local name of *type* in *from_sds*)

(28) TYPE_IS_UNKNOWN_IN_SDS (*from_sds*, *type*)

(29) TYPE_NAME_IS_INVALID (*local_name*)

10.2.20 SDS_INITIALIZE

(1)
```
SDS_INITIALIZE (
    sds     : Sds_designator,
    name    : Name
)
```

(2) SDS_INITIALIZE establishes the SDS *sds* as a known SDS by creating a "known_sds" link with key *name* from the master of the SDS directory to *sds*.

(3) A read lock of the default mode is obtained on *sds* and a write lock of the default mode is obtained on the created link.

Errors

(4) ACCESS_ERRORS (*sds*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(5) ACCESS_ERRORS (master of the SDS directory, ATOMIC, MODIFY, APPEND_LINKS)

(6) If *sds* has successors or predecessors:
ACCESS_ERRORS (successor or predecessor of *sds*, ATOMIC, SYSTEM_ACCESS)

(7) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(8) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(9) SDS_IS_KNOWN (*sds*)

(10) SDS_IS_NOT_EMPTY_NOR_VERSION (*sds***)**

(11) SDS_IS_PREDEFINED (*sds*)

(12) SDS_NAME_IS_DUPLICATE (*name*)

(13) SDS_NAME_IS_INVALID (*name*)

10.2.21 SDS_REMOVE

(1)
```
SDS_REMOVE (
    sds     : Sds_designator
)
```

(2) SDS_REMOVE removes the SDS *sds* from the set of known SDSs.

(3) The "known_sds" link to *sds* from the SDS directory is deleted. If that link is the last composition or existence link to *sds*, then the "sds" object *sds* is deleted. In that case, the "object_on_volume" link from the volume on which *sds* was residing to *sds* is also deleted.

(4) A read lock of the default mode is obtained on *sds* if it is not deleted; a write lock otherwise. Write locks of the default mode are obtained on the deleted links except the deleted "object_on_volume" link.

Errors

(5) ACCESS_ERRORS (the SDS directory, ATOMIC, MODIFY, WRITE_LINKS)

(6) ACCESS_ERRORS (*sds*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(7) If the conditions hold for deletion of the "sds" object *sds*:
ACCESS_ERRORS (*sds,* COMPOSITE, MODIFY, DELETE)

(8) If *sds* has predecessors or successors:
ACCESS_ERRORS (predecessor or successor of *sds*, ATOMIC, SYSTEM_ACCESS)

(9) OBJECT_HAS_LINKS_PREVENTING_DELETION (*sds*)

(10) OBJECT_IS_IN_USE_FOR_DELETE (*sds*)

(11) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(12) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(13) SDS_IS_NOT_EMPTY_NOR_VERSION (*sds*)

(14) SDS_IS_PREDEFINED (*sds*)

(15) SDS_IS_UNKNOWN (*sds*)

10.2.22 SDS_REMOVE_DESTINATION

(1)
```
SDS_REMOVE_DESTINATION(
    sds             : Sds_designator,
    link_type       : Link_type_nominator_in_sds,
    object_type     : Object_type_nominator_in_sds
)
```

(2) SDS_REMOVE_DESTINATION removes the object type in SDS *object_type_in_sds* associated with *object_type* in the SDS *sds* from the destination object types of the link type in SDS *link_type_in_sds* associated with *link_type* in *sds*.

(3) As a result, the "in_destination_set" link from *link_type_in_sds* to *object_type_in_sds* and its reverse "is_destination_of" link are deleted.

(4) If *link_type* has a reverse link type *reverse*, then *reverse* is unapplied (see SDS_UNAPPLY_LINK_TYPE) and the "in_link_set" link existing between *object_type_in_sds* and the "link_type_in_sds" object *reverse_link_type_in_sds* associated with *reverse* in *sds* is deleted.

(5) Write locks of the default mode are obtained on the deleted links.

Errors

(6) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(7) ACCESS_ERRORS (*object_type_in_sds*, ATOMIC, MODIFY, WRITE_LINKS)

(8) ACCESS_ERRORS (*link_type_in_sds*, ATOMIC, MODIFY, WRITE_LINKS)

(9) ACCESS_ERRORS (*reverse_link_type_in_sds*, ATOMIC, MODIFY, WRITE_LINKS)

(10) OBJECT_TYPE_IS_NOT_IN_DESTINATION_SET (*link_type*, *object_type*)

(11) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(12) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(13) SDS_IS_PREDEFINED (*sds*)

(14) SDS_IS_UNKNOWN (*sds*)

(15) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *link_type*)

(16) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *object_type*)

10.2.23 SDS_REMOVE_TYPE

(1)
```
SDS_REMOVE_TYPE (
    sds     : Sds_designator,
    type    : Type_nominator_in_sds
)
```

(2) SDS_REMOVE_TYPE removes from the SDS *sds* the type in SDS *type_in_sds* associated with *type* in *sds*.

(3) When a link type is removed from an SDS, the type in SDS *reverse_link_type_in_sds* associated with the reverse type *reverse_type* of *type* (if any) is also removed from that SDS.

(4) If *type_in_sds* is the last type in SDS associated with *type* in any SDS, then *type* is also removed. If *reverse_link_type_in_sds* exists and is the last type in SDS associated with *reverse_type* in any SDS, then *reverse_type* is also removed. The type identifier of a removed type is never reassigned.

(5) A type can be removed while there are instances of the type in the object base. Instances of the removed type are not directly affected by this operation: objects, attributes and links retain the type properties of the type. The description of the type is however lost; the implications are:

(6) - no instances of a removed type can be created;

(7) - attributes of a removed type are inaccessible. They can however still be consulted or reset to their initial value using the system-generated type identifier of the deleted type (see 23.1.2.5);

(8) - links of a removed type can still be navigated through or deleted using the type identifier of the deleted type;

(9) - objects of a removed type can still be accessed as instances of visible ancestor types of the removed type.

(10) The *removal* of a type in SDS consists of the deletion of the "definition" link and "named_definition" link (if any) between the SDS and the type in SDS. The deletion of the "definition" link may result in the deletion of the type in SDS.

(11) The deletion of a type in SDS also entails the deletion of the "of_type" link from the type in SDS. In the case where this link is the last "of_type" link to the type (i.e. if the last occurrence of the type in SDS is to be deleted) the type is also removed.

(12) In turn, the removal of a type entails the loss of all the typing information held on its associated type object (such as the "parent_type", "key_attribute", "reverse" or "enumeral" links starting from that object) and may imply the deletion of the type itself (if the "of_type" link to be deleted is the last existence link to it and if there are no composition links to it).

(13) For each deleted object, the "object_on_volume" link from the volume on which the deleted object was residing to the deleted object is also deleted. A write lock of the default mode is obtained on *sds* and on the deleted objects and links (except the deleted "object_on_volume" links).

Errors

(14) ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, WRITE_LINKS)

(15) ACCESS_ERRORS (*type_in_sds*, ATOMIC, MODIFY, (WRITE_LINKS, WRITE_IMPLICIT))

(16) If the deletion of a type or type in SDS is implied:
ACCESS_ERRORS (destination object of a link from the type or type in SDS to be deleted which has an implicit reverse link, ATOMIC, CHANGE, WRITE_IMPLICIT)

(17) If conditions hold for the deletion of *type_in_sds*:
ACCESS_ERRORS (*type_in_sds*, COMPOSITE, MODIFY, DELETE)

(18) ACCESS_ERRORS ("type" object associated with *type*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(19) ACCESS_ERRORS (*type_in_sds*, COMPOSITE, MODIFY, DELETE)

(20) If the conditions for the deletion of the "type" object T associated with *type* are satisfied:

(21) ACCESS_ERRORS (T, COMPOSITE, MODIFY, DELETE)

(22) If T is an object type, for each parent type P of T:
ACCESS_ERRORS (P, ATOMIC, CHANGE, WRITE_IMPLICIT)

(23) If T is a link type, for each each key attribute type K of T:
ACCESS_ERRORS (K, ATOMIC, CHANGE, WRITE_IMPLICIT)

(24) If T is a link type with a reverse link type R:

(25) ACCESS_ERRORS (R, COMPOSITE, MODIFY, DELETE)

(26) For each each key attribute type K1 of R:
ACCESS_ERRORS (K1, ATOMIC, CHANGE, WRITE_IMPLICIT)

(27) If T is an enumeration attribute type, for each associated enumeral type E:
ACCESS_ERRORS (E, ATOMIC, CHANGE, WRITE_IMPLICIT)

(28) If the deletion of a type or type in SDS is implied:
OBJECT_HAS_LINKS_PREVENTING_DELETION (type or type in SDS to be deleted)

(29) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(30) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(31) SDS_IS_PREDEFINED (*sds*)

(32) SDS_IS_UNKNOWN (*sds*)

(33) TYPE_HAS_DEPENDENCIES (*sds*, *type*)

(34) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.2.24 SDS_SET_ENUMERAL_TYPE_IMAGE

(1)
```
SDS_SET_ENUMERAL_TYPE_IMAGE (
    sds     : Sds_designator,
    type    : Enumeral_type_nominator_in_sds,
    image   : [ Text ]
)
```

(2) SDS_SET_ENUMERAL_TYPE_IMAGE sets the image of the enumeral type in SDS *type_in_sds* associated with the enumeral type *type* in the SDS *sds* to the text value *image*, if supplied; if *image* is not supplied, the image of *type_in_sds* is set to the empty text value.

(3) A write lock of the default mode is obtained on *type_in_sds.*

Errors

(4) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(5) ACCESS_ERRORS (*type_in_sds*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(6) IMAGE_IS_ALREADY_ASSOCIATED (*image*, *sds*, *type*)

(7) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(8) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(9) SDS_IS_PREDEFINED (*sds*)

(10) SDS_IS_UNKNOWN (*sds*)

(11) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.2.25 SDS_SET_TYPE_MODES

(1)
```
SDS_SET_TYPE_MODES (
    sds            : Sds_designator,
    type           : Object_type_nominator_in_sds | Attribute_type_nominator_in_sds |
                         Link_type_nominator_in_sds,
    usage_mode     : [ Definition_mode_values ],
    export_mode    : [ Definition_mode_values ]
)
```

(2) SDS_SET_TYPE_MODES sets the usage mode and export mode of the type in SDS *type_in_sds* associated with the type *type* in the SDS *sds* to the values of *usage_mode* and *export_mode* respectively, if supplied; if, for either parameter, no value is supplied, then the corresponding value is left unchanged.

(3) If *type* is a link type with a reverse link type *reverse*, the usage mode and export mode of the link type in sds *reverse_type_in_sds* associated with *reverse* in *sds* are set (or not) in the same way.

(4) Write locks of the default mode are obtained on the modified "type_in_sds" objects.

Errors

(5) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(6) ACCESS_ERRORS (*type_in_sds*, COMPOSITE, MODIFY, WRITE_ATTRIBUTES)

(7) If *type* has a reverse link type in SDS *reverse_type_in_sds*:
ACCESS_ERRORS (*reverse_type_in_sds*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(8) DEFINITION_MODE_VALUE_WOULD_BE_INVALID (*export_mode*, *type*)

(9) DEFINITION_MODE_VALUE_WOULD_BE_INVALID (*usage_mode*, *type*)

(10) MAXIMUM_USAGE_MODE_WOULD_BE_EXCEEDED (*type*, *usage_mode*)

(11) MAXIMUM_USAGE_MODE_WOULD_BE_EXCEEDED (*type*, *export_mode*)

(12) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(13) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(14) SDS_IS_PREDEFINED (*sds*)

(15) SDS_IS_UNKNOWN (*sds*)

(16) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.2.26 SDS_SET_TYPE_NAME

(1)
```
SDS_SET_TYPE_NAME (
    sds            : Sds_designator,
    type           : Type_nominator_in_sds,
    local_name     : [ Name ]
)
```

(2) SDS_SET_TYPE_NAME sets the local name of the type in SDS *type_in_sds* associated with the type *type* in the SDS *sds* to *local_name*, if supplied, and otherwise deletes the local name of *type_in_sds* (if any).

(3) If *local_name* is supplied, a "named_definition" link is created from *sds* to *type_in_sds*. *local_name* is used as the key of the new link.

(4) If *type_in_sds* already had a local name, the corresponding "named_definition" link is deleted.

(5) Write locks of the default mode are obtained on the deleted links (if any) and write locks of the default mode are obtained on the created links (if any).

Errors

(6) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(7) If *type_in_sds* already has a local name:
ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, WRITE_LINKS)

(8) If *local_name* is supplied:
ACCESS_ERRORS (*sds*, ATOMIC, MODIFY, APPEND_LINKS)

(9) If *type_in_sds* already has a local name:
ACCESS_ERRORS (*type_in_sds*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(10) If *local_name* is supplied:
ACCESS_ERRORS (*type_in_sds*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(11) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(12) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(13) SDS_IS_PREDEFINED (*sds*)

(14) SDS_IS_UNKNOWN (*sds*)

(15) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

(16) If *local_name* is supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(17) TYPE_NAME_IS_INVALID (*local_name*)

10.2.27 SDS_UNAPPLY_ATTRIBUTE_TYPE

(1)
```
SDS_UNAPPLY_ATTRIBUTE_TYPE (
    sds             : Sds_designator,
    attribute_type  : Attribute_type_nominator_in_sds,
    type            : Object_type_nominator_in_sds | Link_type_nominator_in_sds
)
```

(2) SDS_UNAPPLY_ATTRIBUTE_TYPE removes the application of the attribute type in SDS *attribute_type_in_sds* associated with the attribute type *attribute_type* in the SDS *sds* from the type in SDS *type_in_sds* associated with the object or link type *type* in *sds*.

(3) The "in_attribute_set" link between *type_in_sds* and *attribute_type_in_sds* and its reverse "is_attribute_of" link are deleted.

(4) Write locks of the default mode are obtained on the deleted links.

Errors

(5) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(6) ACCESS_ERRORS (*type_in_sds*, ATOMIC, MODIFY, WRITE_LINKS)

(7) ACCESS_ERRORS (*attribute_type_in_sds*, ATOMIC, MODIFY, WRITE_LINKS)

(8) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(9) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(10) SDS_IS_PREDEFINED (*sds*)

(11) SDS_IS_UNKNOWN (*sds*)

(12) TYPE_IS_NOT_APPLIED (*sds*, *attribute_type*, *type*)

(13) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *attribute_type*)

(14) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.2.28 SDS_UNAPPLY_LINK_TYPE

(1)
```
SDS_UNAPPLY_LINK_TYPE (
    sds            : Sds_designator,
    link_type      : Link_type_nominator_in_sds,
    object_type    : Object_type_nominator_in_sds
)
```

(2) SDS_UNAPPLY_LINK_TYPE removes the application of the link type in SDS *link_type_in_sds* associated with the link type *link_type* in the SDS *sds* from the object type in SDS *object_type_in_sds* associated with the object type *object_type* in *sds*.

(3) The "in_link_set" link between *object_type_in_sds* and *link_type_in_sds* and its reverse "is_link_of" link are deleted.

(4) If *link_type* has a reverse link type *reverse*, then *object_type* is removed from the destination object types of *reverse* (see SDS_REMOVE_DESTINATION) and the "in_destination_set" link between the link type in SDS *reverse_link_type_in_sds* associated with *reverse* in *sds* and *object_type_in_sds* is deleted.

(5) Write locks of the default mode are obtained on the deleted links.

Errors

(6) ACCESS_ERRORS (*object_type_in_sds*, ATOMIC, MODIFY, WRITE_LINKS)

(7) ACCESS_ERRORS (*link_type_in_sds*, ATOMIC, MODIFY, WRITE_LINKS)

(8) ACCESS_ERRORS (*reverse_link_type_in_sds*, ATOMIC, MODIFY, WRITE_LINKS)

(9) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(10) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(11) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(12) SDS_IS_PREDEFINED (*sds*)

(13) SDS_IS_UNKNOWN (*sds*)

(14) TYPE_IS_NOT_APPLIED (*sds*, *link_type*, *object_type*)

(15) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *link_type*)

(16) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *object_type*)

10.3 SDS usage operations

10.3.1 SDS_GET_ATTRIBUTE_TYPE_PROPERTIES

(1)
```
SDS_GET_ATTRIBUTE_TYPE_PROPERTIES (
    sds            : Sds_designator,
    type           : Attribute_type_nominator_in_sds
)
    duplication    : Duplication,
    value_type     : Value_type,
    initial_value  : Attribute_value
```

(2) SDS_GET_ATTRIBUTE_TYPE_PROPERTIES returns the duplication, value type identifier, and initial value of the attribute type in SDS identified by *type* in the SDS *sds*.

(3) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with *type* in *sds*.

Errors

(4) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(5) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_ATTRIBUTES)

(6) SDS_IS_UNKNOWN (*sds*)

(7) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.3.2 SDS_GET_ENUMERAL_TYPE_IMAGE

(1)
```
SDS_GET_ENUMERAL_TYPE_IMAGE (
    sds     : Sds_designator,
    type    : Enumeral_type_nominator_in_sds
)
    image   : Text
```

(2) SDS_GET_ENUMERAL_TYPE_IMAGE returns the image *image* of the enumeral type in SDS identified by *type* in the SDS *sds*.

(3) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with *type* in *sds*.

Errors

(4) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(5) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_ATTRIBUTES)

(6) SDS_IS_UNKNOWN (*sds*)

(7) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.3.3 SDS_GET_ENUMERAL_TYPE_POSITION

(1)
```
SDS_GET_ENUMERAL_TYPE_POSITION (
    sds        : Sds_designator,
    type1      : Enumeral_type_nominator_in_sds,
    type2      : Attribute_type_nominator_in_sds
)
    position   : Natural
```

(2) SDS_GET_ENUMERAL_TYPE_POSITION returns the position *position* of the enumeral type in SDS identified by *type1* in the SDS *sds*, in the value type of the attribute type in SDS identified by *type2* in *sds*, i.e. the key of the "enumeral" link from the "type" object associated with *type2* to the "type" object associated with *type1*.

(3) Read locks of the default mode are obtained on the "type" and "type_in_sds" objects associated with *type1* and *type2* in *sds* and on the "enumeral" link.

Errors

(4) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(5) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type1* and *type2*, ATOMIC, READ, READ_ATTRIBUTES)

(6) ENUMERAL_TYPE_IS_NOT_IN_ATTRIBUTE_VALUE_TYPE (*type1*, *type2*)

(7) SDS_IS_UNKNOWN (*sds*)

(8) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type1*)

(9) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type2*)

10.3.4 SDS_GET_LINK_TYPE_PROPERTIES

(1)
```
SDS_GET_LINK_TYPE_PROPERTIES (
    sds            : Sds_designator,
    type           : Link_type_nominator_in_sds
)
    category       : Category,
    lower_bound    : Natural,
    upper_bound    : Natural,
    exclusiveness  : Exclusiveness,
    stability      : Stability,
    duplication    : Duplication,
    key_types      : Key_types,
    reverse        : [ Link_type_nominator_in_sds ]
```

(2) SDS_GET_LINK_TYPE_PROPERTIES returns the category, lower and upper bounds, exclusiveness, stability, duplication, key attribute types, and reverse link type (if any) of the link type in SDS identified by *type* in the SDS *sds*.

(3) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with *type* in *sds*.

Errors

(4) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(5) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_ATTRIBUTES)

(6) SDS_IS_UNKNOWN (*sds*)

(7) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.3.5 SDS_GET_OBJECT_TYPE_PROPERTIES

(1)
```
SDS_GET_OBJECT_TYPE_PROPERTIES (
    sds            : Sds_designator,
    type           : Object_type_nominator_in_sds
)
    contents_type  : [ Contents_type ],
    parents        : Object_type_nominators_in_sds,
    children       : Object_type_nominators_in_sds
```

(2) SDS_GET_OBJECT_TYPE_PROPERTIES returns the contents type, parents, and children of the object type in SDS identified by *type* in the SDS *sds*.

(3) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with *type* in *sds*.

Errors

(4) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(5) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, (READ_ATTRIBUTES, READ_LINKS))

(6) SDS_IS_UNKNOWN (*sds*)

(7) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.3.6 SDS_GET_TYPE_KIND

(1)
```
SDS_GET_TYPE_KIND (
    sds         : Sds_designator,
    type        : Type_nominator_in_sds
)
    type_kind   : Type_kind
```

(2) SDS_GET_TYPE_KIND returns the kind of the type in SDS identified by *type* in the SDS *sds*.

(3) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with *type* in *sds*.

Errors

(4) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(5) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_ATTRIBUTES)

(6) SDS_IS_UNKNOWN (*sds*)

(7) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.3.7 SDS_GET_TYPE_MODES

(1)
```
SDS_GET_TYPE_MODES (
    sds             : Sds_designator,
    type            : Object_type_nominator_in_sds | Attribute_type_nominator_in_sds |
                          Link_type_nominator_in_sds
)
    usage_mode      : Definition_mode_values,
    export_mode     : Definition_mode_values,
    max_usage_mode  : Definition_mode_values
```

(2) SDS_GET_TYPE_MODES returns in *usage_mode*, *export_mode*, and *max_usage_mode* the usage mode, export mode, and maximum usage mode, respectively, of the type in SDS identified by *type* in the SDS *sds*.

(3) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with *type* in *sds*.

Errors

(4) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(5) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_ATTRIBUTES)

(6) SDS_IS_UNKNOWN (*sds*)

(7) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.3.8 SDS_GET_TYPE_NAME

(1)
```
SDS_GET_TYPE_NAME (
    sds      : Sds_designator,
    type     : Type_nominator_in_sds
)
    name     : [ Name ]
```

(2) SDS_GET_TYPE_NAME returns the full type name *name* of the type in SDS identified by *type* in the SDS *sds*.

(3) If no name is associated with *type* in *sds* no value is returned.

(4) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with *type* in *sds*.

Errors

(5) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(6) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_ATTRIBUTES)

(7) SDS_IS_UNKNOWN (*sds*)

(8) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.3.9 SDS_SCAN_ATTRIBUTE_TYPE

(1)
```
SDS_SCAN_ATTRIBUTE_TYPE (
    sds             : Sds_designator,
    type            : Attribute_type_nominator_in_sds,
    scanning_kind   : Attribute_scan_kind
)
    types           : Object_type_nominators_in_sds | Link_type_nominators_in_sds
```

(2) SDS_SCAN_ATTRIBUTE_TYPE returns a set of types *types* determined by *type*, *sds* and *scanning_kind.*

(3) The returned set of types is determined as follows. It is limited to types associated with types in SDS in the SDS *sds* and by *scanning_kind* as follows.

(4) - OBJECT: object types to which *type* has been applied by means of SDS_APPLY_ATTRIBUTE_TYPE.

(5) - OBJECT_ALL: object types of which *type* is an attribute type, i.e. the union of the object types defined by OBJECT and all their descendants.

(6) - LINK_KEY: link types of which the *type* is a key attribute type.

(7) - LINK_NON_KEY: link types of which the *type* is a non-key attribute type.

(8) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with the returned types in SDS.

Errors

(9) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(10) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_ATTRIBUTES)

(11) SDS_IS_UNKNOWN (*sds*)

(12) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

10.3.10 SDS_SCAN_ENUMERAL_TYPE

(1)
```
SDS_SCAN_ENUMERAL_TYPE (
    sds         : Sds_designator,
    type        : Enumeral_type_nominator_in_sds
)
    types       : Attribute_type_nominators_in_sds
```

(2) SDS_SCAN_ENUMERAL_TYPE returns a set of enumeration attribute types, determined by *sds* and the enumeral type *type*.

(3) The returned set of types is limited to types with an associated type in SDS *type_in_sds* in the SDS *sds* and to enumeration attribute types with value type containing *type*.

(4) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with the returned types in SDS.

Errors

(5) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(6) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_LINKS)

(7) SDS_IS_UNKNOWN (*sds*)

(8) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

(9) The following implementation-defined error may be raised:
ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *types*, ATOMIC, READ, READ_LINKS)

10.3.11 SDS_SCAN_LINK_TYPE

(1)
```
SDS_SCAN_LINK_TYPE (
    sds            : Sds_designator,
    type           : Link_type_nominator_in_sds,
    scanning_kind  : Link_scan_kind
)
    types          : Object_type_nominators_in_sds | Attribute_type_nominators_in_sds
```

(2) SDS_SCAN_LINK_TYPE returns a set of attribute or object types *types* determined by *sds, type*, and *scanning_kind.*

(3) The returned set of types is determined as follows. It is limited to types with an associated type in SDS *type_in_sds* in the SDS *sds* and by *scanning_kind* as follows.

(4) - ORIGIN: object types which have been defined as origin types of *type* by SDS_APPLY_LINK_TYPE or by SDS_ADD_DESTINATION on the reverse link type.

(5) - ORIGIN_ALL: object types which are valid origins of *type*, i.e. the object types as specified by *scanning_kind* = ORIGIN, plus all their descendants.

(6) - DESTINATION: object types which have been defined as destination types of *type* by SDS_ADD_DESTINATION or by SDS_APPLY_LINK_TYPE on the reverse link type.

(7) - DESTINATION_ALL: object types which are valid destinations of *type*, i.e. the object types as specified by *scanning_kind* = DESTINATION plus all their descendants.

(8) - KEY: key attribute types of *type*.

(9) - NON_KEY: non-key attribute types of *type*, i.e. attribute types which have been applied to *type* by SDS_APPLY_ATTRIBUTE_TYPE.

(10) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with the returned types in SDS.

Errors

(11) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(12) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_LINKS)

(13) SDS_IS_UNKNOWN (*sds*)

(14) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

(15) The following implementation-defined error may be raised:
ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *types*, ATOMIC, READ, READ_LINKS)

10.3.12 SDS_SCAN_OBJECT_TYPE

(1)
```
SDS_SCAN_OBJECT_TYPE (
    sds            : Sds_designator,
    type           : Object_type_nominator_in_sds,
    scanning_kind  : Object_scan_kind
)
    types          : Object_type_nominators_in_sds | Attribute_type_nominators_in_sds |
                         Link_type_nominators_in_sds
```

(2) SDS_SCAN_OBJECT_TYPE returns a set of types *types* determined by *object_type*, *sds* and *scanning_kind.*

(3) The returned set of types is determined as follows. It is limited to types with associated types in SDS *type_in_sds* in the SDS *sds* and by *scanning_kind* as follows.

(4) - CHILD: object types which are children of *object_type*.

(5) - DESCENDANT: object types which are descendants of *object_type*.

(6) - PARENT: object types which are parents of *object_type*.

(7) - ANCESTOR: object types which are ancestors of *object_type*.

(8) - ATTRIBUTE: attribute types which have been applied to *object_type* by SDS_APPLY_ATTRIBUTE_TYPE.

(9) - ATTRIBUTE_ALL: attribute types of *object_type*, i.e. attribute types which have been applied to *object_type* or to the ancestors of *object_type*.

(10) - LINK_ORIGIN: link types which have been applied to *object_type* (*object_type* becoming its origin type) by SDS_APPLY_LINK_TYPE or by SDS_ADD_DESTINATION on the reverse link type.

(11) - LINK_ORIGIN_ALL: link types which have *object_type* as an origin type, i.e. link types which have been applied to *object_type* or to its ancestors.

(12) - LINK_DESTINATION: link types whose destination set has been extended to include *object_type* by SDS_ADD_DESTINATION or by SDS_APPLY_LINK_TYPE on the reverse link type.

(13) - LINK_DESTINATION_ALL: link types which have *object_type* as a destination type, i.e. link types which have been added to the destination set of *object_type* or to its ancestors.

(14) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with the returned types in SDS.

Errors

(15) ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)

(16) ACCESS_ERRORS ("type" and "type_in_sds" objects associated with *type*, ATOMIC, READ, READ_LINKS)

(17) SDS_IS_UNKNOWN (*sds*)

(18) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *object_type*)

10.3.13 SDS_SCAN_TYPES

(1)
```
SDS_SCAN_TYPES (
    sds     : Sds_designator,
    kind    : [ Type_kind ]
)
    types   : Type_nominators_in_sds
```

(2) SDS_SCAN_TYPES returns all the types with associated types in SDS in the SDS *sds*, of the kinds given by *kind*.

(3) If *kind* is not supplied, all such types are returned; otherwise all such object types, attribute types, link types, or enumeral types are returned according as *kind* is OBJECT_TYPE, ATTRIBUTE_TYPE, LINK_TYPE, or ENUMERAL_TYPE respectively.

(4) Read locks of the default mode are obtained on *sds* and on the "type" and "type_in_sds" objects associated with *type*.

Errors

(5) ACCESS_ERRORS (*sds*, COMPOSITE, READ, READ_LINKS)

(6) SDS_IS_UNKNOWN (*sds*)

10.4 Working schema operations

10.4.1 WS_GET_ATTRIBUTE_TYPE_PROPERTIES

(1)
```
WS_GET_ATTRIBUTE_TYPE_PROPERTIES (
    type            : Attribute_type_nominator
)
    duplication     : Duplication,
    value_type      : Value_type,
    initial_value   : Attribute_value
```

(2) WS_GET_ATTRIBUTE_TYPE_PROPERTIES returns the duplication, value type identifier, and initial value of the attribute type in working schema associated with the type *type* in the current working schema.

Errors

(3) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.2 WS_GET_ENUMERAL_TYPE_IMAGE

(1)
```
WS_GET_ENUMERAL_TYPE_IMAGE (
    type    : Enumeral_type_nominator
)
    image  : Text
```

(2) WS_GET_ENUMERAL_TYPE_IMAGE returns the image *image* of the type in working schema associated with the enumeral type *type* in the current working schema.

Errors

(3) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.3 WS_GET_ENUMERAL_TYPE_POSITION

(1)
```
WS_GET_ENUMERAL_TYPE_POSITION (
    type1       : Enumeral_type_nominator,
    type2       : Attribute_type_nominator
)
    position    : Natural
```

(2) WS_GET_ENUMERAL_TYPE_POSITION returns the position *position* of the enumeral type in working schema associated with *type1* in the value type of the attribute type in working schema associated with *type2*, i.e. the key of the "enumeral" link from *type2* to *type1*.

Errors

(3) ENUMERAL_TYPE_IS_NOT_IN_ATTRIBUTE_VALUE_TYPE (*type1*, *type2*)

(4) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type1*)

(5) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type2*)

10.4.4 WS_GET_LINK_TYPE_PROPERTIES

(1)
```
WS_GET_LINK_TYPE_PROPERTIES (
    type            : Link_type_nominator
)
    category        : Category,
    lower_bound     : Natural,
    upper_bound     : Natural,
    exclusiveness   : Exclusiveness,
    stability       : Stability,
    duplication     : Duplication,
    key_types       : Key_types,
    reverse         : [ Link_type_nominator ]
```

(2) WS_GET_LINK_TYPE_PROPERTIES returns the category, lower and upper bounds, exclusiveness, stability, duplication, key types, and reverse link type (if any) of the link type in working schema associated with the type *type*.

Errors

(3) ACCESS_ERRORS (*type*, ATOMIC, READ, READ_ATTRIBUTES)

(4) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.5 WS_GET_OBJECT_TYPE_PROPERTIES

(1)
```
WS_GET_OBJECT_TYPE_PROPERTIES (
    type            : Object_type_nominator
)
    contents_type   : [ Contents_type ],
    parents         : Object_type_nominators,
    children        : Object_type_nominators
```

(2) WS_GET_OBJECT_TYPE_PROPERTIES returns the contents type, parents, and children of the object type in working schema associated with the type *type*.

Errors

(3) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.6 WS_GET_TYPE_KIND

(1)
```
WS_GET_TYPE_KIND (
type        : Type_nominator
)
type_kind   : Type_kind
```

(2) WS_GET_TYPE_KIND returns the kind of the type in working schema associated with the type *type* in the current working schema.

Errors

(3) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.7 WS_GET_TYPE_MODES

(1)
```
WS_GET_TYPE_MODES (
    type            : Type_nominator
)
    usage_mode      : Definition_mode_values
```

(2) WS_GET_TYPE_MODES returns the usage mode of the type in working schema associated with the type *type* in the current working schema.

Errors

(3) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.8 WS_GET_TYPE_NAME

(1)
```
WS_GET_TYPE_NAME (
    type    : Type_nominator
)
    name    : [ Name ]
```

(2) WS_GET_TYPE_NAME returns the first non-null composite name *name* of the type in working schema, considering the sequence of SDSs in the working schema, associated with the type *type* in the current working schema.

(3) If no name is associated with *type* in the current working schema, no value is returned.

Errors

(4) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.9 WS_SCAN_ATTRIBUTE_TYPE

(1)
```
WS_SCAN_ATTRIBUTE_TYPE (
    type          : Attribute_type_nominator,
    scanning_kind : Attribute_scan_kind
)
    types         : Object_type_nominators | Link_type_nominators
```

(2) WS_SCAN_ATTRIBUTE_TYPE returns a set of types *types* determined by *type* and *scanning_kind.*

(3) The returned set of types is determined as follows. It is limited to types associated with types in working schema in the working schema of the calling process, and by *scanning_kind* as follows.

(4) - OBJECT: object types to which *type* has been applied by means of SDS_APPLY_ATTRIBUTE_TYPE.

(5) - OBJECT_ALL: object types of which *type* is an attribute type, i.e. the union of the object types defined by OBJECT and all their descendants.

(6) - LINK_KEY: link types of which the *type* is a key attribute type.

(7) - LINK_NON_KEY: link types of which the *type* is a non-key attribute type.

Errors

(8) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.10 WS_SCAN_ENUMERAL_TYPE

(1)
```
WS_SCAN_ENUMERAL_TYPE (
    type   : Enumeral_type_nominator
)
    types  : Attribute_type_nominators
```

(2) WS_SCAN_ENUMERAL_TYPE returns a set of enumeration attribute types, determined by the enumeral type *type*.

(3) The returned set of types is limited to types with an associated type in working schema in the working schema of the calling process, and to enumeration attribute types with value type containing *type.*

Errors

(4) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.11 WS_SCAN_LINK_TYPE

(1)
```
WS_SCAN_LINK_TYPE (
    type          : Link_type_nominator,
    scanning_kind : Link_scan_kind
)
    types         : Object_type_nominators | Attribute_type_nominators
```

(2) WS_SCAN_LINK_TYPE returns a set of attribute or object types *types* determined by *type* and *scanning_kind.*

(3) The returned set of types is determined as follows. It is limited to types with an associated type in working schema in the working schema of the calling process, and by *scanning_kind* as follows.

(4) - ORIGIN: object types which have been defined as origin types of *type* by SDS_APPLY_LINK_TYPE or by SDS_ADD_DESTINATION on the reverse link type.

(5) - ORIGIN_ALL: object types which are valid origins of *type*, i.e. the object types as specified by *scanning_kind* = ORIGIN plus all their descendants.

(6) - DESTINATION: object types which have been defined as destination types of *type* by SDS_ADD_DESTINATION or by SDS_APPLY_LINK_TYPE on the reverse link type.

(7) - DESTINATION_ALL: object types which are valid destinations of *type*, i.e. the object types as specified by *scanning_kind* = DESTINATION plus all their descendants.

(8) - KEY: key attribute types of *type*.

(9) - NON_KEY: non-key attributes of *type*, i.e. attribute types which have been applied to *type* by SDS_APPLY_ATTRIBUTE_TYPE.

Errors

(10) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*type*)

10.4.12 WS_SCAN_OBJECT_TYPE

(1)
```
WS_SCAN_OBJECT_TYPE (
    object_type        : Object_type_nominator,
    scanning_kind      : Object_scan_kind
)
    types              : Object_type_nominators | Attribute_type_nominators |
                           Link_type_nominators
```

(2) WS_SCAN_OBJECT_TYPE returns a set of types *types* determined by *object_type* and *scanning_kind.*

(3) The returned set of types is determined as follows. It is limited to types with associated types in working schema in the working schema of the calling process, and by *scanning_kind* as follows.

(4) - CHILD: object types which are children of *object_type*.

(5) - DESCENDANT: object types which are descendants of *object_type*.

(6) - PARENT: object types which are parents of *object_type*.

(7) - ANCESTOR: object types which are ancestors of *object_type*.

(8) - ATTRIBUTE: attribute types which have been applied to *object_type* by SDS_APPLY_ATTRIBUTE_TYPE.

(9) - ATTRIBUTE_ALL: attribute types of *object_type*, i.e. attribute types which have been applied to *object_type* or to the ancestors of *object_type*.

(10) - LINK_ORIGIN: link types which have been applied to *object_type* (*object_type* becoming its origin type) by SDS_APPLY_LINK_TYPE or by SDS_ADD_DESTINATION on the reverse link type.

(11) - LINK_ORIGIN_ALL: link types which have *object_type* as an origin type, i.e. link types which have been applied to *object_type* or to its ancestors.

(12) - LINK_DESTINATION: link types which have had *object_type* added to their destination object types by SDS_ADD_DESTINATION or by SDS_APPLY_LINK_TYPE on the reverse link type.

(13) - LINK_DESTINATION_ALL: link types which have had *object_type* or to its ancestors added to their destination object types.

Errors

(14) TYPE_IS_UNKNOWN_IN_WORKING_SCHEMA (*object_type*)

10.4.13 WS_SCAN_TYPES

(1)
```
WS_SCAN_TYPES (
    kind    : [ Type_kind ]
)
    types   : Type_nominators
```

(2) WS_SCAN_TYPES returns all the types of the type kind given by *kind* with associated types in working schema in the current working schema.

(3) If *kind* is not supplied, all such types are returned; otherwise all such object types, attribute types, link types, or enumeral types are returned according as *kind* is OBJECT_TYPE, ATTRIBUTE_TYPE, LINK_TYPE, or ENUMERAL_TYPE respectively.

Errors

(4) None.

11 Volumes, devices, and archives

11.1 Volume, device, and archiving concepts

11.1.1 Volumes

(1)
```
Volume_identifier = Natural
```

(2)
```
Volume_accessibility = ACCESSIBLE | INACCESSIBLE  | UNKNOWN
```

(3)
```
Volume_info = Volume_identifier * Volume_accessibility
```

(4)
```
Volume_infos = set of Volume_info
```

(5)
```
Volume_status ::
    TOTAL_BLOCKS          : Natural
    FREE_BLOCKS           : Natural
    BLOCK_SIZE            : Natural
    NUM_OBJECTS           : Natural
    VOLUME_IDENTIFIER     : Volume_identifier
```

(6)
```
sds system:
```

(7)
```
volume_directory: child type of object with
link
    known_volume: (navigate) non_duplicated existence link (volume_identifier) to
        volume;
    volumes_of: implicit link to common_root reverse volumes;
end volume_directory;
```

(8)
```
volume: child type of object with
attribute
    volume_characteristics: (read) string;
link
    object_on_volume: (navigate) non_duplicated designation link (exact_identifier) to
        object;
    mounted_on: (navigate) non_duplicated designation link to
        device_supporting_volume with
    attribute
        read_only: (read) boolean;
    end mounted_on;
end volume;
```

(9)
```
end system;
```

(10) The volume directory is an administrative object (see 9.1.2); it represents the set of known volumes, each with a unique volume identifier which is assigned to the volume on creation and uniquely identifies the volume within the PCTE installation.

(11) The destinations of the "object_on_volume" links from a volume are called the objects *residing on* that volume. The value of the "exact_identifier" attribute is the exact identifier of the object (see 9.1.1). The volume is *mounted* if there is a "mounted_on" link; the destination of the link is the device that the volume is mounted on (see 11.1.3). The "read_only" attribute indicates that the volume may not be written to (except for usage designation links, see 8.3.3). A known volume resides on itself; it is the only known volume residing on a volume and it is the first object created on that volume.

(12) The "volume_characteristics" attribute is an implementation-defined string specifying implementation-dependent characteristics of the volume.

11.1.2 Administration volumes

(1)
```
sds system:
```

(2)
```
administration_volume: (protected) child type of volume with
link
    administration_volume_of: non_duplicated designation link (number) to workstation;
end administration_volume;
```

(3)
```
end system;
```

(4) Each administration volume is either the master volume or a copy volume of the administration replica set. See 17.1.4.

(5) Each administration volume is associated with one or more workstations (the destinations of the "administration_volume_of" links), and is mounted on a device controlled by one of them.

(6) There is exactly one master administration volume in a PCTE installation. It is the master volume of the administration replica set (see 17.1.4), and has volume identifier 0. The master administration volume is part of the initial value of the state (see 8.1).

11.1.3 Devices

(1)
```
Device_identifier = Natural
```

(2)
```
sds system:
```

(3)
```
device_supporting_volume: child type of device with
link
    mounted_volume: (navigate) non_duplicated designation link to volume;
end device_supporting_volume;
```

(4)
```
end system;
```

(5) A device supporting volume is a device (see 12.1) that may have an associated volume, the destination of the "mounted_volume" link, called the volume *mounted on* the device.

(6) A device supporting volume resides on the administration volume of the workstation controlling it.

11.1.4 Archives

(1)
```
Archive_selection = Object_designators | ALL
```

(2)
```
Archive_status = PARTIAL | COMPLETE
```

(3)
```
sds system:
```

(4)
```
archive_directory: child type of object with
link
    saved_archive: (navigate) non_duplicated existence link (archive_identifier: natural)
        to archive;
    archives_of: implicit link to common_root reverse archives;
end archive_directory;
```

(5)
```
archive: child type of object with
attribute
    archiving_time: (read) time;
link
    archived_object: (navigate) non_duplicated designation link (exact_identifier) to
        object;
end archive;
```

(6)
```
end system;
```

(7) The archive directory is an administrative object (see 9.1.2); it represents the set of known archives (the destinations of the "saved_archive" links), each with a unique archive identifier which is assigned to the archive on creation and uniquely identifies the archive within the PCTE installation.

(8) An archive consists of a set of objects (the destinations of the "archived_object" links), called the objects *archived on* the archive.

(9) The archiving time of an archive is the system time at which objects are saved in the archive. An archive may only be used once to save objects in it.

11.2 Volume, device, and archive operations

11.2.1 ARCHIVE_CREATE

(1)
```
ARCHIVE_CREATE (
    archive_identifier   : Natural,
    on_same_volume_as    : Object_designator,
    access_mask          : Atomic_access_rights,
)
    new_archive          : Archive_designator
```

(2) ARCHIVE_CREATE creates a new archive *new_archive* residing on the same volume as the object *on_same_volume_as*.

(3) A new "known_archive" link with key *archive_identifier* is created from the archive directory to *new_archive*.

(4) An "object_on_volume" link is created from the volume on which *on_same_volume_as* resides to *new_archive*. The key of the link is the exact identifier of *new_archive*. *access_mask* is used in conjunction with the default atomic ACL and default object owner of the calling process to define the atomic ACL and the composite ACL which are to be associated with the created object (see 19.1.4).

(5) The labels of *new_archive* are set to the mandatory context of the calling process.

(6) Write locks of the default kind are obtained on *new_archive* and the new "known_archive" link.

Errors

(7) ACCESS_ERRORS (the archive directory, ATOMIC, MODIFY, APPEND_LINKS)

(8) ARCHIVE_EXISTS (*archive_identifier*)

(9) CONTROL_WOULD_NOT_BE_GRANTED (*new_archive*)

(10) LABEL_IS_OUTSIDE_RANGE (*new_archive*, volume on which *on_same_volume_as* resides)

(11) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(12) REFERENCE_CANNOT_BE_ALLOCATED

(13) VOLUME_IS_FULL (volume on which *on_same_volume_as* resides)

11.2.2 ARCHIVE_REMOVE

(1)
```
ARCHIVE_REMOVE (
    archive    : Archive_designator
)
```

(2) ARCHIVE_REMOVE removes the archive *archive* from the archive directory by deleting the "known_archive" link to *archive* from the archive directory.

(3) Write locks of the default kind are obtained on *archive* and the deleted "known_archive" link.

Errors

(4) ACCESS_ERRORS (*archive*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(5) ACCESS_ERRORS (*archive*, ATOMIC, MODIFY, DELETE)

(6) ACCESS_ERRORS (*archive*, ATOMIC, MODIFY, WRITE_LINKS)

(7) ACCESS_ERRORS (the archive directory, ATOMIC, MODIFY, WRITE_LINKS)

(8) ARCHIVE_HAS_ARCHIVED_OBJECTS (*archive*)

(9) ARCHIVE_IS_UNKNOWN (*archive*)

(10) OBJECT_IS_IN_USE_FOR_DELETE (*archive*)

(11) OBJECT_IS_INACCESSIBLE (*archive*, ATOMIC)

(12) If the conditions hold for deletion of the "archive" object *archive*:
PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

11.2.3 ARCHIVE_RESTORE

(1)
```
ARCHIVE_RESTORE (
    device              : Device_designator,
    archive             : Archive_designator,
    scope               : Archive_selection,
    on_same_volume_as   : Object_designator
)
    restoring_status    : Archive_status
```

(2) ARCHIVE_RESTORE restores a set of objects *objects* specified by *scope* to the volume *volume* on which *on_same_volume_as* resides from the archive *archive.*

(3) If *scope* is a set of object designators, the specified set of objects to be restored (called the 'specified set' in this clause) is the intersection of the set of objects archived on *archive* and the set of objects in *scope.*

(4) If *scope* is ALL, the specified set is the set of all the objects archived on *archive*.

(5) The objects to be restored are taken from the contents of *device.*

(6) The objects and their components are moved to *volume*, in an undefined order; as many objects and their components as possible are restored from *device,* and reside on *volume*.

(7) If not all the objects of the specified set can be restored by this operation, as many as possible are restored and *restoring_status* is set to PARTIAL. If all the objects of the specified set are restored, *restoring_status* is set to COMPLETE. If no objects of the specified set can be restored, the error condition VOLUME_IS_FULL is raised.

(8) The "archived_object" links from *archive* to the restored objects are deleted.

(9) For each of the objects which are restored to *volume*, an "object_on_volume" link with key the exact identifier of the object is created.

(10) If any of the objects specified to be restored has not been archived or is already restored on *volume,* then it is not affected.

(11) Write locks of the default mode are obtained on *archive* and on the moved objects and links. A read lock of the default mode is obtained on *device*.

Errors

(12) ACCESS_ERRORS (*device*, ATOMIC, READ, READ_CONTENTS)

(13) ACCESS_ERRORS (an element of *scope*, COMPOSITE, CHANGE, CONTROL_OBJECT)

(14) ARCHIVE_IS_INVALID_ON_DEVICE (*device*, *archive*)

(15) LABEL_IS_OUTSIDE_RANGE (an element of the specified set or a component of such an element, *volume*)

(16) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(17) PROCESS_IS_IN_TRANSACTION

(18) VOLUME_IS_FULL (*volume*)

(19) VOLUME_IS_INACCESSIBLE (*volume*)

(20) VOLUME_IS_READ_ONLY (*scope*, COMPOSITE)

(21) The following implementation-dependent errors may be raised for any object X with a link to an object of *objects*:
OBJECT_IS_INACCESSIBLY_ARCHIVED (X)
VOLUME_IS_INACCESSIBLE (volume on which X resides)
VOLUME_IS_READ_ONLY (volume on which X resides)

11.2.4 ARCHIVE_SAVE

(1)
```
ARCHIVE_SAVE (
    device              : Device_designator,
    archive             : Archive_designator,
    objects             : Object_designators
)
    archiving_status    : Archive_status
```

(2) ARCHIVE_SAVE moves a set of objects to the contents of *device*.

(3) The archive *archive* is updated as follows:

(4) - the archiving time is set to the current system time;

(5) - "archived_object" links are created from *archive* to the archived objects and to each of their components. The keys of the created links are the suffixes of the exact identifier of the destination objects.

(6) For each archived object, the "object_on_volume" link from the volume on which the object resides to the object is deleted.

(7) If *device* has insufficient space to hold all the objects of *objects* and their components, the operation archives as many objects as possible and *archiving_status* is set to PARTIAL. If *device* has insufficient space to hold any objects of *objects* with their components, the error condition DEVICE_SPACE_IS_FULL occurs. Otherwise *archiving_status* is set to COMPLETE.

(8) The operation has no effect on objects or components which are already archived, either on the same archive or on another one.

(9) Read locks of the default mode are obtained on the objects to be archived. Write locks of the default mode are obtained on *archive* and on *device*.

Errors

(10) ACCESS_ERRORS (*device*, ATOMIC, MODIFY, WRITE_CONTENTS)

(11) ACCESS_ERRORS (*archive*, ATOMIC, MODIFY, APPEND_LINKS)

(12) ACCESS_ERRORS (elements of *objects* and their components that are to be archived, ATOMIC, CHANGE, CONTROL_OBJECT)

(13) ARCHIVE_HAS_ARCHIVED_OBJECTS (*archive*)

(14) DEVICE_SPACE_IS_FULL (*device*)

(15) LABEL_IS_OUTSIDE_RANGE (an element or a component of an element of *objects*, *device*)

(16) OBJECT_ARCHIVING_IS_INVALID (*objects*)

(17) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(18) PROCESS_IS_IN_TRANSACTION

(19) The following implementation-dependent errors may be raised for any object X with a link to an object of *objects*:
OBJECT_IS_INACCESSIBLY_ARCHIVED (X)
VOLUME_IS_INACCESSIBLE (volume on which X resides)
VOLUME_IS_READ_ONLY (volume on which X resides)

(20) NOTE - It is intended that the space previously occupied by the archived objects be freed.

11.2.5 DEVICE_CREATE

(1)
```
DEVICE_CREATE (
    station                 : Workstation_designator,
    device_type             : Device_type_nominator,
    access_mask             : Atomic_access_rights,
    device_identifier       : Natural,
    device_characteristics  : String
)
    new_device              : Device_designator
```

(2) DEVICE_CREATE creates a device *new_device* of type *device_type* with a "controlled_device" link to it from *station*. The value of *device_identifier* is the key of the created link. The "device_of" reverse link created from the new object to *station* designates the workstation which controls the device. The "device_characteristics" attribute of *new_device* is set to *device_characteristics*.

(3) *device_identifier* is a value which uniquely identifies the new device within the devices controlled by *station*. Its value is assigned to the "device_identifier" attribute of *new_device*.

(4) *new_device* resides on the same volume as *station* (i.e. the local administration volume of *station*) and cannot be moved to another volume.

(5) *access_mask* is used in conjunction with the default atomic ACL and default object owner of the calling process to define both the atomic ACL and the composite ACL which are to be associated with the created object (see 19.1.4).

(6) An "object_on_volume" link is created from the administration volume of *station* to *new_device*. The created link is keyed by the exact identifier of *new_device*.

(7) The security labels of *new_device* and the labels defining its security ranges are set to the mandatory context of the calling process.

(8) Write locks (of the default kind) are obtained on *new_device* and on the new links (except the new "object_on_volume" link).

Errors

(9) ACCESS_ERRORS (*station*, ATOMIC, MODIFY, APPEND_LINKS)

(10) CONTROL_WOULD_NOT_BE_GRANTED (*new_device*)

(11) DEVICE_CHARACTERISTICS_ARE_INVALID (*device-characteristics*)

(12) DEVICE_EXISTS (*device_identifier*)

(13) LABEL_IS_OUTSIDE_RANGE (*new_device*, *station*)

(14) LIMIT_WOULD_BE_EXCEEDED (MAX_KEY_VALUE)

(15) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL

(16) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(17) REFERENCE_CANNOT_BE_ALLOCATED

(18) USAGE_MODE_ON_OBJECT_TYPE_WOULD_BE_VIOLATED ("object", *device_type*)

(19) WORKSTATION_IS_UNKNOWN (*station*)

11.2.6 DEVICE_REMOVE

(1)
```
DEVICE_REMOVE (
    device  : Device_designator
)
```

(2) DEVICE_REMOVE removes the device object *device* from the set of devices of a workstation. As a result, the device object *device* does not represent a physical device and its associated device identifier can be reused.

(3) The "controlled_device" link from the workstation to *device* is deleted. If it is the only existence link to *device* and there are no composition links to *device*, *device* is also deleted. In that case, the "object_on_volume" link from the volume on which *device* was residing to *device* is also deleted.

(4) A write lock (of the default kind) is obtained on *device* if it is deleted and on the deleted links (except the "object_on_volume" link).

Errors

(5) ACCESS_ERRORS (*device*, ATOMIC, MODIFY, WRITE_LINKS)

(6) ACCESS_ERRORS (*station*, ATOMIC, MODIFY, WRITE_LINKS)

(7) If conditions hold for the deletion of *device*:

(8) ACCESS_ERRORS (*device*, COMPOSITE, MODIFY, DELETE)

(9) For each origin X of an implicit link to *device*:
ACCESS_ERRORS (X, ATOMIC, CHANGE, WRITE_IMPLICIT)

(10) For each atomically stabilizing link L of *device*:
ACCESS_ERRORS (destination of L, ATOMIC, CHANGE, STABILIZE)

(11) For each compositely stabilizing link L of *device*:
ACCESS_ERRORS (destination of L, COMPOSITE, CHANGE, STABILIZE)

(12) DEVICE_IS_IN_USE (*device*)

(13) DEVICE_IS_UNKNOWN (*device*)

(14) If conditions hold for the deletion of *device*:
OBJECT_HAS_LINKS_PREVENTING_DELETION (*device*)
OBJECT_IS_IN_USE_FOR_DELETE (*device*)

(15) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(16) NOTE - This operation prevents further use of the device.

11.2.7 LINK_GET_DESTINATION_ARCHIVE

(1)
```
LINK_GET_DESTINATION_ARCHIVE (
    origin              : Object_designator,
    link                : Link_designator
)
    archive_identifier  : Archive_identifier
```

(2) LINK_GET_DESTINATION_ARCHIVE returns the archive identifier of the destination object of the direct outgoing link *link* of the object *origin*.

(3) A read lock of default mode is obtained on *link*.

Errors

(4) ACCESS_ERRORS (*origin*, ATOMIC, READ, READ_LINKS)

(5) OBJECT_IS_NOT_ARCHIVED (destination object of *link*)

11.2.8 VOLUME_CREATE

(1)
```
VOLUME_CREATE (
    device                    : Device_supporting_volume_designator,
    volume_identifier         : Natural,
    access_mask               : Atomic_access_rights,
    volume_characteristics    : String
)
    new_volume                : Volume_designator
```

(2) VOLUME_CREATE creates a new volume *new_volume* and mounts it on the device *device*. *new_volume* resides on itself.

(3) A new "known_volume" link with key *volume_identifier* is created from the master of the volume directory to *new_volume*.

(4) A "mounted_on" link with "read_only" attribute set to **false** and its reverse are created between *new_volume* and *device*.

(5) An "object_on_volume" link is created from *new_volume* to itself. The key of the link is the exact identifier of *new_volume*. *access_mask* is used in conjunction with the default atomic ACL and default object owner of the calling process to define the atomic ACL and the composite ACL which are to be associated with the created object (see 19.1.4).

(6) The labels of the volume and the labels defining its security ranges are set to the mandatory context of the calling process. Each security range of the created volume must lie within the corresponding security range of *device* (see 20.1.5).

(7) The "volume_characteristics" attribute of *new_volume* is set to *volume_characteristics*.

(8) Write locks of the default mode are obtained on *new_volume* and the new links (except the new "object_on_volume" link).

Errors

(9) ACCESS_ERRORS (*device*, ATOMIC, MODIFY, EXPLOIT_DEVICE)

(10) ACCESS_ERRORS (the directory of volumes, ATOMIC, MODIFY, APPEND_LINKS)

(11) CONTROL_WOULD_NOT_BE_GRANTED (*new_version*)

(12) DEVICE_IS_BUSY (*device*, *volume_identifier*)

(13) DEVICE_IS_UNKNOWN (*device*)

(14) LABEL_IS_OUTSIDE_RANGE (*new_volume*, *device*)

(15) LIMIT_WOULD_BE_EXCEEDED (MAX_KEY_VALUE)

(16) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL

(17) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(18) PROCESS_IS_IN_TRANSACTION

(19) REFERENCE_CANNOT_BE_ALLOCATED

(20) VOLUME_EXISTS (*volume_identifier*)

(21) NOTE - The new volume may need to be initialized by a system tool before this operation is called.

11.2.9 VOLUME_DELETE

(1)
```
VOLUME_DELETE (
    volume : Volume_designator
)
```

(2) VOLUME_DELETE unmounts the volume *volume*, and deletes the "known_volume" link to *volume* from the master of the volume directory and the "mounted_volume" link from the device on which *volume* is mounted.

(3) *volume* must be the only object residing on *volume*, and there must be only the following three links from *volume*:

(4) - the reverse link of the "known_volume" link to *volume* from the volume directory;

(5) - the "object_on_volume" link from *volume* to itself;

(6) - the "mounted_on" link from *volume*.

(7) Write locks (of the default kind) are obtained on *volume* and the deleted links (except the "object_on_volume" link); however the locks on *volume* and on the links from *volume* do not prevent the unmounting of the volume.

Errors

(8) ACCESS_ERRORS (*device*, ATOMIC, MODIFY, EXPLOIT_DEVICE)

(9) ACCESS_ERRORS (the volume directory, ATOMIC, MODIFY, WRITE_LINKS)

(10) ACCESS_ERRORS (*volume*, ATOMIC, MODIFY, WRITE_LINKS)

(11) ACCESS_ERRORS (*volume*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(12) If the conditions hold for deletion of the "volume" object *volume*:
ACCESS_ERRORS (*volume*, ATOMIC, MODIFY, DELETE)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(14) PROCESS_IS_IN_TRANSACTION

(15) VOLUME_HAS_OTHER_LINKS (*volume*)

(16) VOLUME_HAS_OTHER_OBJECTS (*volume*)

(17) VOLUME_IS_INACCESSIBLE (*volume*)

(18) VOLUME_IS_UNKNOWN (*volume*)

11.2.10 VOLUME_GET_STATUS

(1)
```
VOLUME_GET_STATUS (
    volume      : Volume_designator
)
    status      : Volume_status
```

(2) VOLUME_GET_STATUS returns information about the mounted volume *volume*, as follows.

(3) - TOTAL_BLOCKS is the total number of blocks of data available on *volume*.

(4) - FREE_BLOCKS is the number of blocks of data which are free on *volume*.

(5) - BLOCK_SIZE is the size of a block of data on *volume*, in octets.

(6) - NUM_OBJECTS is the number of objects currently residing on *volume*.

(7) - VOLUME_IDENTIFIER is the volume identifier of *volume*.

(8) A read lock of the default mode is obtained on *volume*.

Errors

(9) ACCESS_ERRORS (*volume*, ATOMIC, READ, READ_ATTRIBUTES)

(10) VOLUME_IS_INACCESSIBLE (*volume*)

(11) VOLUME_IS_UNKNOWN (*volume*)

11.2.11 VOLUME_MOUNT

(1)
```
VOLUME_MOUNT (
    device             : Device_supporting_volume_designator,
    volume_identifier  : Volume_identifier,
    read_only          : Boolean
)
```

(2) VOLUME_MOUNT causes the volume *volume* identified by *volume_identifier* to be mounted on the device *device*.

(3) The operation creates a "mounted_on" link from *volume* to *device*, with "read_only" attribute set to *read_only*, and its reverse "mounted_volume" link.

(4) A lock of external and internal mode READ_SEMIPROTECTED is established on *device*.

(5) Write locks (of the default kind) are obtained on the created links.

Errors

(6) ACCESS_ERRORS (*device*, ATOMIC, MODIFY, EXPLOIT_DEVICE)

(7) ACCESS_ERRORS (*volume*, ATOMIC, READ, NAVIGATE)

(8) DEVICE_IS_BUSY (*device*)

(9) DEVICE_IS_UNKNOWN (*device*)

(10) LIMIT_WOULD_BE_EXCEEDED (MAX_MOUNTED_VOLUMES)

(11) PROCESS_IS_IN_TRANSACTION

(12) RANGE_IS_OUTSIDE_RANGE (volume associated with *volume*, *device*)

(13) VOLUME_CANNOT_BE_MOUNTED_ON_DEVICE (*volume*, *device*)

(14) VOLUME_IS_ALREADY_MOUNTED (*volume*)

(15) VOLUME_IS_UNKNOWN (*volume*)

(16) NOTE - When appropriate, the operation causes the physical mounting of the corresponding physical volume on the corresponding physical device. If *read_only* is true then the physical volume is mounted for reading only.

11.2.12 VOLUME_UNMOUNT

(1)
```
VOLUME_UNMOUNT (
    volume : Volume_designator
)
```

(2) VOLUME_UNMOUNT causes the volume *volume* to be unmounted.

(3) The "mounted_on" link from *volume* to the device *device* on which *volume* is mounted is deleted.

(4) Write locks (of the default kind) are obtained on the volume and on the deleted link; however the locks on the volume and on the link from that object do not prevent the unmounting of the volume.

Errors

(5) ACCESS_ERRORS (*device*, ATOMIC, MODIFY, EXPLOIT_DEVICE)

(6) ACCESS_ERRORS (*volume,* ATOMIC, READ, NAVIGATE)

(7) PROCESS_IS_IN_TRANSACTION

(8) VOLUME_HAS_OBJECTS_IN_USE (*volume*)

(9) VOLUME_IS_ADMINISTRATION_VOLUME (*volume*)

(10) VOLUME_IS_INACCESSIBLE (*volume*)

(11) VOLUME_IS_UNKNOWN (*volume*)

12 Files, pipes, and devices

12.1 File, pipe, and device concepts

(1)
```
Open_contents ::
    OPEN_OBJECT_KEY     : Natural
    CURRENT_POSITION: Current_position
```

(2) Current_position :: Token

(3) Contents_handle :: Token

(4) Position_handle :: Token

(5) Contents_access_mode = READ_WRITE | READ_ONLY | WRITE_ONLY | APPEND_ONLY

(6) Seek_position = FROM_BEGINNING | FROM_CURRENT | FROM_END

(7) Set_position = AT_BEGINNING | AT_POSITION | AT_END

(8)
```
File = seq of Octet
    represented by file
```

(9)
```
Pipe = seq of Octet
    represented by pipe
```

(10)
```
Device = seq of Octet
    represented by device
```

(11) Control_data = **seq o**f Octet

(12) Positioning_style = SEQUENTIAL | DIRECT | SEEK

(13) **sds** system:

(14) positioning: (**read**) **enumeration** (SEQUENTIAL, DIRECT, SEEK) := SEQUENTIAL;

(15)
```
file: child type of object with
contents file;
attribute
    contents_size: (read) natural;
    positioning;
end file;
```

(16)
```
pipe: child type of object with
contents pipe;
end pipe;
```

(17)
```
device: child type of object with
contents device;
attribute
    device_characteristics: (read) string;
    positioning;
link
    device_of: (navigate) reference link to workstation reverse controlled_device;
end device;
```

(18)
```
end system;
```

(19) The contents of a file, pipe, or device may be accessed by a process as a sequence of octets if it is *opened* by the process. The file, pipe, or device is then called an *open object*.

(20) The opening of a file, pipe, or device by a process is represented by an "open_object" link from the process to the file, pipe, or device, and an "opened_by" link from the file, pipe or device to the process.

(21) The "open_object_key" key attribute of the "open_object" link uniquely identifies the open object among the other objects opened by the process.

(22) The opening of an object's contents and the operation CONTENTS_GET_HANDLE_FROM_KEY result in the creation of a *contents handle*, which is an implementation-dependent reference to the open contents of an object. A separate open contents value exists for each process that opens a particular contents. The open contents consists of the following:

(23) - An *open object key* which is the key of an "open_object" link to the open object.

(24) - A *current position* which specifies one octet of the sequence within the logical sequence of octets. The position of the first octet is called FIRST, and of the last octet is called LAST. A current position can be shared by several open contents.

(25) The "device_characteristics" attribute of a device is a string with an implementation-defined syntax specifying implementation-dependent characteristics of the device.

(26) The "positioning" attribute of files and devices can be set only by CONTENTS_SET_PROPERTIES. It is defined as follows:

(27) - SEQUENTIAL indicates that the current position can be changed only by writing or reading octets in a sequential way.

(28) - DIRECT indicates that the current position can be changed either as by SEQUENTIAL or by means of a previously saved position, represented by an implementation-dependent *position handle*.

(29) - SEEK indicates that the current position can be changed either as by DIRECT or by an offset from another position.

(30) The contents of a pipe is always accessed sequentially.

(31) The contents size of a file is 0 if the file is empty and otherwise LAST - FIRST + 1.

(32) The "open_object" link has an "opening_mode" attribute which defines how the current position is updated by the contents operations. Let CP be the current position, and CP+*n* the position of

the *n*th octet after the current position. An operation is allowed for any opening mode unless otherwise stated below.

(33) In READ_WRITE opening mode:

(34) - opening the contents by CONTENTS_OPEN sets CP to FIRST;

(35) - successfully reading N octets by CONTENTS_READ returns the octets at positions CP, CP+1, ... CP+N-1, and changes CP to CP+N;

(36) - successfully writing N octets by CONTENTS_WRITE replaces or adds octets at positions CP, CP+1, ... CP+N-1, and changes CP to CP+N; and if LAST < CP+N, changes LAST to CP+N.

(37) In READ_ONLY opening mode:

(38) - opening the contents with CONTENTS_OPEN sets CP to FIRST;

(39) - successfully reading N octets by CONTENTS_READ returns the octets at positions CP, CP+1, ... CP+N-1, and changes CP to CP+N. For pipes and *read-once devices*, e.g. keyboards, the sequence of octets is changed to identify as FIRST the new current position. Which devices are read-once is implementation-defined.

(40) - CONTENTS_WRITE and CONTENTS_TRUNCATE are not allowed.

(41) In WRITE_ONLY opening mode:

(42) - opening the contents by CONTENTS_OPEN sets CP to FIRST;

(43) - CONTENTS_READ is not allowed;

(44) - successfully writing N octets by CONTENTS_WRITE replaces or adds octets at positions CP, CP+1, ... CP+N-1, and changes CP to CP+N; and if LAST < CP+N, changes LAST to CP+N.

(45) In APPEND_ONLY opening mode:

(46) - opening the contents by CONTENTS_OPEN sets CP to LAST+1;

(47) - CONTENTS_READ is not allowed;

(48) - successfully writing N octets by CONTENTS_WRITE sets CP to LAST+1, adds octets at positions LAST+1, LAST+2, ..., LAST+N, and changes LAST to LAST+N;

(49) - CONTENTS_SET_POSITION, CONTENTS_SEEK, and CONTENTS_TRUNCATE are not allowed.

(50) In READ_ONLY, WRITE_ONLY and READ_WRITE opening modes:

(51) - if positioning is DIRECT or SEEK, positioning with CONTENTS_SET_POSITION sets CP to a position identified by a position handle;

(52) - if positioning is SEEK, positioning with CONTENTS_SEEK sets CP to a position identified by an offset from another position.

(53) Pipes can be opened only in READ_ONLY or APPEND_ONLY mode.

(54) The "open_object" link has an "inheritable" attribute; if it is **true**, then a link of the same type, key, non-key system attributes, and destination object is created from any process created by the process origin of the link.

(55) The "open_object" link has a "non_blocking_io" attribute which defines the behaviour of the operations CONTENTS_READ and CONTENTS_WRITE. This property is always **true** for a

file, but is **true** for a pipe or a device only if it supports non-blocking input-output, i.e. a read or write operation does not wait until all data can be read or written, but reads or writes as much as it can. If it is **true**, then the destination pipe or device is said to be *non-blocking* (for the opening process).

(56) If an "open_object" link is inheritable, the associated contents handle is duplicated in the child process. As a consequence, the parent and child process share the current position.

(57) The effect of changing the current position by the operations CONTENTS_READ, CONTENTS_WRITE, CONTENTS_TRUNCATE, CONTENTS_SEEK, and CONTENTS_SET_POSITION is visible to all processes sharing that current position.

NOTES

(58) 1 The "open_object" links keyed by 0, 1, and 2 are called *standard input*, *standard output*, and *standard error* respectively. Conventionally, standard input is used by the process for the reading of commands or input data, standard output is used for the output of data, and standard error is used for the output of error diagnostics.

(59) 2 After creating a process, an "opened_object" link is created for each "open_object" link which has the inheritable property set to true. The designated object is however not open until the process is started: the starting of the process creates a link of type "open_by" from the designated object to the process.

(60) 3 In APPEND_ONLY opening mode, the current position is always LAST + 1. Several processes can append to the same file, pipe, or device and therefore concurrently modify the LAST position subject to locking rules.

(61) 4 On pipes and some kinds of devices (e.g. keyboards), the reading of a sequence of octets deletes the octets: the sequence of octets read is no longer readable and the new current position identifies as FIRST the next unread octet in the sequence. Several processes can concurrently read the same pipe or device in this way.

(62) 5 There are various situations allowing one or more contents handles to be associated with the same object in such a way that the CONTENTS_READ, CONTENTS_WRITE and CONTENTS_TRUNCATE operations performed on the two contents handles may interfere:

(63) - two contents handles opened within the context of the same activity, either within the same process or within different processes;

(64) - contents handles obtained from objects locked within concurrent activities but with compatible locks.

(65) 6 An application needing to manage such interferences without using separate activities and appropriate locks must use its own synchronization mechanisms.

(66) 7 The contents of pipes and devices are not affected by transaction rollback.

12.2 File, pipe, and device operations

12.2.1 CONTENTS_CLOSE

(1)
```
CONTENTS_CLOSE (
    contents    : Contents_handle
)
```

(2) CONTENTS_CLOSE deletes the contents handle contents, releasing any associated resources.

(3) The "open_object" link keyed by the open object key of *contents* and its complementary "opened_by" link (see 12.2.6) are deleted.

Errors

(4) CONTENTS_IS_NOT_OPEN (*contents*)

12.2.2 CONTENTS_GET_HANDLE_FROM_KEY

(1)
```
CONTENTS_GET_HANDLE_FROM_KEY (
    open_object_key   : Natural
)
    contents          : Contents_handle
```

(2) CONTENTS_GET_HANDLE_FROM_KEY returns in *contents* the contents handle of the calling process represented by the "open_object" link *link* with key *open_object_key*.

Errors

(3) LINK_DOES_NOT_EXIST (calling process, *link*)

12.2.3 CONTENTS_GET_KEY_FROM_HANDLE

(1)
```
CONTENTS_GET_KEY_FROM_HANDLE (
    contents          : Contents_handle
)
    open_object_key   : Natural
```

(2) CONTENTS_GET_KEY_FROM_HANDLE returns in *open_object_key* the "open_object_key" key attribute of the "open_object" link associated with the contents handle *contents*.

Errors

(3) CONTENTS_IS_NOT_OPEN (*contents*)

12.2.4 CONTENTS_GET_POSITION

(1)
```
CONTENTS_GET_POSITION (
    contents   : Contents_handle
)
    position   : Position_handle
```

(2) CONTENTS_GET_POSITION returns in *position* a position handle representing the current position of *contents*.

Errors

(3) CONTENTS_IS_NOT_OPEN (*contents*)

(4) If *contents* is a pipe, or if *contents* is a file or a device with positioning SEQUENTIAL:
CONTENTS_OPERATION_IS_INVALID (*contents*)

12.2.5 CONTENTS_HANDLE_DUPLICATE

(1)
```
CONTENTS_HANDLE_DUPLICATE (
    contents       : Contents_handle,
    new_key        : [ Natural ],
    inheritable    : Boolean
)
    new_contents   : Contents_handle
```

(2) CONTENTS_HANDLE_DUPLICATE creates a new open contents for the calling process and the object associated with *contents*, and returns a contents handle identifying it in *new_contents*. A new "open_object" link from the calling process to the object identified by *contents*, and a complementary "opened_by" link, are created.

(3) The "inheritable" attribute of the new "open_contents" link is set to *inheritable*. The key of the new "open_object" link is set to is set to *new_key*, if provided, and otherwise to an unused implementation-defined value. The "opening_mode" and "non_blocking_io" attributes of the new "open_object" link are set to the same values as for the "open_object" link from the calling process associated with *contents*.

(4) The new open contents shares the current position of *contents*.

Errors

(5) CONTENTS_IS_NOT_OPEN (*contents*)

(6) LIMIT_WOULD_BE_EXCEEDED (MAX_OPEN_OBJECTS)

(7) LIMIT_WOULD_BE_EXCEEDED (MAX_OPEN_OBJECTS_PER_PROCESS)

(8) OPEN_KEY_IS_INVALID (*new_key*)

12.2.6 CONTENTS_OPEN

(1)
```
CONTENTS_OPEN (
    object            : File_designator | Pipe_designator | Device_designator,
    opening_mode      : Contents_access_mode,
    non_blocking_io   : Boolean,
    inheritable       : Boolean
)
    contents          : Contents_handle
```

(2) CONTENTS_OPEN opens the contents of *object* in the opening mode *opening_mode* and returns a contents handle for it in *contents*.

(3) An "open_object" link is created from the calling process to *object* with opening mode set to *opening_mode,* non-blocking io set to *non_blocking_io*, and inheritable set to *inheritable*

(4) An "opened_by" link is created with an implementation-dependent key from *object* to the calling process, complementary to the created "open_object" link.

(5) If *opening_mode* is READ_ONLY, a read lock of the default mode is obtained on *object.*

(6) If *opening_mode* is WRITE_ONLY, READ_WRITE or APPEND_ONLY, a write lock of the default mode is obtained on *object.*

(7) After this operation, the object is operated on by the current activity and the lock established is not released before the contents is closed (see 16.1.8).

Errors

(8) If *opening_mode* is READ_ONLY or READ_WRITE:
ACCESS_ERRORS (*object*, ATOMIC, READ, READ_CONTENTS)

(9) If *opening_mode* is WRITE_ONLY or READ_WRITE:
ACCESS_ERRORS (*object*, ATOMIC, MODIFY, WRITE_CONTENTS)

(10) If *opening_mode* is APPEND_ONLY
ACCESS_ERRORS (*object,* ATOMIC, MODIFY, APPEND_CONTENTS)

(11) LIMIT_WOULD_BE_EXCEEDED (MAX_OPEN_OBJECTS)

(12) LIMIT_WOULD_BE_EXCEEDED (MAX_OPEN_OBJECTS_PER_PROCESS)

(13) NON_BLOCKING_IO_IS_INVALID (*object*, *non_blocking_io*)

(14) OPENING_MODE_IS_INVALID (*object, opening_mode*)

(15) STATIC_CONTEXT_IS_IN_USE (*object*)

12.2.7 CONTENTS_READ

(1)
```
CONTENTS_READ (
    contents    : Contents_handle,
    size        : Natural
)
    data        : Unstructured_contents
```

(2) CONTENTS_READ reads a sequence of *size* octets from *contents* at the current position and returns it in *data*, if available. If there are less than *size* octets but at least one octet from the current position to LAST inclusive, the operation returns in *data* that sequence of octets.

(3) The current position is set to the position after the last read octet.

(4) If *contents* is a pipe or a read-once device, the position after the last read octet is identified as FIRST after the operation.

(5) If there are no octets available for reading:

(6) - if *contents* is a non-blocking pipe or device, the operation fails.

(7) - if *contents* is a blocking device or a blocking pipe for which a contents handle is open in APPEND_ONLY mode, then the operation waits until some octets are available for reading. If, in the case of a pipe, the last contents handle open in APPEND_ONLY mode is closed while the operation is waiting, an empty sequence of octets is returned.

(8) - if *contents* is a file, *data* is set to the empty sequence.

Errors

(9) CONFIDENTIALITY_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(10) CONTENTS_IS_NOT_OPEN (*contents*)

(11) CONTENTS_OPERATION_IS_INVALID (*contents*)

(12) DATA_ARE_NOT_AVAILABLE (*contents*)

(13) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(14) OBJECT_IS_INACCESSIBLE (object determined by *contents*, ATOMIC)

(15) PIPE_HAS_NO_WRITERS (*contents*)

(16) VOLUME_IS_INACCESSIBLE (volume on which object determined by *contents* resides, ATOMIC)

12.2.8 CONTENTS_SEEK

(1)
```
CONTENTS_SEEK (
    contents        : Contents_handle,
    offset          : Integer,
    whence          : Seek_position
)
    new_position    : Natural
```

(2) CONTENTS_SEEK sets the current position of *contents* to a position determined by *offset* and *whence*, and returns the new current position as an offset from FIRST.

(3) If *contents* is a file or device which has the SEEK "positioning" property, the current position is set to a position defined as follows:

(4) - if *whence* is FROM_BEGINNING, the new position is FIRST + *offset*;

(5) - if *whence* is FROM_CURRENT, the new position is the current position + *offset*;

(6) - if *whence* is FROM_END, the new position is LAST + *offset* + 1.

(7) The resulting position cannot be smaller than FIRST.

(8) The resulting position RP can be greater than LAST. In this case, if subsequent writing occurs, LAST and the "contents_size" attribute of the file are set to RP + number of written octets. The octets which are between the previous LAST position and RP are returned as octets with the value 0 by subsequent calls of CONTENTS_READ. However, the file remains unchanged if no CONTENTS_WRITE occurs at the new RP position.

(9) If *contents* is a file or a device, the new value of the current position, offset from the beginning of the file, is returned in *new_position*.

Errors

(10) CONTENTS_IS_NOT_OPEN (*contents*)

(11) CONTENTS_OPERATION_IS_INVALID (*contents*)

(12) OBJECT_IS_INACCESSIBLE (object determined by *contents*, ATOMIC)

(13) POSITION_IS_INVALID (resulting position)

12.2.9 CONTENTS_SET_POSITION

(1)
```
CONTENTS_SET_POSITION (
    contents          : Contents_handle,
    position_handle   : Position_handle,
    set_mode          : Set_position
)
```

(2) CONTENTS_SET_POSITION sets the current position of *contents* to a position determined by *set_mode* and *position_handle*.

(3) If *set_mode* is AT_BEGINNING or AT_END, the current position of *contents* is set to FIRST or LAST + 1 respectively.

(4) If *set_mode* is AT_POSITION, the current position of *contents* is set to the position represented by *position_handle* which must have been previously obtained by a call of CONTENTS_GET_POSITION on *contents*.

Errors

(5) CONTENTS_IS_NOT_OPEN (*contents*)

(6) CONTENTS_OPERATION_IS_INVALID (*contents*)

(7) OBJECT_IS_INACCESSIBLE (object determined by *contents*, ATOMIC)

(8) POSITION_HANDLE_IS_INVALID (*position_handle*, *contents*)

12.2.10 CONTENTS_SET_PROPERTIES

(1)
```
CONTENTS_SET_PROPERTIES (
    contents      : Contents_handle,
    positioning   : Positioning_style
)
```

(2) CONTENTS_SET_PROPERTIES sets the positioning of the open file or device determined by *contents* to *positioning*.

(3) If *contents* determines a file, its positioning can be changed only if the file is empty.

Errors

(4) If *contents* determines a file:
CONTENTS_IS_NOT_EMPTY (*contents*)

(5) CONTENTS_IS_NOT_OPEN (*contents*)

(6) If *contents* determines a pipe, or a file or device open in mode READ_ONLY:
CONTENTS_OPERATION_IS_INVALID (*contents*)

(7) OBJECT_IS_INACCESSIBLE (object determined by *contents*, ATOMIC)

(8) POSITIONING_IS_INVALID (*contents*, *positioning*)

(9) NOTE – The change of properties is made on behalf of the activity in which the contents was opened.

12.2.11 CONTENTS_TRUNCATE

(1)
```
CONTENTS_TRUNCATE (
    contents   : Contents_handle
)
```

(2) CONTENTS_TRUNCATE truncates *contents* from the current position to the end.

(3) The "contents_size" attribute of *contents* is set to indicate the new size.

(4) LAST is reset to one less than the current position, which is unchanged, except when the current position is FIRST, in which case LAST is undefined and the file is empty.

(5) This operation applies only to files.

Errors

(6) CONFIDENTIALITY_CONFINEMENT_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(7) CONFIDENTIALITY_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(8) CONTENTS_IS_NOT_FILE_CONTENTS (*contents*)

(9) CONTENTS_IS_NOT_OPEN (*contents*)

(10) CONTENTS_OPERATION_IS_INVALID (*contents*)

(11) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(12) INTEGRITY_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(13) OBJECT_IS_INACCESSIBLE (object determined by *contents*, ATOMIC)

(14) VOLUME_IS_FULL (volume containing object determined by *contents*)

(15) VOLUME_IS_INACCESSIBLE (volume containing object determined by *contents*, ATOMIC)

(16) NOTE - CONTENTS_TRUNCATE can affect the size of a file while other operations are accessing it.

12.2.12 CONTENTS_WRITE

(1)
```
CONTENTS_WRITE (
    contents      : Contents_handle,
    data          : Unstructured_contents
)
    actual_size   : Natural
```

(2) CONTENTS_WRITE writes some or all of a sequence of octets *data* to *contents* at the current position, and returns the number of octets actually written.

(3) If *contents* is a file with opening mode READ_WRITE, WRITE_ONLY, or APPEND_ONLY and if writing to the file would not cause its size to exceed the MAX_FILE_SIZE, the sequence of octets *data* is written from the current position, and the current position is changed to the position following the last written octet. The contents size of *contents* is set to indicate the new size.

(4) If *contents* is a pipe with opening mode APPEND_ONLY and if writing to the pipe would not cause its size to exceed MAX_PIPE_SIZE, the sequence of octets *data* is written from the position LAST + 1, and the current position is changed to the position following the last written octet.

(5) If *contents* is a device with opening mode READ_WRITE, WRITE_ONLY, or APPEND_ONLY and if writing to the device would not cause its size to exceed any device-dependent maximum size limit, the sequence of octets *data* is written from the current position, and the current position is changed to the position following the last written octet.

(6) If the available space does not allow the whole of *data* to be written to *contents:*

(7) - if *contents* is a file and at least one octet can be written, as many octets as possible are written;

(8) - if *contents* is a file and no octet can be written (e.g. MAX_FILE_SIZE has been reached), the operation fails;

(9) - if *contents* is a non-blocking pipe, as many octets from *data* as there is room for are written; if no octets can be written (i.e. MAX_PIPE_SIZE has been reached) the operation fails.

(10) - if *contents* is a pipe which is not non-blocking, the operation waits, but on normal completion (i.e. after space has been made available in the pipe and no interrupt occurred) the operation has written all the octets;

(11) - if *contents* is a device which is not non-blocking, the operation waits until octets can be written;

(12) - if *contents* is a non-blocking device, as many octets as there are room for are written; if no octet can be written, the operation fails.

(13) In all cases, the octets of *data* are written in order starting with the first element, and the actual number of octets written to contents is returned in *actual_size*.

(14) If a concurrent CONTENTS_TRUNCATE operation is performed on the object contents after *contents* is opened, *data* is nevertheless written at the specified current position (i.e. there is no interference implying that the current position is reset) as if prior to the write a CONTENTS_SEEK operation had been performed with the current position as argument.

Errors

(15) CONFIDENTIALITY_CONFINEMENT_WOULD_BE_VIOLATED (object determined by *contents,* ATOMIC)

(16) CONFIDENTIALITY_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(17) CONTENTS_IS_NOT_OPEN (*contents*)

(18) CONTENTS_OPERATION_IS_INVALID (*contents*)

(19) DEVICE_LIMIT_WOULD_BE_EXCEEDED (*data*, *contents*)

(20) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (object determined by *contents,* ATOMIC)

(21) INTEGRITY_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(22) LIMIT_WOULD_BE_EXCEEDED ((MAX_FILE_SIZE, MAX_PIPE_SIZE))

(23) OBJECT_IS_INACCESSIBLE (object determined by *contents*, ATOMIC)

(24) If *contents* is a file:
PROCESS_FILE_SIZE_LIMIT_WOULD_BE_EXCEEDED (*data*, *contents*)(

(25) VOLUME_IS_FULL (volume containing object determined by *contents*)

(26) VOLUME_IS_INACCESSIBLE (volume containing object determined by *contents*)

12.2.13 DEVICE_GET_CONTROL

(1)
```
DEVICE_GET_CONTROL (
    contents      : Contents_handle,
    operation     : Natural
)
    control_data  : Control_data
```

(2) DEVICE_GET_CONTROL returns control information from the device contents *contents* in *control_data*, according to *operation*. The meanings of *operation* and *control_data* are implementation-defined and may be device-dependent.

Errors

(3) CONFIDENTIALITY_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(4) CONTENTS_IS_NOT_OPEN (*contents*)

(5) DEVICE_CONTROL_OPERATION_IS_INVALID (*contents***,** *operation*)

(6) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(7) OBJECT_IS_INACCESSIBLE (volume containing object determined by *contents*, ATOMIC)

(8) VOLUME_IS_INACCESSIBLE (volume on which object determined by *contents* resides, ATOMIC)

12.2.14 DEVICE_SET_CONTROL

(1)
```
DEVICE_SET_CONTROL (
    contents      : Contents_handle,
    operation     : Natural,
    control_data  : Control_data
)
```

(2) DEVICE_SET_CONTROL performs a control operation on the device contents *contents,* according to *operation*. The parameters for the operation are specified in *control_data*. The

meanings of *operation* and *control_data* are implementation-defined and may be device-dependent.

Errors

(3) CONFIDENTIALITY_CONFINEMENT_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(4) CONFIDENTIALITY_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(5) CONTENTS_IS_NOT_OPEN (*contents*)

(6) DEVICE_CONTROL_OPERATION_IS_INVALID (*contents*, *operation*)

(7) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(8) INTEGRITY_WOULD_BE_VIOLATED (object determined by *contents*, ATOMIC)

(9) OBJECT_IS_INACCESSIBLE (object determined by *contents*, ATOMIC)

(10) VOLUME_IS_FULL (volume containing object determined by *contents*)

(11) VOLUME_IS_INACCESSIBLE (volume containing object determined by *contents*, ATOMIC)

13 Process execution

13.1 Process execution concepts

13.1.1 Static contexts

(1)
```
sds system:
```

(2)
```
static_context: child type of file with
attribute
    max_inheritable_open_objects: natural := 3;
    interpretable: boolean := false;
link
    interpreter: reference link to static_context;
    restricted_execution_class: reference link to execution_class;
end static_context;
```

(3)
```
end system;
```

(4) The max (maximum number of) inheritable open objects is the maximum number of open objects that a process running the static context may inherit from the process which created it.

(5) A static context is *interpretable* if "interpretable" is **true**; otherwise it is *executable*.

(6) The *interpreter* of an interpretable static context is the destination of the "interpreter" link, if there is one; it must not itself be interpretable.

(7) A static context is *foreign* if it has a restricted execution class and that execution class (see 13.1.3) has a usable execution site which is a foreign system; otherwise it is *native*.

(8) The *execution class* of an executable static context is the set of execution sites in which the static context may run. If the static context has a "restricted_execution_class" link then its execution class contains just the destination object of that link; otherwise it contains all the execution sites in the PCTE installation. The execution class of an interpretable static context is the intersection of that set and the execution class of the actual interpreter of the static context.

NOTES

(9) 1 A static context (short for static context of a program) is an executable or interpretable program in a static form that can be run by a process, either directly by loading and executing it (executable) or indirectly by running another static context as an interpreter (interpretable). It may be run either by a PCTE implementation or by a foreign system.

(10) 2 The default of 3 for maximum inheritable open objects allows inheritance of standard input, output and error channels as supported by some operating systems. The number of open objects is limited to MAX_OPEN_OBJECTS_PER_PROCESS (see clause 24) so this is the maximum effective value for maximum inheritable open objects.

(11) 3 The format of the contents of an executable static context is implementation-defined by the PCTE implementation (for workstations in the execution class) or the foreign system implementation (for foreign systems in the execution class) of the execution site.

(12) 4 If an interpretable static context has no interpreter, a static context is selected to interpret it as described in PROCESS_START.

(13) 5 A static context has other properties defined in the security SDS (see 19.1.1).

(14) 6 The fact that the interpreter of an interpretable static context is not interpretable is checked by PROCESS_START and PROCESS_CREATE_AND_START, but not by LINK_CREATE, OBJECT_SET_ATTRIBUTE, OBJECT_SET_SEVERAL_ATTRIBUTES, OBJECT_RESET_ATTRIBUTE, or OBJECT_DELETE_ATTRIBUTE.

13.1.2 Foreign execution images

```
(1)   sds system:

(2)   foreign_execution_image: child type of object with
      attribute
          foreign_name: string;
      link
          on_foreign_system: reference link to foreign_system;
      end foreign_execution_image;

(3)   end system;
```

(4) The syntax and semantics of the foreign name are implementation-defined.

(5) The "on_foreign_system" link defines a foreign system which may execute the foreign execution image.

NOTES

(6) 1 A foreign execution image differs from a static context for use on a foreign system in that it is only a representation of the image to be executed. The execution image itself is of undefined format and is not represented in the object base.

(7) 2 The foreign name is intended to provide enough information to determine the foreign system object, e.g. a file, which contains an execution image.

13.1.3 Execution classes

```
(1)   sds system:

(2)   execution_site_identifier: natural;

(3)   execution_class: child type of object with
      link
          usable_execution_site: reference link (execution_site_identifier) to execution_site;
      end execution_class;

(4)   end system;
```

(5) An execution class specifies a set of execution sites (workstations or foreign systems) on which any static context with that execution class may be executed. Execution sites are defined in 18.1.

NOTES

(6) 1 If a static context has no restricted execution class, the choice of execution site may be specified when the static context is run; otherwise it is implementation-defined.

(7) 2 If an execution class has no usable execution site, a static context with that execution class as a restricted execution class is unable to run. Thus it is possible to (temporarily) prevent a static context from running.

(8) 3 The addition and removal of execution sites to and from an execution class is performed using operations of clause 9. An execution site is a usable execution site of an execution class if and only if there is a "usable_execution_site" link between the site and the class. The value of the key of such a link is unimportant.

(9) 4 While it is recommended that tools keep the "execution_site_identifier" key consistent with the execution site identifier of the usable execution site in the execution site directory, a PCTE implementation is not required to enforce this consistency, nor even to ensure that the key is any execution site identifier in the execution site directory.

(10) 5 The definition of an execution class allows both workstations and foreign systems to be of the same execution class. In practice, such a mixed class is unlikely to be useful.

13.1.4 Processes

(1) Initial_status = RUNNING | SUSPENDED | STOPPED

(2) **sds** system:

(3) inheritable: **boolean** := **true**;

(4) referenced_object: (**navigate**) **designation link** (reference_name: **string**) **to** object **with attribute**
    inheritable;
end referenced_object;

(5) open_object: (**navigate**) **designation link** (open_object_key: **natural**) **to** file, pipe, device **with attribute**
    opening_mode: (**read**) **enumeration** (READ_WRITE, READ_ONLY, WRITE_ONLY,
        APPEND_ONLY) := READ_ONLY;
    non_blocking_io: (**read**) **boolean**;
    inheritable;
end open_object;

(6) is_listener: (**navigate**) **non_duplicated designation link** (number) **to**
    message_queue **with attribute**
    message_types: (**read**) **string**;
end is_listener;

(7) process_waiting_for: (**navigate**) **designation link** (number) **to** object **with attribute**
    waiting_type: (**read**) **enumeration** (WAITING_FOR_LOCK, WAITING_FOR_TERMINATION,
        WAITING_FOR_WRITE, WAITING_FOR_READ) := WAITING_FOR_LOCK;
    locked_link_name;
end process_waiting_for;

(8)
```
process: child type of object with
attribute
    process_status: (read) non_duplicated enumeration (UNKNOWN, READY, RUNNING,
        STOPPED, SUSPENDED, TERMINATED) := UNKNOWN;
    process_creation_time: (read) time;
    process_start_time: (read) time;
    process_termination_time: (read) time;
    process_user_defined_result: string;
    process_termination_status: (read) integer;
    process_priority: (read) natural;
    process_file_size_limit: (read) natural;
    process_string_arguments: (read) string;
    process_environment: (read) string;
    process_time_out: (read) natural;
    acknowledged_termination: (read) boolean;
    deletion_upon_termination: (read) boolean := true;
    time_left_until_alarm: (read) non_duplicated natural;
    character_encoding: (read) non_duplicated natural;
link
    process_object_argument: designation link (number) to object;
    executed_on: (navigate) designation link to execution_site;
    referenced_object;
    open_object;
    reserved_message_queue: (navigate) designation link (number) to message_queue;
    is_listener;
    default_interpreter: designation link to static_context;
    actual_interpreter: (navigate) designation link to static_context;
    process_waiting_for;
    parent_process: (navigate, delete) implicit link to process reverse child_process;
    started_in_activity: (navigate) reference link to activity reverse process_started_in;
component
    child_process: (navigate, delete) composition link (number) to process reverse
        parent_process;
    started_activity: (navigate) composition link (number) to activity reverse started_by;
end process;
```

(9)
```
end system;
```

(10)
```
sds metasds:
```

(11)
```
import object type system-process;
```

(12)
```
extend object type process with
link
    sds_in_working_schema: (navigate) designation link (number) to sds;
end process;
```

(13)
```
end metasds;
```

(14) A process is a means of running a static context or foreign execution image. *Creation* of a process refers to the action of PROCESS_CREATE. A process *runs* the static context (executable or interpretable) or foreign execution image specified when the process is created. A process *executes* the static context or foreign execution image specified when the process is created unless an interpretable static context is specified, in which case it executes another static context which is executable.

(15) A process executes by the execution of one or more *threads*. Within a process, threads may execute in parallel (proceed independently), or execution may switch between threads, or both, according to rules not defined in this ECMA Standard. A thread is *suspended* (i.e. its execution

does not progress) when it is executing an operation which is waiting for the occurrence of an event (see 8.7.2). A binding must define the mapping of threads and of their suspension to the binding language. A binding may impose limitations on threads by the definition of the rules for their interaction; in particular, a binding may specify that, except for the activation or waking of a handler (see 14.1), a process always executes by the execution of one and only one thread. The activation of a handler normally involves execution of a separate thread, although there may be special binding-defined rules governing this execution.

(16) Whether other threads of a process (if any) can continue to execute while one thread is suspended, and whether such threads can issue from operation calls, are instances of binding-defined rules governing the execution of threads.

(17) The process status is the status of the process with respect to execution. State transitions occur as the result of operations in 13.2 or of events outside tool control, e.g. a thread reaching a breakpoint. The process status may have the following values:

(18) - READY: ready to execute.

(19) - RUNNING: executing: one or more threads of the process are running or suspended.

(20) - STOPPED: stopped from execution: all threads of the process are stopped; this is for use in process monitoring, see 13.5.

(21) - SUSPENDED: suspended from execution: all threads of the process are suspended; this results from PROCESS_SUSPEND.

(22) - TERMINATED: prevented from further execution.

(23) The status value STOPPED is required only by the monitoring module (see 13.5).

(24) In addition the process status has an initial value UNKNOWN which it is given if the process is created by operations in clause 9. Such a process is prevented from executing.

(25) If one thread of a process is stopped, then all are, and similarly with suspension.

(26) The terms *ready*, *running*, *stopped*, *suspended, terminated* and *unknown* apply to a process whose process status is READY, RUNNING, STOPPED, SUSPENDED, TERMINATED or UNKNOWN, respectively. The terms 'running' and 'stopped' are also applied to a thread of a process. A process *starts* when its status changes from READY.

(27) A *breakpoint* is an implementation-defined marker defining a point in a process such that when execution reaches that point while the process is running, the process status is changed to STOPPED.

(28) If a handler is executed on behalf of a suspended listening process, the status of the listening process is changed to RUNNING (see 14.1).

(29) The precise time of a change of process status as recorded in the process creation, start or termination time is implementation-dependent except that it is between the start and end of any operation that causes the change of process status.

(30) The process creation time is the time when the process was created.

(31) The process start time is the time when the process started to run a static context or foreign execution image. Its value is the default value of time attributes if the process is ready.

(32) The process termination time is the time when the process terminated. Its value is the default value of time attributes unless the process is terminated.

(33) The semantics of the process user defined result are not defined in this ECMA Standard.

(34) The process termination status specifies the conditions under which the process terminated. Its value is the default value of integer attributes unless the process is terminated. The process termination status has two sets of named values, whose actual values are implementation-defined. Other values may be set using PROCESS_SET_TERMINATION_STATUS or PROCESS_TERMINATE but are not defined in this Standard. The sets of named values are:

(35) - Success:

(36) . EXIT_SUCCESS: The process has terminated normally, i.e. not as in the failure cases. The process termination status has this value if a process terminates other than by PROCESS_TERMINATE (explicitly or implicitly called), and the process termination status has not been changed by PROCESS_SET_TERMINATION_STATUS.

(37) - Failure:

(38) . EXIT_ERROR: The process has been terminated abnormally by itself (using PROCESS_TERMINATE).

(39) . FORCED_TERMINATION: The process has been terminated abnormally by another process (using PROCESS_TERMINATE).

(40) . SYSTEM_FAILURE: The process has been terminated abnormally by the PCTE implementation.

(41) . ACTIVITY_ABORTED: The process has been terminated abnormally as a result of the destination of its "started_in_activity" link being aborted by ACTIVITY_ABORT.

(42) The process priority defines the priority of running the process relative to that of other processes. The range of values is from 0 to the implementation-defined limit MAX_PRIORITY_VALUE. Their effect is implementation-defined except that a greater integer value indicates a greater priority.

(43) The process file size limit defines the maximum contents size of each file to which the process writes.

(44) The value of the process string arguments is a string of the following syntax, which defines it as a sequence of zero or more substrings. Each substring is an argument preceded by the length of the argument in hexadecimal notation. The semantics of the sequence of arguments is not defined in this ECMA Standard.

```
(45)    arguments = {substring};
(46)    substring= length, argument;
(47)    length = hex digit, hex digit, hex digit, hex digit;
(48)    hex digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';
(49)    argument = (*any sequence of graphic characters*);
```

(50) The semantics of process environment is not defined in this ECMA Standard. The value has the same syntax as the process string arguments.

(51) The process time out limits the duration of each *indivisible* operation, i.e. each operation whose execution does not cause it to wait, and also of each operation whose execution causes the creation of a "process_waiting_for" link from the calling process. If the value is 0, the limit is infinite, otherwise the limit is the value in seconds. An operation whose duration exceeds the limit terminates with the error OPERATION_HAS_TIMED_OUT.

(52) If the value is greater than 0, the time left until alarm defines the maximum duration in seconds that a process will be suspended when it next suspends or, while the process is suspended, the

maximum duration until it is resumed. If the process is resumed before the alarm goes off, the value of time left until alarm indicates the unexpired duration. Otherwise when the time expires the process receives an implementation-defined alarm message of message type WAKE (provided it has reserved a message queue and is handling wakeup messages) and is resumed. If there is more than one reserved message queue that has a handler enabled to handle WAKE messages, the system chooses in an implementation-defined way which message queue receives the WAKE message.

(53) The acknowledged termination is **true** when the process has terminated and the parent process has continued running after waiting for termination.

(54) If deletion upon termination is **true** and the deletion conditions are satisfied, the process is deleted automatically when it terminates, after acknowledged termination of this process has been set **true** by the parent process.

(55) The character encoding defines the single- or multi-octet character set in terms of which pathnames, keys, and type names are interpreted. The default value of 0 refers to ISO 8859-1; the interpretation of other values is implementation-defined. An unrecognised value is defaulted to 0.

(56) The "sds_in_working_schema" links specify by their key values a sequence of SDSs which determines the working schema of the process (see 8.1). The "sds_in_working_ schema" links are created when a process is created and may be changed by PROCESS_SET_WORKING_SCHEMA.

(57) The semantics of the process object arguments is not defined in this ECMA Standard.

(58) The destination of the "executed_on" link is called the *execution site of the process*.

(59) Referenced objects are used in the construction of object designators through their reference names (see clause 23). Referenced objects are created and deleted by PROCESS_SET_ REFERENCED_OBJECT and PROCESS_UNSET_REFERENCED_OBJECT respectively. Reference name values are restricted to the values of key string value defined in 23.1.2.7.

(60) The following reference names are reserved and refer to the given referenced objects:

(61) - 'self': This process. This referenced object always exists, cannot be changed and has inheritability **false**.

(62) - 'static_context': The static context run by the process. This referenced object always exists, cannot be changed and has inheritability **false**.

(63) - 'common_root': The common root (see 9.1.2). This referenced object always exists, cannot be changed and has inheritability **true**.

(64) - 'home_object': The meaning of the 'home_object' referenced object is not defined in this ECMA Standard.

(65) - 'current_object': The meaning of the 'current_object' referenced object is not defined in this ECMA Standard. Conventions for using it are given in 23.1.2.2.

(66) The referenced objects with reference names "static_context", "common_root", "home_object", and "current_object" are known as the *static context of the process*, *common root*, *home object*, and *current object* respectively.

(67) If inheritability is **true** the referenced object is to be made a referenced object of each child process created by this process (and inheritability is to be set **true** for it). The inheritability of a referenced object may be changed by operations in clause 9. An inherited "referenced_object"

link may be deleted by the child process but this does not affect the referenced objects of the creating process.

(68) An open object is an object opened for access to its contents (see clause 12). If inheritable is set **true**, the open object is to be inherited as opened (and the corresponding current position is to be shared) by each child process created by this process in the manner specified by the attributes of the "open_object" link (and inheritable is to be set **true** for the child's open object). The inheritable attribute of an open object may be changed by operations in clause 9. An inherited open object may be closed by the child process but this does not affect the open objects of the creating process. The semantics of the other attributes of an open object are defined by the operations in clause 12.

(69) For open objects with keys 0, 1 and 2 see 12.1.

(70) The default interpreter, if it exists, is a static context which will interpret the static context run by a process if it is interpretable and has no interpreter. The value of the default interpreter may be changed by operations in clause 9.

(71) The actual interpreter is the static context that interprets an interpretable static context.

(72) For reserved message queue and "is_listener" see 14.1.

(73) The destination of the "process_waiting_for" link is a resource that a thread of the process is waiting for; for further attributes see 16.1.2. A link of this type exists for each operation that is waiting. The waiting type values are:

(74) - WAITING_FOR_LOCK: waiting to establish a lock on a resource which already has an incompatible lock.

(75) - WAITING_FOR_TERMINATION: waiting for a child process to terminate.

(76) - WAITING_FOR_WRITE: waiting to write to a full message queue, a full pipe, a device, an audit file or an accounting log.

(77) - WAITING_FOR_READ: waiting to read from a message queue containing no message of the specified type, an empty pipe, or a device.

(78) The started in activity is the activity which was the current activity of the parent process at the time the process was created. Activities are defined in clause 16.

(79) A process is either the initial process of a workstation (see 18.1.2) or a child process of one other process.

(80) The parent process is the process which created this process or another process nominated by the creating process to be the parent.

(81) For the started activities, see clause 16.

NOTES

(82) 1 The process user defined result is provided for tool-defined use, especially for a child process to pass back results to its parent on termination.

(83) 2 The process priority is intended to be mapped to the process priority of an underlying operating system (if there is one). The number of possible values should be a power of 2.

(84) 3 The process string arguments is intended for passing parameters in the form of strings to a child process running a tool written in a language which specifies a mechanism for passing parameters to the tool. The specification of the length in hexadecimal notation enables the maximum length of an argument to be stored in 2 octets.

(85) 4 The process environment is provided as a mechanism for modifying aspects of the environment in which a child process is to run.

(86) 5 If the acknowledged termination of a process is **true**, then the process has terminated but could not be deleted, e.g. because deletion upon termination is **false**, or because there was a reference link to the process.

(87) 6 The "process_object_argument" link is intended for designating an object to a process, e.g. a print spooler, while it is running. The process may use key values to distinguish the different objects so designated if it does not delete the link to each process object argument after it has been processed.

(88) 7 A process can only be moved (thus changing its volume number) while the process status is READY or TERMINATED. It is recommended that a ready process is only moved to a volume that is controlled by the execution site which is to execute the process or, if the execution site is a discless workstation, to one that can be accessed efficiently.

(89) 8 The child processes of a process are components of that process, but this does not mean that operations on a process apply also to its child processes; e.g. terminating a process does not of itself terminate its child processes.

(90) 9 Many of the links of process that have no reverse link have a corresponding link which is effectively a reverse link except that only the link from the process exists before a process is started.

(91) 10 Operations specific to processes, i.e. those with names starting with "PROCESS_", do not establish any locks on the process, its links or its attributes (and thus these changes are not reversed if the transaction is aborted).

(92) 11 Operations specific to processes do not require discretionary access control on the calling process, its links or its attributes.

(93) 12 A process has other properties defined in the security and accounting SDSs.

(94) 13 The implicit creation and deletion of a usage designation link is allowed by operations defined in clause 13 even if the origin object of the link resides on a read-only volume or is a copy object.

(95) 14 Table 3 shows the available transitions of process status.

Table 3 - Available transitions of process status

	To				
From	**Ready**	**Running**	**Suspended**	**Stopped**	**Terminated**
(nonexistent)	CR	CS	X	X	X
Ready	N	ST	ST	ST	X
Running	X	N	SU	BP	TE
Suspended	X	RE,H	N	X	TE
Stopped	X	CO	X	N	TE
Terminated	X	X	X	X	N

Key

BP	breakpoint
CO	PROCESS_CONTINUE
CR	PROCESS_CREATE
CS	PROCESS_CREATE_AND_START
RE	PROCESS_RESUME
ST	PROCESS_START
SU	PROCESS_SUSPEND
TE	PROCESS_TERMINATE
H	execution of a message handler (see 14.1)
N	null transition
X	impossible transition

(96) 15 It is intended that a PCTE implementation maintain its integrity against operation calls from concurrent threads. In addition, the implementation may provide some degree of concurrency within operations, but that is not

mandatory. Thus operations called in concurrent threads may block immediately until an operation called earlier has terminated. Such implementation dependence is likely to apply to all language bindings supported by the implementation in addition to binding dependences that result from the level of support for threads by the binding language.

(97) 16 When a new process is created, it inherits the value of the character set of the calling process. Thus an implementation can set the character set of the initial process to the preferred value; as all processes are derived from the initial process, they would then work by default with this character set, and no further action would be necessary to select the character set, unless another character set is required for some tools.

13.1.5 Initial processes

(1) Each workstation in a PCTE installation has an *initial process*; this is a process that is created by implementation-dependent means such that, when it starts to run a tool, it is indistinguishable from a process that has been created by PROCESS_CREATE and modified by other PCTE operations, except that the initial process has no parent process. When the first static context runs in the initial process, the initial process has the following particular values for attributes and links:

(2) - the volume on which the process resides is the administration volume of the execution site of the initial process;

(3) - the execution site of the process is the workstation for which the process is the initial process;

(4) - the static context of the process is the static context being run by the initial process;

(5) - the destination of the "actual_interpreter" link is the static context being executed by the initial process, if any;

(6) - the destination of the "started_in_activity" is the outermost activity of the execution site (see 16.1.1);

(7) - the static context of the initial process is a member of the predefined program group PCTE_SECURITY or of a program group which has PCTE_SECURITY as one of its program supergroups.

NOTE **-** The initial process of a workstation is intended to start one or more processes, each of which runs a static context, typically a login or user authentication tool (which may be a portable tool), to perform various tasks when a human user starts or ends a session at the workstation. It has no consumer identity. The tasks to be performed at the start of the session may include, for example:

(8) - authenticating the human user and setting the discretionary and mandatory context appropriate to that user by calling PROCESS_SET_USER_AND_USER_GROUP_IDENTITY; this must be done before any processing on behalf of the user to assure the security of the PCTE installation;

(9) - initializing a general purpose environment for the running of tools by the user;

(10) - tailoring the environment to the user, for example by setting the referenced object "home_object".

13.1.6 Profiling and monitoring concepts

(1) Profile_handle :: Token

(2) Buffer = **seq of** Natural

(3) Address :: Token

(4) Process_data = **seq of** Octet

(5) These types are used in profiling and monitoring operations; see 13.4 and 13.5.

13.2 Process execution operations

13.2.1 PROCESS_CREATE

(1)
```
PROCESS_CREATE (
    static_context      : Static_context_designator | Foreign_execution_image_designator,
    process_type        : Process_type_nominator,
    parent              : [ Process_designator ],
    site                : [ Execution_site_designator ],
    implicit_deletion   : Boolean,
    access_mask         : Atomic_access_rights
)
    new_process         : Process_designator
```

(2) If no value is supplied for *parent, parent* designates the calling process.

(3) PROCESS_CREATE creates a process that is able to run a static context. The new process becomes a child process of *parent* (either the calling process or an ancestor of the calling process).

(4) The new process *new_process* is of type *process_type* with attributes and links as defined below.

(5) - Attributes and links of type "object" defined in SDS 'system' as by OBJECT_CREATE, except that "volume_identifier" is set to "volume_identifier" of *parent*, if *parent* and the new process have the same execution site, otherwise to "volume_identifier" of the new process's execution site.

(6) - Attributes and links of type "process" defined in SDS 'system':

(7) . "process_status" is set to READY;

(8) . "process_creation_time" is set to the current time (a value of system time between the start and end of the operation);

(9) . "process_priority" is set to "process_priority" of the calling process;

(10) . "process_file_size_limit" is set to "process_file_size_limit" of the calling process;

(11) . "deletion_upon_termination" is set to *implicit_deletion*;

(12) . the character encoding is set to the character encoding of the calling process;

(13) . "sds_in_working_schema" links are created, each with the same destination and key as each of the "sds_in_working_schema" links of the calling process;

(14) . an "executed_on" link is created to *site*, or if *site* is absent:

(15) . if *static_context* is executable, to an implementation-dependent member of the execution class of *static_context*;

(16) . if *static_context* is interpretable, to an implementation-dependent member of the intersection of the execution classes of *static_context* and its interpreter;

(17) . if *static_context* is a foreign execution image, to the destination of the "on_foreign_system" link from *static_context*;

(18) . for each "referenced_object" link of the calling process with inheritability **true** (except for the referenced objects "self" and "static_context") a "referenced_object" link is created with the same destination and key; in addition, "referenced_object" links with reference names "self" and "static_context" are created with destinations the new process and *static_context*, respectively, and inheritability **false**;

(19) . for each "open_object" link of the calling process with inheritable **true** an "open_object" link is created with the same destination and key, and with the same opening mode and non-blocking io, in ascending order of key value, up to a limit of "max_inheritable_open_objects" of *static_context*;

(20) . if the calling process has a default interpreter, a "default_interpreter" link is created to the default interpreter of the calling process;

(21) . a "parent_process" link to *parent* and its reverse "child_process" link are created;

(22) . a "started_in_activity" link to the current activity of *parent* and its reverse "process_started_in" link are created.

(23) - Attributes and links of type "object" defined in SDS 'discretionary_security' as by OBJECT_CREATE with *access_mask*, except:

(24) . "atomic_acl" has two additional groups added, if not already present, and in any case these two groups are granted all access rights. These groups are:

(25) . the user of the new process;

(26) . the predefined security group PCTE_EXECUTION;

(27) . "confidentiality_label" is set to "confidentiality_label" of the calling process;

(28) . "integrity_label" is set to "integrity_label" of the calling process.

(29) - Attributes and links of type "process" defined in SDSs 'discretionary_security' and 'mandatory_security':

(30) . "default_atomic_acl" is set to "default_atomic_acl" of the calling process;

(31) . "default_object_owner" is set to "default_object_owner" of the calling process;

(32) . "floating_confidentiality_level" is set to "floating_confidentiality_level" of the calling process;

(33) . "floating_integrity_level" is set to "floating_integrity_level" of the calling process;

(34) . a "user_identity" link is created to the user of the calling process;

(35) . an "adopted_user_group" link is created to the adopted user group of the calling process;

(36) . "adoptable_user_group" links are created, each with the same destination and key as each of those "adoptable_user_group" links of the calling process with "adoptable_for_child" **true**.

(37) - Attributes and links of type "process" defined in SDS 'accounting':

(38) . a "consumer_identity" link is created to the consumer identity of the calling process, if any.

(39) PROCESS_CREATE returns a designator of the new process as *new_process*.

(40) If the workstation controlling the device on which is mounted the volume on which *new_process* resides becomes inaccessible before *new_process* is started, the "sds_in_working_schema", "executed_on", "opened_objects", "user_identity", "adopted_user_group", "adoptable_user_group", "referenced_object", and "consumer_identity" designation links from *new_process* are deleted, the status of *new_process* is set to TERMINATED and the exit status of *new_process* is set to SYSTEM_FAILURE.

Errors

(41) ACCESS_ERRORS (*static_context*, ATOMIC, MODIFY, EXECUTE)

(42) If *static_context* is interpretable:
ACCESS_ERRORS (interpreter of *static_context*, ATOMIC, READ, EXECUTE)

(43) ACCESS_ERRORS (*parent*, ATOMIC, READ, APPEND_LINKS)

(44) ACCESS_ERRORS (the current activity of *parent*, ATOMIC, MODIFY, APPEND_IMPLICIT)

(45) EXECUTION_CLASS_HAS_NO_USABLE_EXECUTION_SITES (execution class of *static_context*)

(46) EXECUTION_SITE_IS_INACCESSIBLE (*site*)

(47) EXECUTION_SITE_IS_NOT_IN_EXECUTION_CLASS (*site*, *static_context*)

(48) EXECUTION_SITE_IS_UNKNOWN (*site*)

(49) If *static_context* is a foreign execution image:
FOREIGN_EXECUTION_IMAGE_HAS_NO_SITE (*static_context*)

(50) LABEL_IS_OUTSIDE_RANGE (*new_process*, the volume on which *new_process* would reside)

(51) LABEL_IS_OUTSIDE_RANGE (*new_process*, the would-be execution site of *new_process*)

(52) LIMIT_WOULD_BE_EXCEEDED (MAX_PROCESSES)

(53) LIMIT_WOULD_BE_EXCEEDED (MAX_PROCESSES_PER_USER)

(54) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL

(55) If *parent* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_process*)

(56) PROCESS_LACKS_REQUIRED_STATUS (*parent*, (READY, RUNNING, STOPPED, SUSPENDED))

(57) If *parent* is not the calling process:
PROCESS_IS_NOT_ANCESTOR (*parent*)

(58) PROCESS_IS_UNKNOWN (*parent*)

(59) REFERENCE_CANNOT_BE_ALLOCATED

(60) STATIC_CONTEXT_REQUIRES_TOO_MUCH_MEMORY (*static_context*)

(61) USAGE_MODE_ON_OBJECT_TYPE_WOULD_BE_VIOLATED ("object", *process_type*)

(62) If *process* is the calling process:
VOLUME_IS_FULL (calling process)

(63) NOTE - It is implementation-dependent which underlying resources (e.g. memory) required for process execution are allocated by PROCESS_CREATE and which are allocated by PROCESS_START.

13.2.2 PROCESS_CREATE_AND_START

(1)
```
PROCESS_CREATE_AND_START (
    static_context     : Static_context_designator | Foreign_execution_image_designator,
    arguments          : String,
    environment        : String,
    site               : [ Execution_site_designator ],
    implicit_deletion  : Boolean,
    access_mask        : Atomic_access_rights
)
    new_process        : Process_designator
```

(2) PROCESS_CREATE_AND_START creates and runs a process asynchronously in one single operation.

(3) The overall effect is as for the following sequence of operations.

```
(4)     new_process := PROCESS_CREATE (static_context, "process", nil, site,
            implicit_deletion, access_mask);
(5)     PROCESS_START (new_process,  arguments, environment, site, RUNNING);
```

(6) PROCESS_CREATE_AND_START is an atomic operation for the calling process.

Errors

(7) ACCESS_ERRORS (*static_context*, ATOMIC, MODIFY, EXECUTE)

(8) For each SDS *sds* which is the destination of an "in_working_schema_of" link from *new_process*:
ACCESS_ERRORS (*sds*, ATOMIC, SYSTEM_ACCESS)
DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*new_process*, *sds*, ATOMIC, EXPLOIT_SCHEMA)

(9) For each open object *object* which is the destination of an "open_object" link from the calling process:

ACCESS_ERRORS (*object*, ATOMIC, SYSTEM_ACCESS)

If the link's opening mode attribute is READ_ONLY or READ_WRITE,
DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*new_process*, *object*, ATOMIC, READ, READ_CONTENTS)

If the link's opening mode attribute is WRITE_ONLY or READ_WRITE,
DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*new_process* , *object*, ATOMIC, READ, WRITE_CONTENTS)

If the link's opening mode attribute is APPEND_ONLY,
DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*new_process* , *object*, ATOMIC, READ, APPEND_CONTENTS)

(10) For the activity *activity* which is the destination of the "started_in_activity" link from the calling process:
ACCESS_ERRORS (*activity*, ATOMIC, SYSTEM_ACCESS)
ACTIVITY_STATUS_IS_INVALID (*activity*, ACTIVE)

(11) For the user *user* which is the destination of the "user_identity" link from the calling process:
ACCESS_ERRORS (*user*, ATOMIC, SYSTEM_ACCESS)

(12) For the user group *group* which is the destination of the "adopted_user_group" link from the calling process:
ACCESS_ERRORS (*group*, ATOMIC, SYSTEM_ACCESS)

(13) For the consumer group *group* which is the destination of the "consumer_identity" link from the calling process:
ACCESS_ERRORS (*group*, ATOMIC, SYSTEM_ACCESS)

(14) If *static_context* has an interpreter:
ACCESS_ERRORS (interpreter of *static_context*, ATOMIC, MODIFY, EXECUTE)

(15) CONTROL_WOULD_NOT_BE_GRANTED (*new_process*)

(16) EXECUTION_CLASS_HAS_NO_USABLE_EXECUTION_SITES (execution class of *static_context*)

(17) EXECUTION_SITE_IS_INACCESSIBLE (*site*)

(18) EXECUTION_SITE_IS_NOT_IN_EXECUTION_CLASS (*site*, *static_context*)

(19) EXECUTION_SITE_IS_UNKNOWN (*site*)

(20) If *static_context* is a foreign execution image:
FOREIGN_EXECUTION_IMAGE_HAS_NO_SITE (*static_context*)

(21) FOREIGN_SYSTEM_IS_INVALID (site, *new_process*, HAS_EXECUTIVE_SYSTEM)

(22) INTERPRETER_IS_INTERPRETABLE (interpreter of *static_context*)

(23) INTERPRETER_IS_NOT_AVAILABLE (*static_context*)

(24) LABEL_IS_OUTSIDE_RANGE (*new_process*, the volume on which *new_process* would reside)

(25) LABEL_IS_OUTSIDE_RANGE (*new_process*, the would-be execution site of *new_process*)

(26) LIMIT_WOULD_BE_EXCEEDED (MAX_PROCESSES_PER_USER)

(27) REFERENCE_CANNOT_BE_ALLOCATED

(28) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL

(29) STATIC_CONTEXT_CONTENTS_CANNOT_BE_EXECUTED (*static_context*, *site*)

(30) STATIC_CONTEXT_IS_BEING_WRITTEN (*static_context*)

(31) STATIC_CONTEXT_REQUIRES_TOO_MUCH_MEMORY (*static_context*)

(32) If *process* is the calling process:
VOLUME_IS_FULL (calling process)

13.2.3 PROCESS_GET_WORKING_SCHEMA

(1)
```
PROCESS_GET_WORKING_SCHEMA(
    process             : [ Process_designator ]
)
    sds_sequence        : Name_sequence
```

(2) If no value is supplied for *process, process* designates the calling process.

(3) PROCESS_GET_WORKING_SCHEMA returns in *sds_sequence* the sequence of SDS names of the SDSs forming the working schema of the process *process*.

(4) If *process* is not the calling process a read lock of the default mode is established on *process*.

Errors

(5) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, READ, READ_LINKS)

(6) PROCESS_LACKS_REQUIRED_STATUS (*process*, (READY, RUNNING, SUSPENDED, STOPPED))

(7) PROCESS_IS_UNKNOWN (*process*)

13.2.4 PROCESS_INTERRUPT_OPERATION

(1)
```
PROCESS_INTERRUPT_OPERATION (
    process         : Process_designator
)
```

(2) PROCESS_INTERRUPT_OPERATION interrupts a process.

(3) There is no effect if *process* is not executing a PCTE operation; otherwise the interruption of all operations currently being executed by *process* is requested, with the following effect on *process*.

(4) After a period of time, each such interrupted operation of *process*, whether suspended or not, is terminated with the error OPERATION_IS_INTERRUPTED. For any waiting operation the corresponding "process_waiting_for" link is deleted.

(5) The time between the start of this operation and the end of the interruption of the operations of *process* is implementation-dependent.

Errors

(6) ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(7) PRIVILEGE_IS_NOT_GRANTED (PCTE_EXECUTION)

(8) PROCESS_LACKS_REQUIRED_STATUS (*process*, RUNNING)

(9) The following implementation-dependent error may be raised:
PROCESS_IS_THE_CALLER (*process*)

(10) PROCESS_IS_UNKNOWN (*process*)

(11) NOTE - This operation is intended to provide the means for a tool to control other tools, e.g. to cause an operation of a deadlocked tool to be abandoned, or to interrupt a tool which is not itself controlling the duration of operations.

13.2.5 PROCESS_RESUME

(1)
```
PROCESS_RESUME (
    process     : Process_designator
)
```

(2) PROCESS_RESUME resumes the suspended process *process*, by changing its status to RUNNING.

Errors

(3) ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(4) If the execution site of *process* is a foreign system:
FOREIGN_SYSTEM_IS_INVALID (execution site of *process*, *process*, SUPPORTS_IPC_AND_CONTROL)

(5) PROCESS_LACKS_REQUIRED_STATUS (*process*, SUSPENDED)

(6) PROCESS_IS_THE_CALLER (*process*)

(7) PROCESS_IS_UNKNOWN (*process*)

13.2.6 PROCESS_SET_ALARM

(1)
```
PROCESS_SET_ALARM (
    duration    : Natural
)
```

(2) PROCESS_SET_ALARM changes the time left until alarm of the calling process to *duration*.

Errors

(3) None.

13.2.7 PROCESS_SET_FILE_SIZE_LIMIT

(1)
```
PROCESS_SET_FILE_SIZE_LIMIT (
    process    : [ Process_designator ],
    fslimit    : Natural
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_FILE_SIZE_LIMIT changes the process file size limit of *process* to *fslimit*.

Errors

(4) If process is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(5) If *fslimit* is greater than the current value of the process file size limit of *process*:
PRIVILEGE_IS_NOT_GRANTED (PCTE_EXECUTION)

(6) PROCESS_IS_UNKNOWN (*process*)

13.2.8 PROCESS_SET_OPERATION_TIME_OUT

(1)
```
PROCESS_SET_OPERATION_TIME_OUT (
    duration   : Natural
)
```

(2) PROCESS_SET_OPERATION_TIME_OUT sets the process time-out of the calling process to *duration*.

Errors

(3) None.

13.2.9 PROCESS_SET_PRIORITY

(1)
```
PROCESS_SET_PRIORITY (
    process    : [ Process_designator ],
    priority   : Natural
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_PRIORITY sets the process priority of *process* to MAX_PRIORITY_VALUE if *priority* is greater than MAX_PRIORITY_VALUE, and to *priority* otherwise.

Errors

(4) If process is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(5) If *priority* is greater than the current value of the process priority of *process*:
PRIVILEGE_IS_NOT_GRANTED (PCTE_EXECUTION)

(6) PROCESS_IS_UNKNOWN (*process*)

13.2.10 PROCESS_SET_REFERENCED_OBJECT

(1)
```
PROCESS_SET_REFERENCED_OBJECT (
    process          : [ Process_designator ],
    reference_name   : Actual_key,
    object           : Object_designator
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_REFERENCED_OBJECT sets a referenced object of *process* to *object*.

(4) If a "referenced_object" link from *process* with the key *reference_name* already exists, its destination is changed to *object*. Otherwise, a "referenced_object" link from *process* to *object* with the key *reference_name* is created.

Errors

(5) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, APPEND_LINKS)

(6) If *process* is not the calling process and there is a referenced object *reference_name*:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_LINKS)

(7) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(8) PROCESS_IS_UNKNOWN (*process*)

(9) REFERENCE_NAME_IS_INVALID (*reference_name*)

(10) REFERENCED_OBJECT_IS_NOT_MUTABLE (*reference_name*)

(11) If *process* is the calling process:
VOLUME_IS_FULL (calling process)

13.2.11 PROCESS_SET_TERMINATION_STATUS

(1)
```
PROCESS_SET_TERMINATION_STATUS (
    termination_status : Integer
)
```

(2) PROCESS_SET_TERMINATION_STATUS provides a value *termination_status* to be stored in the process termination status of the calling process when it terminates, provided it is not terminated by PROCESS_TERMINATE with a termination status parameter.

Errors

(3) VOLUME_IS_FULL (calling process)

13.2.12 PROCESS_SET_WORKING_SCHEMA

(1)
```
PROCESS_SET_WORKING_SCHEMA (
    process          : [ Process_designator ],
    sds_sequence     : Name_sequence
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_WORKING_SCHEMA sets the working schema of a process according to the sequence of SDSs identified by the SDS names in *sds_sequence*, replacing the current working schema, if there is one.

(4) If *process* is the calling process:

(5) - the previous "sds_in_working_schema" links of *process* and the "in_working_schema_of" links from each previous SDS in working schema to *process* are deleted;

(6) - for each SDS *sds* identified by *sds_sequence*(I), an "sds_in_working_schema" link with key I (starting from I = 1) from *process* to *sds* and an "in_working_schema_of" link from *sds* to *process* are created.

(7) If *process* is not the calling process (and *process* is ready):

(8) - the previous "sds_in_working_schema" links of *process* are deleted;

(9) - for each SDS *sds* identified *sds_sequence*(I), an "sds_in_working_schema" link with key I (starting from I = 1) from *process* to *sds* is created.

(10) A new working schema is created as follows:

(11) - The sequence of SDS names is set to *sds_sequence.*

(12) - The set of types in working schema is constituted as follows:

(13) . a type in working schema is created for each type associated with a type in SDS in an SDS of *sds_sequence*;

(14) . the types in SDS of each created type in working schema are set to the types in SDS with the same associated type, and the composite names of those types in SDS;

(15) . the usage mode of each created type in working schema is set to the union of the usage modes of all its types in SDS;

(16) . the other properties of the created types in working schema are determined from their types in SDS (see 8.5).

Errors

(17) If *process* is not the calling process and an "sds_in_working_schema" link exists:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_LINKS)

(18) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, APPEND_LINKS)

(19) ACCESS_ERRORS (SDS with name in *sds_sequence*, ATOMIC, SYSTEM_ACCESS)

(20) If *process* is the calling process:
DISCRETIONARY_ACCESS_IS_NOT_GRANTED (SDS with name in *sds_sequence*, ATOMIC, EXPLOIT_SCHEMA)

(21) If *process* is not the calling process:
DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*process*, SDS with name in *sds_sequence*, ATOMIC, EXPLOIT_SCHEMA)

(22) LIMIT_WOULD_BE_EXCEEDED (MAX_SDS_IN_WORKING_SCHEMA)

(23) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(24) PROCESS_IS_UNKNOWN (*process*)

(25) If process is the calling process:
SDS_IS_UNDER_MODIFICATION (SDS with name in *sds_sequence*)

(26) SDS_IS_UNKNOWN (SDS with name in *sds_sequence*)

(27) SDS_WOULD_APPEAR_TWICE_IN_WORKING_SCHEMA (*sds_sequence*)

(28) If *process* is the calling process:
VOLUME_IS_FULL (calling process)

NOTES

(29) 1 A process need not have the predefined SDSs in its working schema in order to call operations except operations defined in clause 9 operating on objects or links with types and types in SDS defined in the predefined SDSs.

(30) 2 Setting the working schema is independent of activities. In order to maintain the integrity of working schemas, operations which affect the typing information contained in SDSs included in a working schema are explicitly prohibited and a working schema which contains an SDS with uncommitted modifications of the typing information may not be created.

13.2.13 PROCESS_START

(1)
```
PROCESS_START (
    process         : Process_designator,
    arguments       : String,
    environment     : String,
    site            : [ Execution_site_designator ],
    initial_status  : Initial_status
)
```

(2) PROCESS_START starts the execution of the static context or foreign execution image *static_context* of a process that has already been created.

(3) If *site* is supplied the "executed_on" link of *process* is replaced (if different) by one with destination *site*; otherwise *site* is the destination of that link

(4) The process status of *process* is changed to *initial_status*, provided *process* is ready.

(5) A link is created to *process* from each destination of the following links:

(6) - "in_working_schema_of" from each destination of "sds_in_working_schema";

(7) - "running_process" from the destination of "executed_on";

(8) - "opened_by" from each destination of "open_object";

(9) - "process_started_in" from the destination of "started_in_activity";

(10) - "user_identity_of" from the destination of "user_identity";

(11) - "adopted_user_group_of" from the destination of "adopted_user_group";

(12) - "consumer_process" from the destination of "consumer_identity", if any.

(13) These links are created even if any origin object is on a read-only volume or is a replicated copy.

(14) For each "open_object" link of *process*, the contents of the destination are opened and the current position in the object contents is shared with the process that created *process*. Furthermore, if the parent process of *process* is not the calling process, then:

(15) - a lock is acquired by the activity of the parent process on the opened object;

(16) - if the calling process has closed the object contents, then the current position of the opened object is determined in the same way as by CONTENTS_OPEN, in the opening mode of the "open_object" link.

(17) If *static_context* is interpretable, an "actual_interpreter" link is created to the interpreter of the interpretable static context, if it has one, else to the default interpreter of *process* if it has one, else to the default interpreter of the home object of *process*, provided there is a home object and it has a default interpreter.

(18) The "process_string_arguments" and "process_environment" attributes of *process* are set to *arguments* and *environment* respectively.

Errors

(19) ACCESS_ERRORS (*static_context*, ATOMIC, READ, EXECUTE)

(20) ACCESS_ERRORS (any interpreter, ATOMIC, READ, EXECUTE)

(21) ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(22) For each SDS *sds* which is the destination of an "in_working_schema_of" link from *process*:
ACCESS_ERRORS (*sds*, ATOMIC, SYSTEM_ACCESS)
DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*process*, *sds*, ATOMIC, EXPLOIT_SCHEMA)
SDS_IS_UNDER_MODIFICATION (*sds*)
SDS_IS_UNKNOWN (*sds*)

(23) For the execution site *site* which is the destination of an "executed_on" link from *process*:
ACCESS_ERRORS (*site*, ATOMIC, SYSTEM_ACCESS)

(24) For each open object *object* which is the destination of an "open_object" link from *process*:

(25) ACCESS_ERRORS (*object*, ATOMIC, SYSTEM_ACCESS)

(26) If the link's opening mode attribute is READ_ONLY or READ_WRITE,
DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*process*, *object*, ATOMIC, READ, READ_CONTENTS)

(27) If the link's opening mode attribute is WRITE_ONLY or READ_WRITE,
DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*process*, *object*, ATOMIC, READ, WRITE_CONTENTS)

(28) If the link's opening mode attribute is APPEND_ONLY,
DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*process*, *object*, ATOMIC, READ, APPEND_CONTENTS)

(29) For the activity *activity* which is the destination of the "started_in_activity" link from *process*:
ACCESS_ERRORS (*activity*, ATOMIC, SYSTEM_ACCESS)
ACTIVITY_STATUS_IS_INVALID (*activity*, ACTIVE)

(30) For the user *user* which is the destination of the "user_identity" link from *process*:
ACCESS_ERRORS (*user*, ATOMIC, SYSTEM_ACCESS)

(31) For the user group *group* which is the destination of the "adopted_user_group" link from *process*:
ACCESS_ERRORS (*group*, ATOMIC, SYSTEM_ACCESS)

(32) For the consumer group *group* which is the destination of the "consumer_identity" link from *process*:
ACCESS_ERRORS (*group*, ATOMIC, SYSTEM_ACCESS)

(33) ACCESS_ERRORS (parent process of *process*, ATOMIC, SYSTEM_ACCESS)

(34) EXECUTION_SITE_IS_INACCESSIBLE (*site*)

(35) EXECUTION_SITE_IS_NOT_IN_EXECUTION_CLASS (*site*, *static_context*)

(36) EXECUTION_SITE_IS_UNKNOWN (*site*)

(37) If *site* is a foreign system:
FOREIGN_SYSTEM_IS_INVALID (*site*, *process*, (HAS_EXECUTIVE_SYSTEM, SUPPORTS_EXECUTIVE_CONTROL, SUPPORTS_MONITOR))

(38) INTERPRETER_IS_INTERPRETABLE (interpreter of *static_context*)

(39) INTERPRETER_IS_NOT_AVAILABLE (*static_context*)

(40) LABEL_IS_OUTSIDE_RANGE (*process*, *site*)

(41) LIMIT_WOULD_BE_EXCEEDED (MAX_PROCESSES)

(42) PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(43) PROCESS_IS_UNKNOWN (*process*)

(44) STATIC_CONTEXT_CONTENTS_CANNOT_BE_EXECUTED (*static_context*, *site*)

(45) STATIC_CONTEXT_IS_BEING_WRITTEN (*static_context*)

(46) STATIC_CONTEXT_REQUIRES_TOO_MUCH_MEMORY (*static_context*)

13.2.14 PROCESS_SUSPEND

(1)
```
PROCESS_SUSPEND (
    process    : [ Process_designator ],
    alarm      : [ Natural ]
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SUSPEND suspends a running process.

(4) PROCESS_SUSPEND changes the status of *process* to SUSPENDED, provided it already has the value RUNNING; and if *alarm* is supplied and *process* is the calling process sets the value of time left until alarm to *alarm*.

(5) If time left until alarm is non-zero, it defines a maximum duration in seconds for the suspension of the process. If this duration expires, the process receives an implementation-dependent alarm message of message type WAKE (provided it has reserved a message queue and is the listened-to process for the message queue) and is resumed.

Errors

(6) If process is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(7) If the execution site of *process* is a foreign system:
FOREIGN_SYSTEM_IS_INVALID (execution site of *process*, *process*, SUPPORTS_IPC_AND_CONTROL)

(8) PROCESS_LACKS_REQUIRED_STATUS (*process*, RUNNING)

(9) If *alarm* is supplied:
PROCESS_IS_NOT_THE_CALLER (*process*)

(10) PROCESS_IS_UNKNOWN (*process*)

13.2.15 PROCESS_TERMINATE

(1)
```
PROCESS_TERMINATE (
    process              : [ Process_designator ],
    termination_status   : [ Integer ]
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_TERMINATE terminates a process. In certain conditions this results in the (composite) deletion of the process or its parent.

(4) Any ongoing operations invoked from *process* are interrupted in the same way as by PROCESS_INTERRUPT_OPERATION.

(5) PROCESS_TERMINATE changes the links and attributes of *process* as follows:

(6) - "process_status" is set to TERMINATED;

(7) - "process_termination_time" is set to the current time;

(8) - "process_termination_status" is set to *termination_status*, if supplied, otherwise to EXIT_ERROR if *process* is the calling process or FORCED_TERMINATION if not.

(9) Destinations of "open_object" links from *process* are closed.

(10) The following links from *process* and their reverse links are deleted:

(11) - "sds_in_working_schema" links and their reverse "in_working_schema_of" links;

(12) - "executed_on" links and their reverse "running_process" links;

(13) - "opened_object" links and their reverse "opened_by" links;

(14) - "reserved_message_queue" links and their reverse "reserved_by" links;

(15) - "user_identity" links and their reverse "user_identity_of" links;

(16) - "adopted_user_group" links and their reverse "adopted_user_group_of" links;

(17) - "consumer_identity" links and their reverse "consumer_process" links.

(18) The "adoptable_user_group" links from *process* are deleted.

(19) If the parent of *process* is waiting for termination of *process* then the parent of *process* discontinues waiting and "acknowledged_termination" of *process* is set to **true**.

(20) If "deletion_upon_termination" and "acknowledged_termination" of *process* are both **true**, all component processes of *process* are ready or terminated, and the conditions for the object deletion of *process* hold (see 9.3.5), then *process* is deleted.

(21) If the parent of *process* is terminated, "deletion_upon_termination" and "acknowledged_termination" of the parent of *process* are both **true**, all component processes of the parent of *process* are ready or terminated, and the conditions for the object deletion of the parent of *process* hold (see 9.3.5), then the parent of *process* is deleted.

(22) If deletion of a "process" object is not possible, then the "child_process" link remains.

(23) If an activity is initiated by a process and no corresponding ACTIVITY_ABORT or ACTIVITY_END call is made before termination of the process, then an ACTIVITY_END call is implied if the termination status of the terminating process is EXIT_SUCCESS and an ACTIVITY_ABORT call is implied otherwise.

(24) When a process is started, and "deletion_upon_termination" is **true**, then the activity in which the process starts acquires a default delete lock on the process. If "delete_upon_termination" is **false** when the process starts and is set **true** while the process is running then the delete lock is acquired at that point.

Errors

(25) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, (WRITE_ATTRIBUTES, WRITE_LINKS))

(26) PROCESS_LACKS_REQUIRED_STATUS (*process*, (READY, RUNNING, STOPPED, SUSPENDED))

(27) PROCESS_IS_INITIAL_PROCESS (*process*)

(28) PROCESS_IS_UNKNOWN (*process*)

13.2.16 PROCESS_UNSET_REFERENCED_OBJECT

(1)
```
PROCESS_UNSET_REFERENCED_OBJECT (
    process          : [ Process_designator ],
    reference_name   : Actual_key
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_UNSET_REFERENCED_OBJECT unsets a referenced object of *process*.

(4) If there is no "referenced_object" link from *process* with key *reference_name*, the operation has no effect. Otherwise, the "referenced_object" link from *process* with key *reference_name* is deleted.

Errors

(5) If *process* is not the calling process and there is a "referenced_object" link from *process* with key *reference_name*:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_LINKS)

(6) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(7) PROCESS_IS_UNKNOWN (*process*)

(8) REFERENCE_NAME_IS_INVALID (*reference_name*)

(9) REFERENCED_OBJECT_IS_NOT_MUTABLE (*reference_name*)

13.2.17 PROCESS_WAIT_FOR_ANY_CHILD

(1)
```
PROCESS_WAIT_FOR_ANY_CHILD (
)
    termination_status : Integer,
    child              : Natural
```

(2) PROCESS_WAIT_FOR_ANY_CHILD sets the calling thread of the calling process waiting until any of its child processes has terminated.

(3) If any child of the calling process has process status TERMINATED and acknowledged termination **false**, the acknowledged termination of the terminated child process is set to **true** and that child process is deleted if it has deletion upon termination **true** and the deletion conditions (see 13.2.15) are satisfied. If more than one child process fulfils the condition, one is selected in an implementation-defined manner to fill the role of terminated child process.

(4) If no child of the calling process has process status TERMINATED, the operation waits and a "process_waiting_for" link is created to the calling process with waiting type WAITING_FOR_TERMINATION. The operation continues when the process status of any child process changes to TERMINATED.

(5) The process termination status of the terminated child is returned in *termination_status*, unless the confidentiality label of the calling process does not dominate that of the terminated child process or the integrity label of the calling process is not dominated by that of the terminated

child process, in which cases *termination_status* is set to UNAVAILABLE (a binding-defined value different from the named values of the "process_termination_status" attribute). The key of the "child_process" link from the calling process to the terminated child is returned in *child*.

Errors

(6) DISCRETIONARY_ACCESS_IS_NOT_GRANTED (the terminated child process, ATOMIC, WRITE_ATTRIBUTES)

(7) PROCESS_HAS_NO_UNTERMINATED_CHILD

13.2.18 PROCESS_WAIT_FOR_CHILD

(1)
```
PROCESS_WAIT_FOR_CHILD (
    child              : Process_designator
)
    termination_status : Integer
```

(2) PROCESS_WAIT_FOR_CHILD sets the calling thread of the calling process waiting until the nominated child process has terminated.

(3) If *child* has process status TERMINATED and acknowledged termination **false**, the acknowledged termination of *child* is set to **true** and *child* is deleted if it has deletion upon termination **true** and the deletion conditions (see 13.2.15) are satisfied.

(4) Otherwise, the operation waits and a "process_waiting_for" link is created to *child* with waiting type WAITING_FOR_TERMINATION. The operation continues when the process status of *child* changes to TERMINATED.

(5) PROCESS_WAIT_FOR_CHILD returns the process termination status of *child* in *termination_status*, unless the confidentiality label of the calling process does not dominate that of *child* or the integrity label of the calling process is not dominated by that of *child*, in which cases *termination_status* is set to UNAVAILABLE (a binding-defined value different from the named values of the "process_termination_status" attribute).

Errors

(6) DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*child*, ATOMIC, WRITE_ATTRIBUTES)

(7) PROCESS_IS_NOT_TERMINABLE_CHILD (*child*)

(8) PROCESS_IS_UNKNOWN (*child*)

(9) PROCESS_TERMINATION_IS_ALREADY_ACKNOWLEDGED (*child*)

13.3 Security operations

13.3.1 PROCESS_ADOPT_USER_GROUP

(1)
```
PROCESS_ADOPT_USER_GROUP (
    process     : [ Process_designator ],
    user_group  : User_group_designator
)
```

(2) If no value is supplied for *process, process* designates the calling process.

(3) PROCESS_ADOPT_USER_GROUP changes the adopted user group of *process* to *user_group*.

(4) Let G be *user_group,* P be *process,* and G' be the previous adopted user group.

(5) If *process* is the calling process:

(6) - The following links are deleted:

(7) . "adopted_user_group" from P to G';

(8) . "adopted_user_group_of" from G' to P;

(9) . "adoptable_user_group" from P to G.

(10) - The following links are created, setting the key values to the next available natural in each case:

(11) . "adopted_user_group" from P to G;

(12) . "adopted_user_group_of" from G to P;

(13) . "adoptable_user_group" from P to G'.

(14) - The effective security groups of the process are changed to consist of:

(15) . The user *user* of *process* (no change);

(16) . *user_group*;

(17) . all the supergroups of *user_group;*

(18) . all the program groups to which the static contexts run or executed by the calling process belong (no change);

(19) . all the supergroups to which these program groups belong (no change).

(20) If *process* is not the calling process (and is ready):

(21) - The following links are deleted:

(22) . "adopted_user_group" from P to G';

(23) . "adoptable_user_group" from P to G.

(24) - The following links are created, setting the key values to the next available natural in each case:

(25) . "adopted_user_group" from P to G;

(26) . "adoptable_user_group" from P to G'.

(27) If P is the calling process, there is a "consumer_identity" link from P to a consumer group object C, and if the new effective security groups are such that EVALUATE_PROCESS (P, C, EXPLOIT_CONSUMER_IDENTITY) is **false** (see 19.1.2), then the "consumer_identity" link from P to C and the "consumer_process" link from C to P are deleted.

(28) The working schema of *process* is reset to empty by deleting all "sds_in_working_schema" links from *process* and their reverse "in_working_schema_of" links.

Errors

(29) Access errors are determined on the basis of the discretionary context in force before the change in the effective security groups which this operation produces.

(30) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, APPEND_LINKS)

(31) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_LINKS)

(32) OBJECT_IS_INACCESSIBLE (*user_group*, ATOMIC)

(33) OBJECT_IS_INACCESSIBLE (G', ATOMIC)

(34) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(35) PROCESS_IS_UNKNOWN (*process*)

(36) SECURITY_GROUP_IS_NOT_ADOPTABLE (*user_group*)

(37) SECURITY_GROUP_IS_UNKNOWN (*user_group*)

(38) USER_IS_NOT_MEMBER (*user*, *user_group*)

(39) If *process* is the calling process:
VOLUME_IS_FULL (calling process)

NOTES

(40) 1 This operation changes the user group which is currently adopted by the designated process, and therefore changes the role in which the user is acting.

(41) 2 Users may be removed from user groups at any time. It is therefore necessary to check that the current user is still a member of the designated user group before adopting it. It is insufficient to rely on the "adoptable_user_group" links.

13.3.2 PROCESS_GET_DEFAULT_ACL

(1)
```
PROCESS_GET_DEFAULT_ACL (
)
    acl : Acl
```

(2) PROCESS_GET_DEFAULT_ACL returns the default atomic ACL of the calling process as *acl*.

Errors

(3) None.

13.3.3 PROCESS_GET_DEFAULT_OWNER

(1)
```
PROCESS_GET_DEFAULT_OWNER (
)
    group   : Group_identifier
```

(2) PROCESS_GET_DEFAULT_OWNER returns the group identifier of the default object owner of the calling process as *group*.

Errors

(3) None.

13.3.4 PROCESS_SET_ADOPTABLE_FOR_CHILD

(1)
```
PROCESS_SET_ADOPTABLE_FOR_CHILD (
    process        : [ Process_designator ],
    user_group     : User_group_designator,
    adoptability   : Boolean
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_ADOPTABLE_FOR_CHILD changes the "adoptable_for_child" attribute of the "adoptable_user_group" link from *process* to *user_group* to *adoptability*.

Errors

(4) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_LINKS)

(5) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(6) PROCESS_IS_UNKNOWN (*process*)

(7) SECURITY_GROUP_IS_UNKNOWN (*user_group*)

(8) SECURITY_GROUP_IS_NOT_ADOPTABLE (*user_group*, *process*)

13.3.5 PROCESS_SET_DEFAULT_ACL_ENTRY

(1)
```
PROCESS_SET_DEFAULT_ACL_ENTRY (
    process     : [ Process_designator ],
    group       : Group_identifier,
    modes       : Atomic_access_rights
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_DEFAULT_ACL_ENTRY changes the default atomic ACL of the process *process*.

(4) The ACL entry for *group* in the "default_atomic_acl" attribute of *process* is set to *modes*.

Errors

(5) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(6) DEFAULT_ACL_WOULD_BE_INVALID (*process*, *group*, *modes*)

(7) DEFAULT_ACL_WOULD_BE_INCONSISTENT_WITH_DEFAULT_OBJECT_OWNER (*process*, *group*)

(8) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(9) PROCESS_IS_UNKNOWN (*process*)

(10) SECURITY_GROUP_IS_UNKNOWN (*group*)

13.3.6 PROCESS_SET_DEFAULT_OWNER

(1)
```
PROCESS_SET_DEFAULT_OWNER (
    process     : [ Process_designator ],
    group       : Group_identifier
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_DEFAULT_OWNER changes the default object owner of *process* to the security group identifier *group*.

Errors

(4) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(5) DEFAULT_ACL_WOULD_BE_INCONSISTENT_WITH_DEFAULT_OBJECT_OWNER (*process*, *group*)

(6) PROCESS_IS_UNKNOWN (*process*)

(7) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(8) SECURITY_GROUP_IS_UNKNOWN (*group*)

13.3.7 PROCESS_SET_USER

(1)
```
PROCESS_SET_USER (
    user            : User_designator,
    user_group      : User_group_designator
)
```

(2) PROCESS_SET_USER sets the user of the calling process to *user* and changes the adopted user group of the calling process to *user_group*.

(3) Let P be the calling process, U be the previous user of the process, G be the previous adopted user group, U' be *user*, and G' be *user_group*.

(4) The following links are deleted:

(5) - "user_identity" from P to U;

(6) - "user_identity_of" from U to P;

(7) - "adopted_user_group" from P to G;

(8) - "adopted_user_group_of" from G to P;

(9) - "adoptable_user_group" from P to the set of user groups currently so linked, excluding G.

(10) The following links are created, setting the key values to the next available integer in each case:

(11) - "user_identity" from P to U';

(12) - "user_identity_of" from U' to P;

(13) - "adopted_user_group" from P to G';

(14) - "adopted_user_group_of" from G' to P;

(15) - "adoptable_user_group" from P to each user group of which U' is a member, excluding G'.

(16) The effective security groups of the process are changed to consist of:

(17) - *user*;

(18) - *user_group*;

(19) - all the supergroups of *user_group*;

(20) - all the program groups to which the static contexts run or executed by the calling process belong, unchanged;

(21) - all the supergroups to which these program groups belong (no change).

(22) Let W be the execution site of P and V be the volume on which P resides, and let P' be P as updated by the operation. The confidentiality label of P is set to the confidentiality label C which is the conjunction of confidentiality low label of W and the confidentiality low label of V, providing that the following are all **true** (see 20.1.3, 20.1.4):

(23) LABEL_DOMINATES (confidentiality clearance of *user*, C)

(24) CONFIDENTIALITY_LABEL_WITHIN_RANGE (P', W)

(25) CONFIDENTIALITY_LABEL_WITHIN_RANGE (P', V)

(26) The integrity context of P is set to the integrity label I which is the disjunction of the user's integrity clearance, the integrity high label of W, and the integrity high label of V, providing that the following are all **true**:

(27) LABEL_DOMINATES (I, integrity clearance of *user*)

(28) INTEGRITY_LABEL_WITHIN_RANGE (P', W)

(29) INTEGRITY_LABEL_WITHIN_RANGE (P', V)

(30) If there is a link of type "consumer_identity" from P to a consumer group C, and if the new effective security groups are such that

EVALUATE_PROCESS (P, C, EXPLOIT_CONSUMER_IDENTITY)

is **false**, then the "consumer_identity" link from P to C and the "consumer_process" link from C to P are deleted.

(31) The working schema of the calling process is reset to its initial value (empty) by deleting all "sds_in_working_schema" links from the calling process and their reverse "in_working_schema_of" links.

Errors

(32) Error conditions are determined on the basis of the discretionary context in force before the change in the effective security groups which this operation produces.

(33) ACCESS_ERRORS (*user*, ATOMIC, SYSTEM_ACCESS)

(34) ACCESS_ERRORS (*user_group,* ATOMIC, SYSTEM_ACCESS)

(35) OBJECT_IS_INACCESSIBLE (U, ATOMIC)

(36) OBJECT_IS_INACCESSIBLE (G, ATOMIC)

(37) PRIVILEGE_IS_NOT_GRANTED (PCTE_SECURITY)

(38) PROCESS_LABELS_WOULD_BE_INCOMPATIBLE (*user*)

(39) SECURITY_GROUP_IS_UNKNOWN (*user*)

(40) SECURITY_GROUP_IS_UNKNOWN (*user_group*)

(41) USER_IS_NOT_MEMBER (*user, user_group*)

(42) NOTE - This operation establishes the user on behalf of which the current process will run, and the role in which the user will act. It is intended to be used by the user authentication tool.

13.4 Profiling operations

13.4.1 PROCESS_PROFILING_OFF

(1)
```
PROCESS_PROFILING_OFF (
    handle : Profile_handle
)
    buffer   : Buffer
```

(2) PROCESS_PROFILING_OFF terminates the profiling of the calling process initiated with the profile handle *handle*, and returns the results in *buffer*.

Errors

(3) PROFILING_IS_NOT_SWITCHED_ON (*handle*)

13.4.2 PROCESS_PROFILING_ON

(1)
```
PROCESS_PROFILING_ON (
    start   : Address,
    end     : Address,
    count   : Natural
)
    handle : Profile_handle
```

(2) PROCESS_PROFILING_ON initiates profiling of the calling process. Profiling is an implementation-defined action. It continues until the operation PROCESS_PROFILING_ OFF is called with the returned profile handle *handle* or the calling process terminates.

(3) If profiling is already initiated for the calling process, it is reinitiated with a new profiling buffer identified by the returned profile handle.

Errors

(4) MEMORY_REGION_IS_NOT_IN_PROFILING_SPACE (*start*, *end*)

(5) NOTE - Profiling is implementation-defined but is intended to provide in each element of a profiling buffer identified by *handle* a count of the number of times the process was executing at, or accessing, a memory address associated with that element. *start* and *end* specify a region of the process memory to be profiled: the mapping to elements of the buffer is implementation-defined. Other calls of PROCESS_PROFILING_ON can request profiling into other buffers.

13.5 Monitoring operations

13.5.1 PROCESS_ADD_BREAKPOINT

(1)
```
PROCESS_ADD_BREAKPOINT (
    process     : Process_designator,
    breakpoint  : Address
)
```

(2) PROCESS_ADD_BREAKPOINT adds a breakpoint for *process*. The effect is implementation-defined.

Errors

(3) ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_CONTENTS)

(4) MEMORY_ADDRESS_IS_OUT_OF_PROCESS (*breakpoint*, *process*)

(5) PROCESS_LACKS_REQUIRED_STATUS (*process*, STOPPED)

(6) PROCESS_IS_NOT_CHILD (*process*)

(7) PROCESS_IS_UNKNOWN (*process*)

(8) NOTE - The format of a breakpoint is implementation-defined but it is intended to define an instruction or data address, access to which will cause the accessing thread of *process* to stop.

13.5.2 PROCESS_CONTINUE

(1)
```
PROCESS_CONTINUE (
    process   : Process_designator
)
```

(2) PROCESS_CONTINUE continues any stopped threads of a process.

(3) The status of *process* is set to RUNNING.

Errors

(4) ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_CONTENTS)

(5) PROCESS_LACKS_REQUIRED_STATUS (*process*, STOPPED)

(6) PROCESS_IS_NOT_CHILD (process)

(7) PROCESS_IS_UNKNOWN (process)

13.5.3 PROCESS_PEEK

(1)
```
PROCESS_PEEK (
    process   : Process_designator,
    address   : Address
)
    value     : Process_data
```

(2) PROCESS_PEEK returns as *value* the contents at *address* of *process*.

Errors

(3) ACCESS_ERRORS (*process*, ATOMIC, READ, READ_CONTENTS)

(4) MEMORY_ADDRESS_IS_OUT_OF_PROCESS (*address*, *process*)

(5) PROCESS_LACKS_REQUIRED_STATUS (*process*, STOPPED)

(6) PROCESS_IS_NOT_CHILD (*process*)

(7) PROCESS_IS_UNKNOWN (*process*)

13.5.4 PROCESS_POKE

(1)
```
PROCESS_POKE (
    process   : Process_designator,
    address   : Address,
    value     : Process_data
)
```

(2) PROCESS_POKE modifies *process* at *address* to *value*.

Errors

(3) ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_CONTENTS)

(4) MEMORY_ADDRESS_IS_OUT_OF_PROCESS (*address*, *process*)

(5) PROCESS_LACKS_REQUIRED_STATUS (*process*, STOPPED)

(6) PROCESS_IS_NOT_CHILD (*process*)

(7) PROCESS_IS_UNKNOWN (*process*)

13.5.5 PROCESS_REMOVE_BREAKPOINT

(1)
```
PROCESS_REMOVE_BREAKPOINT (
    process    : Process_designator,
    breakpoint : Address
)
```

(2) PROCESS_REMOVE_BREAKPOINT removes a breakpoint *breakpoint* of process *process*.

Errors

(3) ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_CONTENTS)

(4) BREAKPOINT_IS_NOT_DEFINED (*breakpoint*)

(5) PROCESS_LACKS_REQUIRED_STATUS (*process*, STOPPED)

(6) PROCESS_IS_NOT_CHILD (*process*)

(7) PROCESS_IS_UNKNOWN (*process*)

13.5.6 PROCESS_WAIT_FOR_BREAKPOINT

(1)
```
PROCESS_WAIT_FOR_BREAKPOINT (
    process    : Process_designator
)
    breakpoint : Address
```

(2) PROCESS_WAIT_FOR_BREAKPOINT sets the calling thread of the calling process waiting until the process *process* is stopped or terminated.

(3) If *process* has process status TERMINATED, the error condition PROCESS_LACKS_REQUIRED_STATUS occurs.

(4) Otherwise, a "process_waiting_for" link is created to *process* with waiting type WAITING_FOR_TERMINATION. The operation waits until *process* reaches a breakpoint, in which case the breakpoint is returned in *breakpoint*, or the process status of *process* changes to TERMINATED, in which case the error condition PROCESS_IS_TERMINATED occurs.

Errors

(5) ACCESS_ERRORS (*process*, ATOMIC, MODIFY, READ_CONTENTS)

(6) PROCESS_IS_NOT_CHILD (*process*)

(7) PROCESS_LACKS_REQUIRED_STATUS (*process*, (READY, RUNNING, STOPPED, SUSPENDED))

(8) PROCESS_IS_UNKNOWN (*process*)

14 Message queues

14.1 Message queue concepts

(1)
```
Message ::
    DATA          : seq of Octet
    MESSAGE_TYPE  : Message_type
```

(2)
```
Received_message ::
    MESSAGE    : Message
    POSITION   : Natural
```

(3)
```
Message_type = Standard_message_type | Notification_message_type
    | Implementation_defined_message_type | Undefined_message_type
```

(4)
```
Message_types = set of Message_type | ALL_MESSAGE_TYPES
```

(5)
```
Standard_message_type = interrupt | quit | finish | suspend | end | abort |
    deadlock | wake
```

(6)
```
Implementation_defined_message_type :: Token
```

(7)
```
Undefined_message_type :: Token
```

(8)
```
Handler :: Token
```

(9)
```
sds system:
```

(10)
```
message_queue: child type of object with
attribute
    reader_waiting: (read) non_duplicated boolean;
    writer_waiting: (read) non_duplicated boolean;
    space_used: (read) non_duplicated natural;
    total_space: (read) natural;
    message_count: (read) non_duplicated natural;
    last_send_time: (read) non_duplicated time;
    last_receive_time: (read) non_duplicated time;
link
    reserved_by: (navigate) non_duplicated designation link to process;
    listened_to: (navigate) non_duplicated designation link to process;
    notifier: (navigate) non_duplicated designation link (notifier_key: natural) to
        object with
    attribute
        modification_event: (read) boolean;
        change_event: (read) boolean;
        delete_event: (read) boolean;
        move_event: (read) boolean;
    end notifier;
end message_queue;
```

(11)
```
end system;
```

(12) Messages and message queues allow processes to communicate. A message queue has an associated sequence of messages. A message contains data, and has a message type. The space occupied by a message is implementation-defined. For notification message types see 15.1.2.

(13) Implementation_defined_message_type and Undefined_message_type are implementation-defined types disjoint from each other and from Standard_message_type and Notification_message_type. The meanings of implementation-defined message types are implementation-defined. For the intended meanings of standard message types see Note 3 below.

(14) The value ALL_MESSAGE_TYPES denotes the set of all message types, including implementation-defined and undefined message types.

(15) Each message in a message queue is assigned a *position number* which it retains while it is in the queue. The position numbers are positive naturals, monotonically increasing with time of arrival in the queue but otherwise implementation-dependent.

(16) Reader waiting is **true** if and only if one or more processes are waiting to receive a message.

(17) Writer waiting is **true** if and only if one or more processes are waiting to send a message.

(18) The space used is the space currently required by the message queue to hold its messages, in octets.

(19) The total space is the maximum possible size of the space used. This may vary between message queues and is initialized to an implementation-defined value which must not be less than four times MAX_MESSAGE_SIZE (see clause 24). An implementation may place an upper limit on the total space of a message queue; this must not be less than MAX_MESSAGE_QUEUE_SPACE (see clause 24).

(20) The message count is the number of messages in the message queue.

(21) The last send time and last receive time record the system time on the last occasion that a message was sent to the queue and received from the queue, respectively. Initially they are equal to the default initial value for time attributes. If the last sent message was sent through MESSAGE_SEND_WAIT, the last send time is the system time when the message actually entered the queue (at the end of the waiting period). If the last received message was received through MESSAGE_RECEIVE_WAIT, last receive time is the system time when the message was actually received from the queue (at the end of the waiting period).

(22) The destination of the "reserved_by" link, if any, is called the *reserving* process of the message queue; it is also said to have *reserved* the message queue. The reserving process is the only process which can receive or peek messages from the message queue, and it must have adequate read access permission to the message queue. If there is no reserving process then any process can receive or peek messages, subject to access permission.

(23) The destination of the "listened_to" link, if any, is the reserving process (which must exist), and indicates that the reserving process has an associated procedure which is executed on the raising of a message queue event by the appearance of a message of one of a specified set of message types in the message queue. Such a procedure is called a *handler*. In this case the reserving process is called the *listening process* of the message queue. The "listened_to" link is reversed by an "is_listener" link, with an attribute which defines the message types of messages for which the handler is executed. The handler is invoked with a single argument, which denotes the affected message queue in a binding-defined manner. The types of a handler and of its argument are binding-defined.

(24) Notifiers are described in 15.1.

NOTES

(25) 1 An implicit modification of predefined attributes of a message queue does not require a write lock on the message queue.

(26) 2 The intended meanings of the standard message types are as follows.

(27) - INTERRUPT: user interruption;

(28) - QUIT: user wants to quit;

(29) - FINISH: the receiving process should terminate;

(30) - SUSPEND: the receiving process should suspend itself;

(31) - END: the current activity of the receiving process should be normally terminated;

(32) - ABORT: the current activity of the receiving process should be abnormally terminated;

(33) - DEADLOCK: deadlock has been detected;

(34) - WAKE: the receiving process's time left until alarm has expired (see 13.1.4).

(35) 3 A process can send messages to itself. A message queue can have several concurrent readers if there is no reserving process. A message queue can have several concurrent writers. If more than one process is eligible to receive a message, it is not defined which of the eligible processes receives it.

(36) 4 The associated sequences of messages of message queues are not affected by transaction rollback.

(37) 5 The handler is executed and the listening process status is changed to RUNNING, even if the listening process is suspended — see 14.1.

14.2 Message queue operations

14.2.1 MESSAGE_DELETE

(1)
```
MESSAGE_DELETE (
    queue       : Message_queue_designator,
    position    : Natural
)
```

(2) MESSAGE_DELETE removes the message with message number *position* from the message queue *queue*. The space used of *queue* is decremented by the space used by the removed message, and the message count of *queue* is decremented by 1.

(3) A read lock of the default mode is obtained on *queue*.

(4) A read lock of the default mode is obtained on *queue*.

Errors

(5) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, WRITE_CONTENTS)

(6) MESSAGE_POSITION_IS_NOT_VALID (*position, queue*)

(7) MESSAGE_QUEUE_IS_RESERVED (*queue*)

14.2.2 MESSAGE_PEEK

(1)
```
MESSAGE_PEEK (
    queue       : Message_queue_designator,
    types       : Message_types,
    position    : [ Natural ]
)
    message     : [ Received_message ]
```

(2) MESSAGE_PEEK reads a message from the message queue *queue* without removing it from *queue*. *types* specifies the set of acceptable message types. *position* specifies a position in *queue*; if it is 0 or not supplied, the position is the beginning of *queue*; otherwise it is the position immediately before the message with position number *position*.

(3) If *queue* contains no messages of an acceptable message type after the specified position, then no message is returned. Otherwise a copy of the next message of an acceptable message type after the specified position is returned. In either case *queue* is unchanged.

Errors

(4) ACCESS_ERRORS (*queue*, ATOMIC, READ, READ_CONTENTS)

(5) MESSAGE_POSITION_IS_NOT_VALID (*position, queue*)

(6) MESSAGE_QUEUE_IS_RESERVED (*queue*)

14.2.3 MESSAGE_RECEIVE_NO_WAIT

(1)
```
MESSAGE_RECEIVE_NO_WAIT (
    queue      : Message_queue_designator,
    types      : Message_types,
    position   : [ Natural ]
)
    message    : [ Received_message ]
```

(2) MESSAGE_RECEIVE_NO_WAIT reads and removes a message from the message queue *queue*, but does not wait if there is no message of an acceptable message type in *queue*. *types* specifies the set of acceptable message types. *position* specifies a position in *queue*; if it is 0 or not supplied, the position is the beginning of the *queue*; otherwise it is the position immediately before the message with position number *position*.

(3) If the *queue* contains no messages of an acceptable message type after the specified position, then no message is returned. Otherwise the first message of an acceptable type after the specified position is returned, and that message is removed from *queue*. The last receive time of *queue* is set to the system time, the space used of *queue* is decremented by the space used of the removed message, and the message count of *queue* is decremented by 1.

(4) A read lock of the default mode is obtained on *queue*.

Errors

(5) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, WRITE_CONTENTS)

(6) MESSAGE_POSITION_IS_NOT_VALID (*position*, *queue*)

(7) MESSAGE_QUEUE_IS_RESERVED (*queue*)

(8) MESSAGE_TYPES_NOT_FOUND_IN_QUEUE (*queue*, *types*, *position*)

14.2.4 MESSAGE_RECEIVE_WAIT

(1)
```
MESSAGE_RECEIVE_WAIT (
    queue      : Message_queue_designator,
    types      : Message_types,
    position   : [ Natural ]
)
    message    : Received_message
```

(2) MESSAGE_RECEIVE_WAIT reads and removes a message from the message queue *queue*, waiting if necessary for a message of an acceptable message type to arrive. *types* specifies the set of acceptable message types. *position* specifies a position in the message queue *queue*; if it is 0 or not supplied, the position is the beginning of the *queue*; otherwise it is the position immediately before the message with position number *position*.

(3) If the message queue *queue* contains one or more messages of an acceptable message type after the specified position, the first such message is returned, and that message is removed from the queue. The last receive time of *queue* is set to the system time, the space used of *queue* is decremented by the space used of the removed message, and the message count of *queue* is decremented by 1.

(4) If *queue* contains no messages of any acceptable message type after the specified position, then the operation waits and reader waiting for *queue* is set to **true**, until one of the following happens.

(5) - A message of an acceptable type is placed on the queue. If the calling process is the listening process for *queue* and the message type of the message is one of the specified set of message types for the associated handler, then the handler is executed; otherwise the operation proceeds as described above.

(6) - A reserved message queue of the calling process receives a message of message type WAKE. The error condition MESSAGE_QUEUE_HAS_BEEN_WOKEN then holds.

(7) - The message queue is removed from the object base. The error condition MESSAGE_QUEUE_HAS_BEEN_DELETED then holds.

(8) - The caller is denied mandatory read access, mandatory write access, or WRITE_CONTENTS discretionary access to *queue*. The error condition CONFIDENTIALITY_WOULD_BE_
VIOLATED or INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED then holds in the first case, CONFIDENTIALITY_CONFINEMENT_WOULD_BE_VIOLATED or INTEGRITY_WOULD_BE_VIOLATED in the second case, and DISCRETIONARY_
ACCESS_IS_NOT_GRANTED in the third case (all under ACCESS_ERRORS).

(9) - The message queue becomes reserved by another process. The error condition MESSAGE_QUEUE_IS_RESERVED then holds.

(10) A read lock of the default mode is obtained on *queue*.

Errors

(11) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, WRITE_CONTENTS)

(12) MESSAGE_POSITION_IS_NOT_VALID (*position*, *queue*)

(13) MESSAGE_QUEUE_IS_RESERVED (*queue*)

(14) MESSAGE_QUEUE_HAS_BEEN_DELETED (*queue*)

(15) MESSAGE_QUEUE_HAS_BEEN_WOKEN (*queue*)

14.2.5 MESSAGE_SEND_NO_WAIT

(1)
```
MESSAGE_SEND_NO_WAIT (
    queue      : Message_queue_designator,
    message    : Message
)
```

(2) MESSAGE_SEND_NO_WAIT appends the message *message* to the message queue *queue*.

(3) The last send time of *queue* is set to the system time. The space used of *queue* is incremented by the space used by *message*. The message count of *queue* is incremented by 1.

(4) A read lock of the default mode is obtained on *queue*.

Errors

(5) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, APPEND_CONTENTS)

(6) LIMIT_WOULD_BE_EXCEEDED (MAX_MESSAGE_SIZE)

(7) MESSAGE_QUEUE_WOULD_BE_TOO_BIG (*queue*)

14.2.6 MESSAGE_SEND_WAIT

(1)
```
MESSAGE_SEND_WAIT (
    queue      : Message_queue_designator,
    message    : Message
)
```

(2) MESSAGE_SEND_WAIT appends the message *message* to the message queue *queue*, *waiting if necessary until* queue has enough space for it.

(3) If the space used of the message queue *queue* would not exceed the total space of *queue*, the message *message* is appended to *queue*. The last send time of *queue* is set to the system time when the message is sent (at the end of the waiting period, if any). The space used of *queue* is incremented by the space used by *message*. The message count of *queue* is incremented by 1.

(4) If the space used of *queue* would exceed the total space of *queue*, then writer waiting of *queue* is set to **true** and the operation waits until one of the following occurs.

(5) - The space used of *queue* would no longer exceed the total space of *queue*. The operation then proceeds as described above.

(6) - The calling process receives a message. of message type WAKE. The error condition MESSAGE_QUEUE_HAS_BEEN_WOKEN then holds.

(7) - The message queue is removed from the object base. The error condition MESSAGE_QUEUE_HAS_BEEN_DELETED then holds.

(8) - The caller is denied mandatory read access, mandatory write access, or APPEND_CONTENTS discretionary access. The error condition CONFIDENTIALITY_ WOULD_BE_VIOLATED or INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED then holds in the first case, CONFIDENTIALITY_CONFINEMENT_WOULD_BE_ VIOLATED or INTEGRITY_WOULD_BE_VIOLATED in the second case, and DISCRETIONARY_ACCESS_IS_NOT_GRANTED in the third case (all under ACCESS_ERRORS).

(9) A read lock of the default mode is obtained on *queue*.

Errors

(10) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, APPEND_CONTENTS)

(11) LIMIT_WOULD_BE_EXCEEDED (MAX_MESSAGE_SIZE)

(12) MESSAGE_QUEUE_HAS_BEEN_DELETED (*queue*)

(13) MESSAGE_QUEUE_HAS_BEEN_WOKEN (*queue*)

14.2.7 QUEUE_EMPTY

(1)
```
QUEUE_EMPTY (
    queue  : Message_queue_designator
)
```

(2) QUEUE_EMPTY empties the message queue *queue*, i.e. removes all messages from it.

(3) A read lock of the default mode is obtained on *queue*.

Errors

(4) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, WRITE_CONTENTS)

(5) MESSAGE_QUEUE_IS_RESERVED (*queue*)

14.2.8 QUEUE_HANDLER_DISABLE

(1)
```
QUEUE_HANDLER_DISABLE (
    queue  : Message_queue_designator
)
```

(2) QUEUE_HANDLER_DISABLE makes the calling process no longer the listening process for the message queue *queue*. *queue* must be reserved by the calling process.

(3) The "is_listener" link from the calling process to *queue* and its reverse are deleted.

Errors

(4) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, WRITE_LINKS)

(5) MESSAGE_QUEUE_HAS_NO_HANDLER (*queue*)

(6) MESSAGE_QUEUE_IS_NOT_RESERVED (*queue*)

14.2.9 QUEUE_HANDLER_ENABLE

(1)
```
QUEUE_HANDLER_ENABLE (
    queue      : Message_queue_designator,
    types      : Message_types,
    handler    : Handler
)
```

(2) QUEUE_HANDLER_ENABLE makes the calling process the listening process for the message queue *queue*, with associated message types specified by *types*, and handler *handler*.

(3) An "is_listener" link is created from the calling process to *queue*, with "message_types" attribute set to a value representing *types*.

(4) The previous handler, if any, for *queue* is disabled as by a prior call of QUEUE_HANDLER_DISABLE.

Errors

(5) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, WRITE_LINKS)

(6) MESSAGE_QUEUE_IS_NOT_RESERVED (*queue*)

14.2.10 QUEUE_RESERVE

(1)
```
QUEUE_RESERVE (
    queue  : Message_queue_designator
)
```

(2) QUEUE_RESERVE reserves the message queue *queue* for the calling process. If *queue* is already reserved for the calling process, QUEUE_RESERVE has no effect.

(3) A "reserved_message_queue" link reversed by a "reserved_by" link is created from the calling process to *queue*.

Errors

(4) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, APPEND_LINKS)

(5) MESSAGE_QUEUE_IS_RESERVED (*queue*)

14.2.11 QUEUE_RESTORE

(1)
```
QUEUE_RESTORE (
    queue  : Message_queue_designator,
    file   : File_designator
)
```

(2) QUEUE_RESTORE reconstructs the message queue *queue* from the contents of the object designated by *file*.

(3) The last access time of *file* is set to the system time.

(4) A write lock of the default mode is obtained on *queue* and a read lock of the default mode is obtained on *file*.

Errors

(5) ACCESS_ERRORS (*file*, ATOMIC, READ, READ_CONTENTS)

(6) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, WRITE_CONTENTS)

(7) CONTENTS_FORMAT_IS_INVALID (*file*)

(8) LIMIT_WOULD_BE_EXCEEDED (MAX_MESSAGE_QUEUE_SPACE)

(9) MESSAGE_QUEUE_IS_BUSY (*queue*)

(10) MESSAGE_QUEUE_IS_RESERVED (*queue*)

(11) MESSAGE_QUEUE_WOULD_BE_TOO_BIG (*queue*)

14.2.12 QUEUE_SAVE

(1)
```
QUEUE_SAVE (
    queue  : Message_queue_designator,
    file   : File_designator
)
```

(2) QUEUE_SAVE copies all the messages from the message queue *queue* to the contents of the file *file*. The existing contents of *file* is overwritten. The format of the contents of *file* is implementation-defined.

(3) The message queue *queue* is unaffected, except that any "notifier" links from *queue* are deleted.

(4) The last change time and last modification time of *file* are set to the system time of the call.

(5) A write lock of the default mode is obtained on *file* and a read lock on *queue*.

Errors

(6) ACCESS_ERRORS (*file*, ATOMIC, MODIFY, WRITE_CONTENTS)

(7) ACCESS_ERRORS (*queue*, ATOMIC, READ, READ_CONTENTS)

(8) MESSAGE_QUEUE_IS_RESERVED (*queue*)

14.2.13 QUEUE_SET_TOTAL_SPACE

(1)
```
QUEUE_SET_TOTAL_SPACE (
    queue        : Message_queue_designator,
    total_space  : Natural
)
```

(2) QUEUE_SET_TOTAL_SPACE sets the total space of the message queue *queue* to the value of *total_space*.

(3) A write lock of the default mode is obtained on *queue*.

Errors

(4) ACCESS_ERRORS (*queue,* ATOMIC, CHANGE, CONTROL_OBJECT)

(5) LIMIT_WOULD_BE_EXCEEDED (MAX_MESSAGE_QUEUE_SPACE)

(6) MESSAGE_QUEUE_IS_RESERVED (*queue*)

(7) MESSAGE_QUEUE_TOTAL_SPACE_WOULD_BE_TOO_SMALL (*queue, total_space*)

14.2.14 QUEUE_UNRESERVE

(1)
```
QUEUE_UNRESERVE (
    queue  : Message_queue_designator
)
```

(2) QUEUE_UNRESERVE unreserves the message queue *queue* for the calling process. If *queue* is not reserved for the calling process, QUEUE_UNRESERVE has no effect.

(3) The "reserved_message_queue" and "reserved_by" links between the calling process and *queue* are deleted. If the calling process has an "is_listener" link to *queue* then that link and its reverse "listened_to" link are deleted.

Errors

(4) ACCESS_ERRORS (*queue,* ATOMIC, MODIFY, WRITE_LINKS)

(5) NOTE - The termination of a process implies the unreserving of all the process's reserved message queues.

15 Notification

15.1 Notification concepts

15.1.1 Access events and notifiers

(1) Access_event = MODIFICATION_EVENT | CHANGE_EVENT | DELETE_EVENT | MOVE_EVENT

(2) Access_events = **set of** Access_event

(3) A *notifier* is a "notifier" link from a message queue to an object with key attribute "notifier_key" and the attributes "modification_event", "change_event", "delete_event", and "move_event", referred to as *monitored access attributes*. See 14.1 for the DDL definition of notifiers.

(4) A *monitored object* is a destination object of a notifier. The values of the monitored access attributes of the notifier define the events on which the monitored object is monitored:

(5) - Modification event is **true**. Modification events: an operation implicitly sets the last modification time of the object.

(6) - Change event is **true**. Change events: an operation implicitly sets the last change time but not the last modification time of the object. If the operation only sets the volume identifier of the object, the CHANGE_EVENT event is not raised.

(7) - Delete event is **true**. Delete events: an operation results in the deletion of the object.

(8) - Move event is **true**. Move events: an operation results in a change to the volume identifier of the object, including archiving the object and restoring it from archive.

(9) The notification mechanism sends notification messages to message queues when a specified access is carried out on a monitored object. The specified access event is said to be *raised* by the operation that accessed the object. The notification mechanism is said to be *triggered* by the raised event.

(10) If there is a notifier from a message queue to any object then that message queue has a reserving process.

NOTES

(11) 1 The monitored access attributes of the notifier define the access events for which the destination of the notifier is to be monitored. Their initial value is **false**. For each attribute, if the value of the attribute is **true**, then the object is being monitored for that event.

(12) 2 Each value of the notifier key identifies a specific notifier in the context of the associated message queue. As implied by the DDL specification, the notifier key is unique in the context of the associated message queue.

(13) 3 In order to carry out notification mechanism operations, a process must reserve the message queue which is to be used as recipient of the notification messages, and in order to be notified, the message queue must remain reserved by the process.

(14) 4 If a process unreserves a message queue then any notifiers from the message queue are deleted.

(15) 5 A process can reserve several different message queues for notification purposes. For each of these message queues, it can create several notifiers (one for each object under monitoring). An object can be monitored using several message queues by one process or by several processes.

(16) 6 There are additional possibilities for the deletion and moving of objects other than by the OBJECT_DELETE and OBJECT_MOVE operations.

15.1.2 Notification messages

(1)
```
Notification_message_type = MODIFICATION_MSG | CHANGE_MSG | DELETE_MSG |
    MOVE_MSG | NOT_ACCESSIBLE_MSG | LOST_MSG
```

(2) A *notification message* is a message (see 14.1) sent by the notification mechanism to one or more message queues each time an object under monitoring is accessed in a way which has been specified to be monitored. The type of such a message specifies the access event that has been carried out on the monitored object or the information that the monitored object is no longer accessible or that modification messages have been lost. The possible values of the type of a notification message are defined as follows:

(3) - MODIFICATION_MSG: Notifies that a modification access event has been raised (except CONTENTS_WRITE or CONTENTS_TRUNCATE).

(4) - CHANGE_MSG: Notifies that a change access event has been raised.

(5) - DELETE_MSG: Notifies that a delete access event has been raised.

(6) - MOVE_MSG: Notifies that a move access event has been raised.

(7) - NOT_ACCESSIBLE_MSG: A message of this type is sent to a message queue each time a monitored object becomes no longer accessible from the workstation on which that message queue resides.

(8) - LOST_MSG: When a message queue is full and there is not sufficient space on the queue to store a notification message, the notification messages to be sent by the notification mechanism are lost. In this case, when the message queue empties sufficiently to give space for a notification message, a message is sent to the message queue by the notification mechanism saying that some messages have been lost.

(9) The data of the message includes in an implementation-defined way the notifier key that associates the message queue and the monitored object.

(10) At most four notification messages are sent to a message queue, one for each type of access carried out on the object during the period it was explicitly locked. The order of these four messages is implementation-defined.

(11) When a monitored object is archived, a message of type MOVE_MSG and a message of type NOT_ACCESSIBLE_MSG are both sent; when a monitored object is restored from archive, a message of type MOVE_MSG is sent.

15.1.3 Time of sending notification messages

(1) The end of an operation and the releasing of a lock define the points in time at which the notification messages are sent to processes as defined in 15.1.4.

(2) At the appropriate point in time, the switched on access events are raised, triggering the notification mechanism which sends the notification messages to the message queues associated by notifiers with the object that has had been modified, changed, deleted, or moved.

(3) A message of message type NOT_ACCESSIBLE_MSG is sent by the notification mechanism when WORKSTATION_REDUCE_CONNECTION or WORKSTATION_DISCONNECT is called or when a network partition is detected, such that the monitored object becomes inaccessible in the specified manner.

15.1.4 Range of concerned message queues

(1) For an operation modifying, changing, deleting, or moving an object, a notification message is sent to all the message queues associated with that object by a notifier when the update becomes available to the process reserving the message queue.

(2) If the message queue security labels are such that writing to the queue by the process accessing the object would give rise to a mandatory security violation, then no notification message is sent.

(3) On transaction rollback, notification messages are sent notifying rollback and no messages are sent to non-enclosed activities.

15.2 Notification operations

15.2.1 NOTIFICATION_MESSAGE_GET_KEY

(1)
```
NOTIFICATION_MESSAGE_GET_KEY(
    message         : Message,
)
    notifier_key    : Natural
```

(2) NOTIFICATION_MESSAGE_GET_KEY returns a notifier key *notifier_key* derived from the data of the notification message *message*.

(3) *notifier_key* is the notifier key of the notifier whose monitored object underwent the access event which triggered the sending of *message*.

Errors

(4) MESSAGE_IS_NOT_A_NOTIFICATION_MESSAGE (*message*)

(5) NOTE - The notifier identified by *notifier_key* may no longer exist.

15.2.2 NOTIFY_CREATE

(1)
```
NOTIFY_CREATE (
    notifier_key    : Natural,
    queue           : Message_queue_designator,
    object          : Object_designator
)
```

(2) NOTIFY_CREATE creates a notifier from the message queue *queue* to the object *object*.

(3) The notifier key of the notifier is set to *notifier_key*. The monitored access attributes of the notifier are all set to **false**.

Errors

(4) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, APPEND_LINKS)

(5) CONFIDENTIALITY_WOULD_BE_VIOLATED (*object*, ATOMIC)

(6) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (*object*, ATOMIC)

(7) MESSAGE_QUEUE_IS_NOT_RESERVED (*queue*)

(8) NOTIFIER_KEY_EXISTS (*notifier_key*)

(9) OBJECT_IS_INACCESSIBLE (*object*, ATOMIC)

(10) OBJECT_IS_ARCHIVED (*object*)

NOTES

(11) 1 The creation of a notifier from a message queue to an object means that a notification message will be sent to the message queue whenever the object is accessed with some specified access events.

(12) 2 Initially, on creation, the monitored access attributes are set to **false**, so no events are specified. The monitored access events may be changed by NOTIFY_SWITCH_EVENTS.

(13) 3 As implied by the DDL specification, the *notifier_key* value must be unique in the context of the message queue and must be greater than or equal to zero; apart from these constraints, it may be freely chosen by the user.

15.2.3 NOTIFY_DELETE

(1)
```
NOTIFY_DELETE (
    notifier_key    : Natural,
    queue           : Message_queue_designator
)
```

(2) NOTIFY_DELETE deletes the notifier with notifier key *notifier_key* from the message queue *queue*.

Errors

(3) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, WRITE_LINKS)

(4) MESSAGE_QUEUE_IS_NOT_RESERVED (*queue*)

(5) NOTIFIER_KEY_DOES_NOT_EXIST (*notifier_key*)

(6) NOTE - The object which was monitored by *notifier* may continue to be monitored by other notifiers into other message queues. Other objects may continue to be monitored by other notifiers associated with *queue*.

15.2.4 NOTIFY_SWITCH_EVENTS

(1)
```
NOTIFY_SWITCH_EVENTS (
    notifier_key   : Natural,
    queue          : Message_queue_designator,
    access_events  : Access_events
)
```

(2) NOTIFY_SWITCH_EVENTS sets each of the monitored access attributes of the notifier with notifier key *notifier_key* from the message queue *queue* to **true** if the corresponding access event is in *access_events*, and to **false** otherwise.

Errors

(3) ACCESS_ERRORS (*queue*, ATOMIC, MODIFY, WRITE_LINKS)

(4) MESSAGE_QUEUE_IS_NOT_RESERVED (*queue*)

(5) NOTIFIER_KEY_DOES_NOT_EXIST (*notifier_key*)

(6) NOTE - Switching on an access event of a notifier (setting the attribute value to true) means that the associated object is then under monitoring for that access event. Switching off an access event (setting the attribute value to false) means that the associated object is no longer under monitoring for that access event.

16 Concurrency and integrity control

16.1 Concurrency and integrity control concepts

16.1.1 Activities

(1) Activity_class = UNPROTECTED | PROTECTED | TRANSACTION

(2) **sds** system:

(3) activity_class: (**read**) **enumeration** (UNPROTECTED, PROTECTED, TRANSACTION) :=
    UNPROTECTED;

(4) activity_status: (**read**) **non_duplicated enumeration** (UNKNOWN, ACTIVE, COMMITTING,
    ABORTING, COMMITTED, ABORTED) := UNKNOWN;

(5)
```
activity: child type of object with
attribute
    activity_class;
    activity_status;
    activity_start_time: (read) time;
    activity_termination_start_time: (read) time;
    activity_termination_end_time: (read) time;
link
    started_by: (navigate) reference link to process reverse started_activity;
    nested_in: (navigate) reference link to activity reverse nested_activity;
    nested_activity: (navigate) implicit link (system_key) to activity reverse nested_in;
    process_started_in: (navigate) implicit link (system_key) to process reverse
        started_in_activity;
end activity;
```

(6) **end** system;

(7) An activity is the framework in which a set of related operations takes place. Each operation is always carried out on behalf of just one activity. An activity is started at the time it is created and remains in existence until the deactivation of the process which started it.

(8) The activity class of an activity describes the degree of protection which the activity requires; it affects the default level of concurrency control applicable to operations carried out on behalf of the activity. There are three activity classes:

(9) - UNPROTECTED. An *unprotected* activity, used when it is not necessary to protect data from concurrent activities.

(10) - PROTECTED. A *protected* activity, used when data to be accessed needs protection from concurrent activities.

(11) - TRANSACTION. A *transaction* activity (or *transaction*), used when the activity has a significant effect on the object base and its integrity needs to be protected.

(12) The activity status records the current state of the activity. The possible states of an activity are:

(13) - UNKNOWN. The "activity" object has been created by an operation defined in clause 9.

(14) - ACTIVE. The activity is started and its termination is not yet initiated.

(15) - COMMITTING. The activity's normal termination is initiated but not completed.

(16) - ABORTING. The activity's abnormal termination is initiated but not completed.

(17) - COMMITTED. The activity is normally terminated.

(18) - ABORTED. The activity is abnormally terminated.

(19) The activity start time records the time when the activity was started.

(20) The activity termination start time records the time when the termination of the activity was started.

(21) The activity termination end time records the time when the termination of the activity was completed.

(22) The "started_by" process is the process that started the activity.

(23) The "nested_in" activity, called the *enclosing activity* of the activity, is the activity within which the activity was started. The *nested activities* of an activity are the activities for which the activity is the enclosing activity.

(24) The "process_started_in" processes are the processes which were created while the activity was the current activity.

(25) Within each process there is only one *current activity*. When a process is initiated, its current activity is the current activity of its parent process. When an activity is started in a process it becomes the current activity of the process; the current activity is then the activity of the process with the highest key in the "started_activity" link from the process and which is still active. When an activity is terminated in a process its immediate enclosing activity becomes the current activity of the process.

(26) Each workstation in a PCTE installation has an outermost activity. The *outermost activity* of a workstation is an unprotected activity that is created by implementation-dependent means such that it is indistinguishable from an activity created by ACTIVITY_START except that it has no "started_by" or "nested_in" link. It has a "process_started_in" link to the initial process.

(27) Updates by an activity to a resource are *available* if, when the updated resource is read by another activity not enclosed by the updating activity, data derived from the updated state of the resource is obtained. Data derived from the updated state of the resource is obtained when read by the updating activity and by nested activities without necessarily being generally available.

NOTES

(28) 1 Activities can be internal to one process or can extend over several descendant processes. A process is free to start an activity, but a process is only allowed to terminate activities that it has started.

(29) 2 Operations performed by a process, other than those on an open contents, are carried out on behalf of the current activity of the process at the time the operation is called.

(30) 3 A nested transaction may be terminated without implying the termination of its enclosing transaction. When a transaction is normally terminated then all the read locks it has acquired are released and all the write locks of default mode it has acquired or inherited from its nested transactions are inherited by its enclosing transaction. When a transaction is abnormally terminated then the changes made by it and all its nested transactions are unmade (unless explicitly excluded from rollback) and all the locks it has acquired, including the write locks it has inherited from its nested transactions, are released. This effect is transitive so that, in the case of successive normal terminations of transactions nested one in another, nested transaction write locks are not released, and the changes not committed, until the outermost transaction is normally terminated.

(31) 4 Protected or unprotected activities may also be nested within transactions. In this case, modifications made within the nested activities are considered also to be changes made within their closest enclosing transaction. Accordingly, when locks are acquired by nested protected or unprotected activities, locks are implicitly acquired at the same time by their closest enclosing transaction (see 16.1.6)

(32) 5 In the same way when a lock whose mode is not the default write mode is acquired by a nested transaction, a lock is implicitly acquired at the same time by the closest enclosing transaction.

(33) 6 A process running on behalf of a transaction can explicitly exclude from rollback changes made to certain resources by explicitly locking such resources in unprotected or protected modes (i.e. not in default write modes) (see 16.1.5). However creating or deleting of objects and links cannot be excluded from rollback.

(34) 7 The outermost activity of a workstation is implicitly set up by the system. It is intended to provide a valid activity framework for the initial process of the workstation. Each workstation has its own outermost activity, i.e. the outermost activity of a workstation cannot be the outermost activity of another workstation. An initial process is initiated in the context of that activity. It is intended that the initial process should then start an activity suitable for its own requirements.

(35) 8 Transactions do not protect the local data of a process, hence, for example, changes to contents handles, object references, and other local variables made within the scope of a transaction are not reversed if the transaction is aborted.

16.1.2 Resources and locks

(1) Lock_internal_mode = READ_UNPROTECTED | READ_SEMIPROTECTED |
WRITE_UNPROTECTED | WRITE_SEMIPROTECTED | DELETE_UNPROTECTED |
DELETE_SEMIPROTECTED | READ_PROTECTED | DELETE_PROTECTED |
WRITE_PROTECTED

(2) Lock_set_mode = Lock_internal_mode | WRITE_TRANSACTIONED |
DELETE_TRANSACTIONED | READ_DEFAULT | WRITE_DEFAULT | DELETE_DEFAULT

(3) **sds** system:

(4) lock_mode: READ_UNPROTECTED, READ_SEMIPROTECTED, WRITE_UNPROTECTED,
WRITE_SEMIPROTECTED, DELETE_UNPROTECTED, DELETE_SEMIPROTECTED,
READ_PROTECTED, DELETE_PROTECTED, WRITE_PROTECTED,
WRITE_TRANSACTIONED, DELETE_TRANSACTIONED;

(5) lock_external_mode: (**read**) **enumeration** (lock_mode) := READ_UNPROTECTED;

(6) lock_internal_mode: (**read**) **enumeration** (lock_external_mode **range**
READ_UNPROTECTED .. WRITE_PROTECTED) := READ_UNPROTECTED;

(7)
```
extend object type activity with
link
    lock: (navigate) non_duplicated designation link (lock_identifier) to object with    attribute
        locked_link_name;
        lock_external_mode;
        lock_internal_mode;
        lock_explicitness: (read) enumeration (EXPLICIT, IMPLICIT) := IMPLICIT;
        lock_duration: (read) enumeration (SHORT, LONG) := SHORT;
    end lock;
end activity;
```

(8)
```
extend link type process_waiting_for with
attribute
    lock_external_mode;
    lock_internal_mode;
end process_waiting_for;
```

(9)
```
end system;
```

(10) A *resource* is either an object resource or a link resource.

(11) An *object resource* is an object restricted to the following:

(12) - its contents,

(13) - its type,

(14) - its preferred link type and preferred link key,

(15) - its attributes, except the predefined attributes "last_access_time", "last_change_time", "last_modification_time", "last_composite_access_time", "last_composite_change_time", "last_composite_modification_time", "num_incoming_links", "num_incoming_ composition_links", "num_incoming_existence_links", "num_incoming_reference_links", "num_incoming_stabilizing_links", "num_outgoing_composition_links", and "num_outgoing_existence_links",

(16) - its incoming "object_on_volume" link.

(17) A *link resource* is a link, identified by its link name and restricted to the following:

(18) - its link type,

(19) - its sequence of key attributes,

(20) - its set of non-key attributes,

(21) - the object designator of its destination.

(22) The fact that a resource is locked is represented by a "locked_by" link from the object resource or the origin of a link resource to the activity which holds the lock. The locked link name of the link specifies whether the resource is an object resource or a link resource:

(23) - if the locked resource is a link resource, the "locked_link_name" attribute is set to the link name in canonical form (see 23.1.2.4).

(24) - if the locked resource is an object resource, the "locked_link_name" attribute is set to the empty string.

(25) A *lock* is represented by a "lock" link from an activity to a resource; the activity is said to *hold* the lock *on* the resource. The link is created at the time the lock is established and remains until the lock is released or inherited. Locks ensure the consistency of object base data access operations by controlling the synchronization of concurrent operations on the same resources.

(26) A lock is characterized by a unique lock identifier, the value of which is implementation-dependent.

(27) The *concerned domain* of a resource is the set of resources which can be affected by modifications of that resource:

(28) - if the resource is an object, the concerned domain is the object resource and the set of links (link resources) originating from the object.

(29) - if the resource is a link, the concerned domain is the link resource and the object (object resource) from which the link starts.

(30) A resource is said to be *operated on* by an activity when:

(31) - either the resource is an object whose contents are currently open (see clause 12), by CONTENTS_OPEN or PROCESS_START on behalf of that activity, in which case the resource is operated on while the contents is open;

(32) - or the resource (i.e. object or link) is the subject of operations other than operations on "lock" and "locked_by" links and on the contents of objects, in which case the resource is operated on for the duration of the operation.

(33) An activity can lock a resource just once; i.e. two locks originating from the same activity cannot have the same locked resource and the same destination.

(34) A lock has the following attributes:

(35) - A lock external mode, which controls synchronization of resource accesses between an activity and all other activities which are not nested (either directly or transitively) to it.

(36) - A lock internal mode, which controls synchronization of resource accesses between an activity and all activities which are nested (either directly or transitively) to it.

(37) The lock internal mode is equal to or weaker than the lock external mode (see below). See below for a definition of lock modes.

(38) - A lock explicitness, which records how the lock was established:

(39) . EXPLICIT. An *explicit* lock, i.e. it was established explicitly by one of locking operations.

(40) . IMPLICIT. An *implicit* lock, i.e. it was established implicitly as the resource was implicitly acquired.

(41) - A lock duration, which records the duration of the lock:

(42) . LONG. A *long* lock, i.e. one which, once established, holds until the termination of the activity.

(43) . SHORT. A *short* lock, i.e. one which can be released before the termination of the activity.

(44) A long lock can be held only by a transaction.

(45) A short lock can be held only by a protected or an unprotected activity.

(46) NOTE – The incoming "object_on_volume" links of an object resource are created or deleted by operations which create, move, or delete objects. When these operations are performed in a transaction which is then rolled back, the creation or deletion of these links is also rolled back.

16.1.3 Lock modes

(1) The meanings of the lock mode values are as follows. The abbreviations shown are used in the tables at the end of this clause.

(2) - READ_UNPROTECTED (RUN). The activity holding the lock can read the resource. Other activities can concurrently read or write to the same resource or delete it.

(3) - READ_SEMIPROTECTED (RSP) (for object resources only). The activity holding the lock can read the resource. Other activities can concurrently read from the same resource. Other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED, WRITE_SEMIPROTECTED, WRITE_PROTECTED or WRITE_TRANSACTIONED locks but cannot delete it.

(4) - WRITE_UNPROTECTED (WUN). The activity holding the lock can read or write to the resource.

(5) If the resource is an object, other activities can concurrently read or write to the same resource or delete it with DELETE_UNPROTECTED or DELETE_SEMIPROTECTED locks, other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED or WRITE_SEMIPROTECTED locks and other activities can concurrently read the resource with READ_UNPROTECTED or READ_SEMIPROTECTED locks.

(6) If the resource is a link, other activities can concurrently read or write to the same resource or delete it with WRITE_UNPROTECTED locks and other activities can concurrently read the resource with READ_UNPROTECTED locks.

(7) Updates to the resource are available if an enclosing activity does not hold a WTR or DTR lock on the resource.

(8) - WRITE_SEMIPROTECTED (WSP) (for object resources only). The activity holding the lock can read or write to the resource. Other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED or WRITE_SEMIPROTECTED locks but cannot delete it and other activities can concurrently read the resource with READ_UNPROTECTED or READ_SEMIPROTECTED locks. Updates to the resource are available if an enclosing activity does not hold a WTR or DTR lock on the resource.

(9) - DELETE_UNPROTECTED (DUN) (for object resources only). The activity holding the lock can read or write to the resource or delete it. Other activities can concurrently read or write to the same resource or delete it with DELETE_UNPROTECTED locks, other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED and other activities can concurrently read the resource with READ_UNPROTECTED locks. Updates to the resource are available if an enclosing activity does not hold a WTR or DTR lock on the resource.

(10) - DELETE_SEMIPROTECTED (DSP) (for object resources only). The activity holding the lock can read or write to the resource or delete it. Other activities can concurrently read or write to the same resource with WRITE_UNPROTECTED locks but cannot delete it and other activities can concurrently read the resource with READ_UNPROTECTED locks. Updates to the resource are available if an enclosing activity does not hold a WTR or DTR lock on the resource.

(11) - READ_PROTECTED (RPR). The activity holding the lock can read the resource. Other activities can concurrently read the same resource with READ_UNPROTECTED,

READ_SEMIPROTECTED or READ_PROTECTED locks. No other activities can concurrently write to the same resource.

(12) - WRITE_PROTECTED (WPR). The activity holding the lock can read or write to the resource. Other activities can concurrently read the same resource with READ_UNPROTECTED or READ_SEMIPROTECTED locks. No other activities can concurrently write to the same resource. Updates to the resource are available if an enclosing activity does not hold a WTR or DTR lock on the resource.

(13) - DELETE_PROTECTED (DPR) (for object resources only). The activity holding the lock can read or write to the resource or delete it. Other activities can concurrently read the same resource with READ_UNPROTECTED locks. No other activities can concurrently write to the same resource. Updates to the resource are available if an enclosing activity does not hold a WTR or DTR lock on the resource.

(14) - WRITE_TRANSACTIONED (WTR). Transaction holding the lock can read or write to the resource. Other activities can concurrently read the same resource with READ_UNPROTECTED or READ_SEMIPROTECTED locks. No other activities can concurrently write to the same resource.

(15) - DELETE_TRANSACTIONED (DTR) (for object resources only). Transaction holding the lock can read or write to the resource or delete it. Other activities can concurrently read the same resource with READ_UNPROTECTED locks. No other activities can concurrently write to the same resource.

(16) It is implementation-defined whether or not updates to a resource are available if the updates are performed while an activity holds a WTR or DTR lock on the resource.

(17) Locks of the following modes, whether internal or external, can be held only on an object resource: READ_SEMIPROTECTED, WRITE_SEMIPROTECTED, DELETE_SEMIPROTECTED, DELETE_PROTECTED, DELETE_TRANSACTIONED.

(18) The modes of a lock on a given resource must be compatible with the modes of locks held by other activities on resources in the concerned domain of that resource. Its external mode must be compatible with the external mode of all locks held on the resources in the concerned domain by other activities which are not enclosing, nor nested to the issuing activity, and with the internal mode of all locks already established by the enclosing activities on the resources in the concerned domain. Its internal mode must be compatible with the external modes of all locks already established by the nested activities on the resources in the concerned domain

(19) The lock modes are grouped into two categories:

(20) - *Read lock modes*: READ_UNPROTECTED, READ_SEMIPROTECTED, READ_PROTECTED

(21) - *Write lock modes*: WRITE_PROTECTED, WRITE_TRANSACTIONED, WRITE_UNPROTECTED, WRITE_SEMIPROTECTED, DELETE_PROTECTED, DELETE_UNPROTECTED, DELETE_SEMIPROTECTED, DELETE_TRANSACTIONED

(22) There are three relations defined between lock modes: *relative strength*, *relative weakness*, and *compatibility*. The relative strength relation between lock modes is defined by table 4. The relative weakness relation is the inverse of the relative strength relation (i.e. L1 is weaker than L2 if and only if L2 is stronger than L1). The compatibility relation is defined by table 5.

(23) Updates to an accounting log or an audit file, the "message_count", "last_send_time" and "last_receive_time" attributes of a message queue, and the "last_access_time" attribute of an object, are never made on behalf of the current activity.

(24) If the current activity is a transaction, it may be terminated in one of two ways:

(25) - For updates performed on behalf of the current activity while transaction locks were established, the operation ACTIVITY_END, which *commits* the transaction, results in those updates becoming permanent, providing the transaction locks are not inherited (see 16.1.4).

(26) - The operation ACTIVITY_ABORT, which *aborts* the transaction, causes the updates performed on behalf of the current activity while transaction locks were established to be undone, apart from updates applied to contents of "pipe", "message_queue", and "device" objects.

16.1.4 Inheritance of locks

(1) Inheritance of locks occurs only between transactions nested one in the other. A transaction inherits write locks of default modes (i.e. locks of modes WTR or DTR) from its (immediate) nested transactions each time such a nested transaction terminates normally (i.e. when it commits).

(2) When a transaction T1 terminates, the lock it holds on a resource X is inherited by the nearest enclosing transaction T of T1. If T already holds a lock on X, the lock is promoted according to the rules of implicit promotion (see 16.1.5).

(3) NOTE - Updates to a resource are committed or cancelled when there cease to be any WTR or DTR locks on that resource. When an enclosing transaction inherits WTR and DTR locks, it also inherits the updates. Normally, when there is no further enclosing transaction, updates are committed when the current transaction activity ends. If however the enclosing transaction T1 holds a non-transaction lock on a resource updated under enclosed transaction T, then when T ends the transaction lock is not inherited and neither are the updates which are committed. An exception to this is when a transaction T2 enclosing T1 exists and has a WTR or DTR lock on the resource; in this case the updates are inherited by T2 when T ends and are not committed at that point.

16.1.5 Establishment and promotion of locks

(1) A lock is *requested* on a resource on behalf of an activity if an attempt is made to create a lock on that resource on behalf of that activity.

(2) A lock is *established* on a resource when a lock is requested on the resource on behalf of an activity and no lock has yet been acquired by the activity.

(3) A lock is *explicitly established* by means of operation LOCK_SET_OBJECT. A lock is *implicitly established* if the resource is implicitly acquired by some operation (other than LOCK_SET_OBJECT) operating on the resource and carried out on behalf of the activity. Locks of mode RSP, WSP, and DSP can only be established explicitly.

(4) The modes of a lock, once established, can evolve either implicitly, according to the way the resource is operated on, or explicitly by means of the lock set operations.

(5) The following enumerates, for each activity class, the implicit lock modes which are requested depending on the access performed on the acquired resource. Locks on link resources are always implicit and therefore always adopt default modes.

(6) - Default external modes:

(7) . for unprotected activities the external mode is RUN when reading a resource, WUN when creating or updating a resource or deleting a link resource, and DUN when deleting an object resource;

(8) . for protected activities the external mode is RPR when reading a resource, WPR when creating or updating a resource or deleting a link resource, and DPR when deleting an object resource;

(9) . for transaction activities the external mode is RPR when reading a resource, WTR when updating a resource or creating or deleting a link resource, and DTR when creating or deleting an object resource.

(10) - Default internal modes: for every activity class, internal modes are WUN for resources being created or updated and link resources being deleted, DUN for object resources being deleted, and RUN in all other cases.

(11) The lock mode actually acquired depends on whether the lock is established or promoted. If it is established then the lock mode acquired is the default lock mode. If a promotion occurs, see below. Tables 8 and 9 summarize the default external modes.

(12) When a lock is requested on a resource on behalf of an activity and a lock has already been acquired by the activity, then the lock may be promoted. To *promote* a mode of a lock is to transform it to a stronger mode which is compatible (as for the establishment of a new lock) with other locks on resources in the concerned domain. See table 6 and 16.1.7.

(13) *Implicit promotion* of either or both the internal and the external modes of a lock occurs when an operation performing a write access (e.g. OBJECT_SET_ATTRIBUTE or CONTENTS_OPEN with an opening mode allowing write access) is applied to a resource already acquired by the activity with a lock whose modes allow only read access to that resource, or when an operation deleting an object (e.g. LINK_DELETE or OBJECT_DELETE) is applied to an object resource already acquired by the activity with a lock whose modes do not allow deletion of that object resource.

(14) *Explicit promotion* of either the internal or the external lock mode occurs when the lock set operations are applied to a resource already locked (either explicitly or implicitly) by the activity on behalf of which the lock set operation is carried out. The new mode must obey the promotion rules for lock modes (see below).

(15) Any explicit attempt to promote an explicit or an implicit external or internal mode M1, or implicit attempt to promote an implicit external or internal mode M1, to a mode M2 which has no relation of relative strength with M1 is converted into an attempt to promote M1 to the weakest mode which is stronger than both M1 and M2 (e.g. an attempt to promote a RPR mode to a WUN mode is implicitly converted into an attempt to promote the mode to WPR). Table 6 defines the implicit promotion of lock modes when the prior lock is explicit; table 7 defines the promotion of lock modes for the other cases.

(16) In the operation definitions, the phrase 'a *mode* lock of the default mode is *obtained* on *object*' where *object* is an object and *mode* is 'read' or 'write', is used to mean that an attempt is made to establish an implicit lock on the object resource *object* of a mode given by table 8, depending on the class of the current activity and the default lock mode of 'read' or 'write', and if 'write', whether *object* is being created, updated or deleted.

(17) Similarly, the phrase 'a *mode* lock of the default mode is obtained on *link*' where *link* is a link and *mode* is 'read' or 'write', is used to mean that an attempt is made to establish an implicit lock

on the link resource *link* of a mode given by table 9, depending on the class of the current activity and the default lock mode of 'read' or 'write', and if 'write', whether *link* is being created, updated or deleted. If *link* is a 'lock' or 'locked_by' link then no lock is established on it.

(18) If no lock currently exists on the resource (*object* or *link*) for that activity, an attempt is made to establish the lock, otherwise implicit promotion is attempted.

16.1.6 Implied locks

(1) Locks can be established or promoted on a resource as a result of establishing or promoting another lock.

(2) When locks are acquired by nested activities, this implies that implicit locks are acquired at the same time by their closest enclosing transaction:

(3) - The establishing of (or promotion to) a WUN, WSP, WPR external mode lock for an activity also implies an implicit establishment (or implicit promotion) of a lock of external mode WTR on the resource on behalf of the closest enclosing transaction.

(4) - The establishing of (or promotion to) a DUN, DSP, DPR external mode lock for an activity also implies an implicit establishment (or implicit promotion) of a lock of external mode DTR on the resource on behalf of the closest enclosing transaction.

(5) - The establishing of (or promotion to) a RUN, RSP, RPR, WTR, or DTR external mode lock for an activity also implies an implicit establishment (or implicit promotion) of a lock of external mode RPR on the resource on behalf of the closest enclosing transaction.

(6) Establishing or promoting a lock on a link also implies the implicit establishment (or promotion) of a read lock of the default mode on the origin of that link for the current activity, if the link type of the link has an upper bound or a non-zero lower bound and the link is being deleted or created.

(7) The establishing of (or promotion to) a write lock on the last composition or existence link leading to an object for the purpose of its deletion also implies the implicit establishment (or implicit promotion) of a write lock allowing deletion on this object for the current activity (i.e. a DUN, DSP, DPR, or DTR lock according to the class of the activity and the promotion rules). The establishing of (or promotion to) a write lock on a composition or existence link for the purpose of deletion of the link results in the implicit establishment (or implicit promotion) of a read lock on the destination of the link.

(8) In all these cases the internal lock mode of the implied lock is RUN.

16.1.7 Conditions for establishment or promotion of a lock

(1) The following conditions must be satisfied to establish or promote a lock on a given resource:

(2) - Access rights: the current activity of a process can explicitly or implicitly establish a lock on a resource if and only if the process has at least one discretionary access right to the resource if it is an object resource, or to its origin if it is a link resource.

(3) - Lock mode compatibility: an activity can establish (or promote) a lock on a resource if

(4) . its external mode is compatible with the external mode of all locks held on the resources in the concerned domain by other activities which are not enclosing, nor nested to the issuing activity;

(5) . its external mode is compatible with the internal mode of all locks already established by the enclosing activities on the resources in the concerned domain;

(6) . its internal mode is compatible with the external modes of all locks already established by the nested activities on the resources in the concerned domain;

(7) . The implied lock (if any) must also be compatible with existing locks as defined above.

(8) If the conditions do not hold, either the issuing operation waits, waiting for the resource to become available, or the request returns an error without delay.

(9) When an operation waits as a result of attempting to lock a resource in a mode which is incompatible with the existing locks on that resource held by other discrete activities, the operation is said to be *waiting on the resource*. When an operation is waiting on a resource, a "process_waiting_for" link is created from the process of the waiting operation to the resource. The "waiting_type" attribute of that link is set to WAITING_FOR_LOCK, the required external and internal modes of the lock set in the "lock_external_mode" and "lock_internal_mode" attributes respectively of the link, and the "locked_link_name" attribute is set to the link name, in canonical form, of the resource on which the lock is to be established if the resource is a link, and to the empty string otherwise. The link is removed when the operation which is waiting on the resource is interrupted, or the resource is acquired.

(10) If a lock is held on an object resource by an activity, then any attempt to establish a lock on any of its links by that activity has no effect unless the lock mode resulting from the request is stronger than or has no relation of relative strength to the external mode of the lock on the object.

(11) If a lock is held on a link resource by an activity, then when a lock is established on its origin by that activity the lock on the link is discarded, unless the external lock mode on the link is stronger than or has no relation of relative strength to the external mode of the lock on the origin.

16.1.8 Releasing locks

(1) A distinction is made between *discarding* a lock (to get rid of it) and *releasing* a lock. Releasing a lock implies discarding the lock for the current activity, and if the lock has a WTR or a DTR mode then the closest enclosing transaction inherits the modifications to the resource. If there is no such transaction then modifications are *committed* (i.e. modifications can no longer be discarded).

(2) In any case, this results in the deletion of the "lock" link and of the "locked_by" link associated with the released lock. In the case that the lock is inherited by the closest enclosing transaction, if the resource was not already locked on behalf of that transaction, new "lock" and "locked_by" links are created between this activity and the locked resource, in order to represent the inherited lock.

(3) Long locks are released at the end of the activity. In the case of short locks two cases can apply:

(4) - The lock was explicitly established: it is released either at the end of the activity or at the explicit unlock of the resource, whichever occurs first.

(5) - The lock was implicitly established: it is released as soon as the locked resource is no longer being operated on behalf of the activity holding the lock (for example when the last open contents handle to the object contents is closed by CONTENTS_CLOSE).

(6) When a lock on a resource is discarded, if one or more operations are waiting on the resource then an attempt is made to establish or promote a lock on that resource in the modes given by the attributes "lock_external_mode" and "lock_internal_mode" of a "process_waiting_for" link to that resource on behalf of the current activity of the process which is the origin of that link. If a lock can be established or promoted on behalf of one of those activities, then the corresponding "process_waiting_for" link is deleted. If a lock can be established or promoted on behalf of more than one such activity, it is not defined on behalf of which activity it is established or promoted. The resource on which the lock is established or promoted is the destination of the "process_waiting_for" link if the "locked_link_name" attribute of that link is empty, otherwise it is the link with that attribute as link name.

NOTES

(7) 1 The description of each of the operations defines the resources, if any, which are operated on by the operation.

(8) 2 Nested parallel activities should be achieved by using parallel processes.

(9) 3 The internal mode of a lock held by an activity affects only the activity and its nested activities.

16.1.9 Permanence of updates

(1) When an update, whether made while a lock is established or not, is made *permanent*, the resulting change to objects in the object base is such that if a volume failure, device failure, or network failure event occurs so as to render one or more of those objects inaccessible, the objects retain their updated state. Conversely, if an update is not made permanent and such a failure event occurs then the objects revert to a state which existed before the update. An update to a link is considered to be an update to its origin.

(2) If an update is made to an object or link while only non-transaction locks are established on that object or link then the update is made permanent at the latest when the activity in which the update occurred is terminated.

(3) Updates made to objects or links, while WTR or DTR locks are established on those objects or links, are made permanent atomically when no transaction locks remain after a transaction end.

(4) Updates made which do not require a lock to be established on the object or link, for example some operations defined in clause 13 and updates to audit files and accounting logs, are made permanent at an implementation-defined time.

16.1.10 Tables for locks

Table 4 - Relative strength of lock modes

	mode2										
mode1	**RUN**	**RSP**	**WUN**	**WSP**	**RPR**	**WPR**	**WTR**	**DUN**	**DSP**	**DPR**	**DTR**
RUN	=	<	<	<	<	<	<	<	<	<	<
RSP	>	=	-	<	<	<	<	-	<	<	<
WUN	>	-	=	<	-	<	<	<	<	<	<
WSP	>	>	>	=	-	<	<	-	<	<	<
RPR	>	>	-	-	=	<	<	-	-	<	<
WPR	>	>	>	>	>	=	<	-	-	<	<
WTR	>	>	>	>	>	>	=	-	-	-	<
DUN	>	-	>	-	-	-	-	=	<	<	<
DSP	>	>	>	>	-	-	-	>	=	<	<
DPR	>	>	>	>	>	>	-	>	>	=	<
DTR	>	>	>	>	>	>	>	>	>	>	=

Key

<	*mode1* is weaker than *mode2*
>	*mode1* is stronger than *mode2*
=	*mode1* and *mode2* are the same
-	there is no relation of relative strength between *mode1* and *mode2*

Table 5 - Compatibility of lock modes

	mode2										
mode1	**RUN**	**RSP**	**WUN**	**WSP**	**RPR**	**WPR**	**WTR**	**DUN**	**DSP**	**DPR**	**DTR**
RUN	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
RSP	yes	yes	yes	yes	yes	yes	yes	no	no	no	no
WUN	yes	yes	yes	yes	no	no	no	yes	yes	no	no
WSP	yes	yes	yes	yes	no	no	no	no	no	no	no
RPR	yes	yes	no	no	yes	no	no	no	no	no	no
WPR	yes	yes	no	no	no	no	no	no	no	no	no
WTR	yes	yes	no	no	no	no	no	no	no	no	no
DUN	yes	no	yes	no	no	no	no	no	no	no	no
DSP	yes	no	yes	no	no	no	no	no	no	no	no
DPR	yes	no	no	no	no	no	no	no	no	no	no
DTR	yes	no	no	no	no	no	no	no	no	no	no

Key

yes	mode1 and mode2 are compatible
no	mode1 and mode2 are not compatible

Table 6 - Implicit promotion of explicit lock of mode *mode1* to *mode2*

	mode2							
mode1	**RUN**	**WUN**	**RPR**	**WPR**	**WTR**	**DUN**	**DPR**	**DTR**
RUN	no	WUN	no	WUN	WUN	DUN	DUN	DTR
RSP	no	WSP	no	WSP	WSP	DSP	DSP	DTR
WUN	no	no	no	no	no	DUN	DUN	DTR
WSP	no	no	no	no	no	DSP	DSP	DTR
RPR	no	WPR	no	WPR	WPR	DPR	DPR	DTR
WPR	no	no	no	no	no	DPR	DPR	DPT
WTR	-	-	-	-	no	-	-	DTR
DUN	no	no	no	no	no	no	no	DTR
DSP	no	no	no	no	no	no	no	DTR
DPR	no	no	no	no	no	no	no	DTR
DTR	-	-	-	-	no	-	-	DTR
Key								
no	there is no promotion							
-	the case does not apply							

Table 7 - Promotion of *mode1* to *mode2*: other cases

	mode2										
mode1	**RUN**	**RSP**	**WUN**	**WSP**	**RPR**	**WPR**	**WTR**	**DUN**	**DSP**	**DPR**	**DTR**
RUN	no	RSP	WUN	WSP	RPR	WPR	WTR	DUN	DSP	DPR	DTR
RSP	no	no	WSP	WSP	RPR	WPR	WTR	DSP	DSP	DPR	DTR
WUN	no	WSP	no	WSP	WPR	WPR	WTR	DUN	DSP	DPR	DTR
WSP	no	no	no	no	WPR	WPR	WTR	DSP	DSP	DPR	DTR
RPR	no	no	WPR	WPR	no	WPR	WTR	DPR	DPR	DPR	DTR
WPR	no	no	no	no	no	no	WTR	DPR	DPR	DPR	DTR
WTR	no	no	no	no	no	no	no	DTR	DTR	DTR	DTR
DUN	no	DSP	no	DSP	DPR	DPR	DTR	no	DSP	DPR	DTR
DSP	no	no	no	no	DPR	DPR	DTR	no	no	DPR	DTR
DPR	no	no	no	no	no	no	DTR	no	no	no	DTR
DTR	no	no	no	no	no	no	no	no	no	no	no

Table 8 - Default External Lock Modes for Object Resources

	Default lock mode			
Activity class	**Read**	**Write**		
		Update	**Object creation**	**Object deletion**
UNPROTECTED	RUN	WUN	WUN	DUN
PROTECTED	RPR	WPR	WPR	DPR
TRANSACTION	RPR	WTR	DTR	DTR

Table 9 - Default External Lock Modes for Link Resources

	Default lock mode			
Activity class	**Read**	**Write**		
		Update	**Link creation**	**Link deletion**
UNPROTECTED	RUN	WUN	WUN	WUN
PROTECTED	RPR	WPR	WPR	WPR
TRANSACTION	RPR	WTR	WTR	WTR

16.2 Concurrency and integrity control operations

16.2.1 ACTIVITY_ABORT

(1)
```
ACTIVITY_ABORT (
)
```

(2) ACTIVITY_ABORT terminates the current activity of the calling process and discards uncommitted updates. The following actions are performed in order:

(3) - The activity status and activity start termination time of the current activity of the calling process are set to ABORTING and the system current time respectively.

(4) This implies abnormal termination of any process P which was initiated in the context of the current activity and the execution of which has been started but not yet terminated (i.e a process in one of the states RUNNING, SUSPENDED, and STOPPED), in the same way as by calling PROCESS_TERMINATE (P, ACTIVITY_ABORTED).

(5) ACTIVITY_ABORT waits until all those processes have terminated (i.e. have the state TERMINATED). If while in this phase ACTIVITY_ABORT is interrupted, either because the time-out period for the calling process has expired, or because another process has called PROCESS_INTERRUPT_OPERATION for the calling process, then the current activity and the associated resources are not affected. In particular, the activity status and activity start termination time of the activity are reset to their previous values.

(6) - If the current activity is a transaction, all updates made on behalf of the activity to the resources acquired with WRITE_TRANSACTIONED or DELETE_TRANSACTIONED external mode lock since the establishing of the locks or the promotion of their external mode to WRITE_TRANSACTIONED (or DELETE_TRANSACTIONED for the locks which were directly established or promoted to this mode) are discarded.

(7) - All the locks held by the activity are discarded. This results in the deletion of the "lock" and "locked_by" links associated with those locks.

(8) - The activity status and activity start termination time of the activity are set to ABORTED and the current system time respectively. The activity remains in existence until the calling process is deleted.

(9) As a result of these actions, the activity on whose behalf the aborted activity was initiated becomes the calling process's current activity.

Errors

(10) ACTIVITY_WAS_NOT_STARTED_BY_CALLING_PROCESS

(11) ACTIVITY_IS_OPERATING_ON_A_RESOURCE

16.2.2 ACTIVITY_END

(1)
```
ACTIVITY_END (
)
```

(2) ACTIVITY_END terminates the current activity of the calling process normally. The effect of this operation is immediately to commit all outstanding updates in the context of the enclosing activities and to release all locks still held by the activity. The following actions are performed in order:

(3) - The activity status and activity start termination time of the current activity of the calling process are set to COMMITTING and the current system time respectively.

(4) The operation then waits until all the processes which were initiated on behalf of the activity and the execution of which has been started but not yet terminated (i.e. processes which are running, suspended, or stopped) have terminated.

(5) If, while in this phase, the operation is interrupted, either because the time-out period defined for the calling process has expired or because another process has called PROCESS_INTERRUPT_OPERATION for the calling process, then the current activity and the associated resources are not affected. In particular, the activity status and activity start termination time of the activity are reset to their previous values.

(6) - The locks still held by the activity are released. For locks established with external mode WRITE_TRANSACTIONED or DELETE_TRANSACTIONED, all the updates made to the locked resource on behalf of the activity since the establishment of the lock are committed in the context of the enclosing transactions. This means the WRITE_TRANSACTIONED and DELETE_TRANSACTIONED locks are inherited by the closest enclosing transaction if any, otherwise the modification are committed (i.e. modification can no longer be discarded) and those locks are discarded (see 16.1.4 and 16.1.8). In any case, this results in the deletion of the "lock" and "locked_by" links associated with those locks.

(7) - The activity status and activity start termination time of the activity are set to COMMITTED and the current system time respectively.

(8) The activity object remains in existence until the calling process is deleted. The activity on whose behalf the terminated activity was initiated becomes the calling process's current activity.

Errors

(9) ACTIVITY_WAS_NOT_STARTED_BY_CALLING_PROCESS

(10) ACTIVITY_IS_OPERATING_ON_A_RESOURCE

(11) TRANSACTION_CANNOT_BE_COMMITTED

16.2.3 ACTIVITY_START

(1)
```
ACTIVITY_START (
    activity_class   : Activity_class
)
```

(2) ACTIVITY_START creates a new activity of activity class *activity_class***,** nested within the current activity of the calling process.

(3) The activity is created on the same volume as the calling process, with a "started_activity" link to it from the calling process. The activity has the same mandatory labels and the same atomic and composite ACLs as the calling process.

(4) A "nested_in" link and a "nested_activity" link are created between the new activity and the current activity of the calling process.

(5) The activity class and activity start time of the new activity are set to *activity_class* and the current system time respectively.

(6) The new activity then becomes the current activity for the calling process.

Errors

(7) LIMIT_WOULD_BE_EXCEEDED (MAX_ACTIVITIES)

(8) LIMIT_WOULD_BE_EXCEEDED (MAX_ACTIVITIES_PER_PROCESS)

(9) If the calling process has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (new activity)

(10) VOLUME_IS_FULL (volume on which the calling process resides)

(11) NOTE - The class of an activity influences system behaviour with respect to lock durations and the default external mode of implicit locks.

16.2.4 LOCK_RESET_INTERNAL_MODE

(1)
```
LOCK_RESET_INTERNAL_MODE (
    object   : Object_designator
)
```

(2) LOCK_RESET_INTERNAL_MODE resets to READ_UNPROTECTED the internal mode of the lock associated with the object resource *object*.

(3) As result, the internal lock mode of the associated "lock" link is set to the value READ_UNPROTECTED.

Errors

(4) DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*object*, ATOMIC)

(5) LOCK_IS_NOT_EXPLICIT (*object*)

(6) OBJECT_IS_NOT_LOCKED (*object*)

(7) OBJECT_IS_OPERATED_ON (*object*, ATOMIC)

16.2.5 LOCK_SET_INTERNAL_MODE

(1)
```
LOCK_SET_INTERNAL_MODE (
    object      : Object_designator,
    lock_mode   : Lock_internal_mode,
    wait_flag   : Boolean
)
```

(2) LOCK_SET_INTERNAL_MODE promotes the internal mode of the lock on the object resource designated by *object*.

(3) If the required lock internal mode is weaker than the existing one, no action is performed.

(4) If *lock_mode* is not stronger than the internal mode of the lock currently held by the activity on *object*, then, whenever possible, the operation results in an explicit promotion of the internal mode of that lock to the weakest mode which is stronger than both the current internal mode and *lock_mode*. E.g. if the current internal mode is READ_PROTECTED and *lock_mode* is WRITE_UNPROTECTED then, if the operation succeeds, it results in the promotion of the internal mode of the existing lock to WRITE_PROTECTED.

(5) Let *new_lock_mode* be the actual value of this lock internal mode: either the specified value *lock_mode* or the value derived from it by the above promotion rule; then LOCK_SET_INTERNAL_MODE sets the internal lock mode of the associated "lock" link to *new_lock_mode*.

(6) In case of conflict between the required internal mode and other concurrent acquisitions of the resources in the concerned domain of the object resource *object* (see 16.1.7), the behaviour of the operation depends on the value of *wait_flag*:

(7) - **true** : the operation waits on the resource until it acquires the resource or until the operation is interrupted or until the process is terminated, whichever comes first;

(8) - **false**: the operation does not wait on the resource and the operation fails with the error condition LOCK_INTERNAL_MODE_CANNOT_BE_CHANGED and has no effect.

Errors

(9) DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*object*, ATOMIC)

(10) LOCK_INTERNAL_MODE_CANNOT_BE_CHANGED (*object*, *lock_mode*)

(11) LOCK_IS_NOT_EXPLICIT (*object*)

(12) LOCK_MODE_IS_TOO_STRONG (*lock_mode*, *object*)

(13) OBJECT_IS_NOT_LOCKED (*object*)

16.2.6 LOCK_SET_OBJECT

(1)
```
LOCK_SET_OBJECT (
    object      : Object_designator,
    lock_mode   : Lock_set_mode,
    wait_flag   : Boolean,
    scope       : Object_scope
)
```

(2) LOCK_SET_OBJECT either establishes a new lock on the object resource *object*, if *object* is not yet assigned to the current activity, or promotes an existing lock on *object* otherwise. If *scope* is COMPOSITE, LOCK_SET_OBJECT also does the same for the object resource of each component of the object *object*.

(3) If *lock_mode* is READ_DEFAULT, WRITE_DEFAULT or DELETE_DEFAULT, the external mode of the lock is chosen according to the class of the current activity as shown in table 10.

Table 10 - Interpretation of default lock modes

Activity class	**READ_DEFAULT**	**WRITE_DEFAULT**	**DELETE_DEFAULT**
UNPROTECTED	RUN	WUN	DUN
PROTECTED	RPR	WPR	DPR
TRANSACTION	RPR	WTR	DTR

(4) The locks are established or promoted as follows, where *resource* is the object resource *object*, and each of the object resources of the components of *object* if *scope* is COMPOSITE.

(5) If the current activity of the calling process has a lock on *resource* then the lock's external mode is promoted to *lock_mode*. If this is weaker than the lock's current external mode, the operation is successful but no action is performed.

(6) The internal mode of a new lock is set to READ_UNPROTECTED. If *lock_mode* is not stronger than the lock's current external mode, then the operation results in an explicit promotion of the external mode of the lock according to the rule of explicit promotion of an external mode defined in 16.1.5.

(7) Let *new_lock_mode* be the actual value of this lock mode: either the specified value *lock_mode* or the value derived from it as defined by the above rule of explicit promoting an external mode.

(8) If the current activity is enclosed in a transaction, LOCK_SET_OBJECT also results in an attempt to implicitly establish or to implicitly promote a lock on the object resource *resource* on behalf of the closest enclosing transaction. The external mode *implied_lock_mode* of this implied lock is derived from *lock_mode* as defined in 16.1.6, and its internal mode *implied_internal_lock_mode* is READ_UNPROTECTED.

(9) If new locks are to be established, for each of these locks, LOCK_SET_OBJECT creates a "lock" and a "locked_by" link (each reversing the other) between the activity on behalf of which the lock is established (i.e. the current activity of the calling process or its closest enclosing transaction) and the locked object. The links remain in existence until the corresponding locks are released or inherited. The keys of the "lock" and "locked_by" links are implementation-dependent.

(10) In this case LOCK_SET_OBJECT initializes the attributes of the new links as follows:

(11) - Attributes of "lock" link of the current activity:

(12) . lock duration is set to LONG if the current activity is a transaction, otherwise it is set to SHORT;

(13) . lock explicitness is set to EXPLICIT;

(14) . lock internal mode and lock external mode are set to READ_UNPROTECTED and to *new_lock_mode* respectively.

(15) - Attributes of "lock" link of the enclosing transaction:

(16) . lock duration is set to LONG;

(17) . lock explicitness is set to IMPLICIT;

(18) . lock internal mode and lock external mode are set to *implied_internal_lock_mode* and to *implied_lock_mode* respectively.

(19) The locked resource of each of these links is not set and has its initial value which is the empty string.

(20) If the locks are promoted then only the external lock mode and internal lock mode of the existing locks are modified; they are set as described above.

(21) In case of conflict between the locks required on *object* or on any of its components (if *scope* is COMPOSITE) and other concurrent acquisitions of the resources in the corresponding concerned domain, the behaviour of the operation depends on the value of *wait_flag*:

(22) - **true**: no lock is established or promoted by the operation, and the operation waits on the resource until the resource becomes available, until the operation is interrupted, or until the process is terminated, whichever comes first;

(23) - **false**: no lock is established or promoted by the operation, the operation does not wait on the resource, and the operation fails with the LOCK_COULD_NOT_BE_ESTABLISHED error condition.

(24) If *scope* is COMPOSITE, then none of the locks that the operation is trying to establish or promote are established or promoted until all of them can be established or promoted.

Errors

(25) DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*object*, *scope*)

(26) LOCK_COULD_NOT_BE_ESTABLISHED (*object*, *scope*)

(27) LOCK_MODE_IS_NOT_ALLOWED (*lock_mode*)

(28) If *scope* is ATOMIC:
OBJECT_IS_ARCHIVED (*object*)

(29) If *scope* is COMPOSITE:
OBJECT_IS_ARCHIVED (*object* or a component of *object*)

(30) OBJECT_IS_INACCESSIBLE (*object*, *scope*)

16.2.7 LOCK_UNSET_OBJECT

(1)
```
LOCK_UNSET_OBJECT (
    object  : Object_designator,
    scope   : Object_scope
)
```

(2) LOCK_UNSET_OBJECT releases the lock established by the current activity of the calling process on the object resource *object*. If *scope* is COMPOSITE, LOCK_UNSET_OBJECT also does the same thing for the object resource of each component of *object*.

(3) This results in the deletion of the "lock" and "locked_by" links associated with the released lock or locks.

Errors

(4) DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*object*, *scope*)

(5) LOCK_IS_NOT_EXPLICIT (*object*)

(6) If *scope* is ATOMIC:
OBJECT_IS_ARCHIVED (*object*)

(7) If *scope* is COMPOSITE:
OBJECT_IS_ARCHIVED (*object* or a component of *object*)

(8) OBJECT_IS_INACCESSIBLE (*object*, *scope*)

(9) OBJECT_IS_OPERATED_ON (*object*, *scope*)

(10) UNLOCKING_IN_TRANSACTION_IS_FORBIDDEN

17 Replication

17.1 Replication concepts

17.1.1 Replica sets

(1)
```
Replica_set_identifier = Natural
```

(2)
```
sds system:
```

(3)
```
replica_set_identifier: natural;
```

(4)
```
replica_set_directory: child type of object with
link
    known_replica_set: (navigate) non_duplicated existence link (replica_set_identifier)
        to replica_set reverse known_replica_set_of;
    replica_sets_of: implicit link to common_root reverse replica_sets;
end replica_set_directory;
```

(5)
```
replica_set: child type of object with
link
    master_volume: (navigate) reference link to administration_volume reverse
        master_volume_of;
    copy_volume: (navigate) reference link (volume_identifier) to administration_volume
        reverse copy_volume_of;
    known_replica_set_of: implicit link to replica_set_directory reverse known_replica_set;
end replica_set;
```

(6)
```
extend object type administration_volume with
link
    master_volume_of: (navigate) reference link (replica_set_identifier) to replica_set
        reverse master_volume;
    copy_volume_of: (navigate) reference link (replica_set_identifier) to replica_set reverse
        copy_volume;
end administration_volume;
```

(7)
```
end system;
```

(8) The replica set directory represents the set of known replica sets. Each replica set has a unique replica set identifier which is assigned to the replica set on creation and uniquely identifies the replica set within the PCTE installation.

(9) The replica set directory is the destination of a "replica_sets" link from the common root. Each known replica set is the destination of a "known_replica_set" link from the replica set directory whose key is the replica set identifier of that replica set.

(10) A replica set has exactly one master volume which is chosen when the replica set is created. It also has a set of copy volumes. Master and copy volumes are administration volumes. The key of a "copy_volume" link is the volume identifier of its destination volume. The keys of "master_volume" and "copy_volume" links are the replica set identifiers of their destination replica sets.

(11) A copy volume of a replicated set cannot also be the master volume for that same set.

17.1.2 Replicated objects

(1)
```
sds system:
```

(2)
```
extend object type replica_set with
link
    includes_object: (navigate) reference link (exact_identifier) to object reverse
        replicated_as_part_of;
end replica_set;
```

(3)
```
extend object type administration_volume with
link
    replica: (navigate) reference link (exact_identifier) to object reverse replica_on;
end administration_volume;
```

(4)
```
extend object type object with
link
    replicated_as_part_of: (navigate) implicit link to replica_set reverse includes_object;
    replica_on: implicit link to administration_volume reverse replica;
end object;
```

(5)
```
end system;
```

(6) Objects are classified as *normal*, *master* or *copy*, according to the value of the replicated state (see 9.1.1):

(7) - A *master* object has replicated state MASTER; it belongs to exactly one replica set, and it resides on the master volume of that replica set.

(8) - A *copy* object has replicated state COPY; it belongs to exactly one replica set, and is a replica on a copy volume of that replica set, but does not reside on any volume (i.e. there is no "object_on_volume" link to it; see 11.1.1).

(9) - A *normal* object has replicated state NORMAL; it belongs to no replica set, and can reside on any volume.

(10) For each master object there may be a corresponding copy object (with the same exact identifier) on each of its replica set's copy volumes; for each copy object there is a corresponding master object on its replica set's master volume. Such a set of corresponding master and copy objects is called a *replicated object*.

(11) Operations which modify a copy object are forbidden, and master objects can be modified only by processes for which PCTE_REPLICATION is an effective security group.

(12) The destinations of the "includes_object" links leaving a replica set are called the objects *replicated as part of* that replica set. The key of an "includes_object" link is the exact identifier of its destination object.

(13) If a link is created to a replicated object then the link is created to the master object.

(14) A replica set is replicated as part of itself.

(15) The following objects cannot be replicated: processes, activities, pipes, devices, execution sites, volumes, message queues, audit files, and accounting logs.

NOTES

(16) 1 There is intended to be a copy of each object of a replicated set on each of the copy volumes of the replicated set.

(17) 2 The master and all copies of a replicated object are intended to be kept identical, except for the volume identifier, last access time, replicated state, composite last access time, composite last change time, composite last modification time and "replica_on" and usage designation links. It is expected that system tools automatically propagate modifications and enforce convergence among the various copies in a PCTE installation. Instantaneous updating of all copies of a replicated object as the master evolves is not expected, so that the replication mechanism has to manage temporary inconsistencies among the various copies of the replicated objects, supporting suitable procedures for the propagation of updates.

17.1.3 Selection of an appropriate replica

(1)
```
sds system:
```

(2)
```
extend object type workstation with
link
    replica_set_chosen_volume: (navigate) designation link (replica_set_identifier) to
        administration_volume;
end workstation;
```

(3)
```
end system;
```

(4) At each moment for each workstation W and replica set S, there is a unique volume V called the *chosen volume* of W for accessing S. This is defined as follows:

(5) - it is an *explicitly chosen volume*. This explicit choice is modelled as a "replica_set_chosen_volume" link from W to V with the replica set identifier of S as its key.

(6) - it is an *implicitly chosen volume*. This implicit choice is made when there is no "replica_set_chosen_volume" link from W with the replica set identifier of S as its key.

(7) If a link destination resides on an administration volume this is considered to be a potential link to a replicated object.

(8) If a link identifies an object O that is replicated as part of a replica set S then replication redirection may occur. *Replication redirection* means that the object reached or navigated through as the destination of a link (including "replica" links) is the copy object of O on the chosen volume for accessing S of the execution site of the calling process (see 17.1.3) if such a copy object exists, and the master object of O otherwise. There is an exception for service designation links in that replication redirection applies to the state of the object base at the time the link was created rather than when it is navigated through.

(9) When an object is referenced using an internal object reference or a contents handle the designated object is always the one that was reached at the time the corresponding pathname evaluation was performed, regardless of whether a new local copy has been created or deleted since that evaluation.

(10) An object replicated as part of a replica set S can only be updated by processes having PCTE_REPLICATION privilege and running on a workstation whose chosen volume for accessing S is the master volume of S. If a link having referential integrity is created to a replicated object then the calling process must have PCTE_REPLICATION privilege.

NOTES

(11) 1 Since "replica_set_chosen_volume links" are designation links they may be replaced when the volumes that they designate become unavailable, thus allowing an alternative volume containing that replica set to be chosen.

(12) 2 A PCTE implementation may automatically decide which of the master or copy volumes of a replica set should be implicitly chosen for each workstation. This can be used to allow the PCTE implementation to minimize network load and to recover from machine or network failure.

(13) 3 A navigation to or from a replicated object should not fail because the master object is not accessible as long as the chosen volume of the workstation of the process on whose behalf the navigation is being performed contains a copy of that object.

(14) 4 Access to a direct component of an object is made as if the corresponding composition link is navigated.

17.1.4 Administration replica set

(1) There is exactly one *administration replica set*. It has replica set identifier 0. Its master volume is the master administration volume (see 11.1.2). Each administration volume other than the master administration volume is a copy volume of the administration replica set.

(2) The administration replica set includes the *predefined replicated objects*, representing certain system entities; each predefined replicated object has a master on the master administration volume and a copy on each other administration volume; a predefined replicated object may not have its replicated state changed. The predefined replicated objects are:

(3) - the common root;

(4) - the administrative objects.

(5) NOTE - Copies of predefined replicated objects cannot be deleted.

17.2 Replication operations

17.2.1 REPLICA_SET_ADD_COPY_VOLUME

(1)
```
REPLICA_SET_ADD_COPY_VOLUME (
    replica_set      : Replica_set_designator,
    copy_volume      : Administration_volume_designator
)
```

(2) REPLICA_SET_ADD_COPY_VOLUME adds *copy_volume* to the set of copy volumes for the replica set *replica_set*.

(3) A "copy_volume" link with key equal to the volume identifier of *copy_volume* is created from *replica_set* to *copy_volume*. The key of its reverse link is the replica set identifier of *replica_set*.

(4) A copy of *replica_set* is created in *copy_volume*. A "replica" link with key equal to the exact identifier of *replica_set* is created from *copy_volume* to this copy object.

(5) A read lock of the default mode is set of the master of *replica_set*. Write locks of the default modes are set on the created "copy_volume", "copy_volume_of", "replica" and "replica_on" links and the created copy of *replica_set*.

Errors

(6) ACCESS_ERRORS (*copy_volume*, ATOMIC, MODIFY, APPEND_LINKS)

(7) ACCESS_ERRORS (master of *replica_set*, ATOMIC, MODIFY, APPEND_LINKS)

(8) ACCESS_ERRORS (master of *replica_set*, ATOMIC, READ, READ_ATTRIBUTES)

(9) ACCESS_ERRORS (master of *replica_set*, ATOMIC, READ, READ_LINKS)

(10) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(11) REPLICA_SET_IS_NOT_KNOWN (*replica_set*)

(12) VOLUME_IS_ALREADY_COPY_VOLUME_OF_REPLICA_SET (*replica_set*, *copy_volume*)

(13) VOLUME_IS_MASTER_VOLUME_OF_REPLICA_SET (*replica_set*, *copy_volume*)

17.2.2 REPLICA_SET_CREATE

(1)
```
REPLICA_SET_CREATE (
    master_volume    : Administration_volume_designator,
    identifier       : Natural
)
    replica_set      : Replica_set_designator
```

(2) REPLICA_SET_CREATE creates replica set *replica_set* with master volume *master_volume*. The newly created object resides on *master_volume*, and is replicated as part of itself.

(3) A "known_replica_set" link is created from the replica set directory to *replica_set* with key *identifier*, together with its reverse link. *identifier* becomes the replica set identifier of the replica set.

(4) A "master_volume" link is created from *replica_set* to *master_volume*. The key of its reverse link is assigned the value *identifier*.

(5) An "includes_object" link is created from *replica_set* to *replica_set* with key equal to the exact identifier of *replica_set*. Its reverse link is also created.

(6) Write locks of the default modes are obtained on *replica_set* and the created "known_replica_set", "known_replica_set_of", "master_volume", "master_volume_of", "includes_object" and "replicated_as_part_of" links.

Errors

(7) ACCESS_ERRORS (replica set directory, ATOMIC, MODIFY, APPEND_LINKS)

(8) ACCESS_ERRORS (*master_volume*, ATOMIC, MODIFY, APPEND_LINKS)

(9) LINK_EXISTS (replica set directory, "known_replica_set" link from replica set directory with key *identifier*.)

(10) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

17.2.3 REPLICA_SET_REMOVE

(1)
```
REPLICA_SET_REMOVE (
    replica_set : Replica_set_designator
)
```

(2) REPLICA_SET_REMOVE removes replica set *replica_set* from the replicated set directory.

(3) The "master_volume" link from *replica_set* to its master volume is deleted, together with its reverse link.

(4) The "includes_object" link from *replica_set* to *replica_set* is deleted, together with its reverse link.

(5) The "known_replica_set" link from the replica set directory to *replica_set* is deleted, together with its reverse link. If this link is the last existence link leading to *replica_set*, *replica_set* is deleted.

(6) Write locks of the default modes are set on *replica_set* and the deleted "known_replica_set", "known_replica_set_of", "master_volume", "master_volume_of", "includes_object" and "replicated_as_part_of" links.

Errors

(7) ACCESS_ERRORS (replica set directory, ATOMIC, MODIFY, WRITE_LINKS)

(8) ACCESS_ERRORS (*replica_set*, ATOMIC, MODIFY, WRITE_LINKS)

(9) ACCESS_ERRORS (*replica_set*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(10) ACCESS_ERRORS (master volume of *replica_set*, ATOMIC, MODIFY, WRITE_LINKS)

(11) OBJECT_HAS_LINKS_PREVENTING_DELETION (*replica_set*)

(12) OBJECT_IS_IN_USE_FOR_DELETE (*replica_set*)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(14) REPLICA_SET_HAS_COPY_VOLUMES (*replica_set*)

(15) REPLICA_SET_IS_NOT_EMPTY (*replica_set*)

(16) REPLICA_SET_IS_NOT_KNOWN (*replica_set*)

17.2.4 REPLICA_SET_REMOVE_COPY_VOLUME

(1)
```
REPLICA_SET_REMOVE_COPY_VOLUME (
    replica_set    : Replica_set_designator,
    copy_volume    : Administration_volume_designator
)
```

(2) REPLICA_SET_REMOVE_COPY_VOLUME removes *copy_volume* from the set of copy volumes of the replica set replica set. The copy of *replica_set* on *copy_volume* is deleted.

(3) The "copy_volume" link with key equal to the volume identifier of *copy_volume* and its reverse link are deleted.

(4) The link of type "replica" from *copy_volume* to the copy of *replica_set* on *copy_volume* is deleted, as is its reverse link.

(5) Write locks of the default modes are set on the deleted "copy_volume", "copy_volume_of", "replica" and "replica_on" links and the deleted copy of *replica_set*.

Errors

(6) ACCESS_ERRORS (master of *replica_set*, ATOMIC, MODIFY, WRITE_LINKS)

(7) ACCESS_ERRORS (deleted copy of *replica_set*, ATOMIC, MODIFY)

(8) ACCESS_ERRORS (*copy_volume*, ATOMIC, MODIFY, WRITE_LINKS)

(9) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(10) REPLICA_SET_COPY_IS_NOT_EMPTY (*replica_set*, *copy_volume*)

(11) REPLICA_SET_IS_NOT_KNOWN (*replica_set*)

(12) VOLUME_IS_NOT_COPY_VOLUME_OF_REPLICA_SET (*replica_set*, *copy_volume*)

17.2.5 REPLICATED_OBJECT_CREATE

(1)
```
REPLICATED_OBJECT_CREATE (
    replica_set   : Replica_set_designator,
    object        : Object_designator
)
```

(2) REPLICATED_OBJECT_CREATE converts the normal object *object* to a master object belonging to replica set *replica_set*. The replicated state of *object* is set to MASTER.

(3) A "replica" link is created from the master volume of *replica_set* to *object* with key equal to the exact identifier of *object*. Its reverse link is also created to the master volume.

(4) An "includes_object" link is created from *replica_set* to *object* with key equal to the exact identifier of *object*. Its reverse link is also created.

(5) A write lock of the default mode is obtained on *object* and a write lock of the default mode is obtained on the created "replica" link.

Errors

(6) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, CONTROL_OBJECT)

(7) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(8) ACCESS_ERRORS (*replica_set*, ATOMIC, MODIFY, APPEND_LINKS)

(9) ACCESS_ERRORS (master volume of *replica_set,* ATOMIC, MODIFY, APPEND_LINKS)

(10) OBJECT_IS_NOT_ON_MASTER_VOLUME_OF_REPLICA_SET (*replica_set*, *object*)

(11) OBJECT_IS_REPLICATED (*object*)

(12) OBJECT_IS_NOT_REPLICABLE (*object*)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(14) REPLICA_SET_IS_NOT_KNOWN (*replica_set*)

(15) The following implementation-dependent errors may be raised for any object X with a link to *object*:
VOLUME_IS_NOT_MOUNTED (X, ATOMIC)
VOLUME_IS_READ_ONLY (X, ATOMIC)

17.2.6 REPLICATED_OBJECT_DELETE_REPLICA

(1)
```
REPLICATED_OBJECT_DELETE_REPLICA (
    object        : Object_designator,
    copy_volume   : Administration_volume_designator
)
```

(2) REPLICATED_OBJECT_DELETE_REPLICA deletes the copy object *object* from the volume *copy_volume*. The "replica" link leading to *object* and its reverse "replica_on" link are deleted.

(3) If the copy object has contents and this is currently opened by one or more processes, the deletion of the contents is postponed until all processes have closed the contents; i.e. the object is no longer accessible for example using internal object references or for replication redirection, but an operation using a contents handle to access its contents is not affected by the deletion until the contents handle is closed.

(4) Write locks of the default mode are obtained on *object* and on the deleted replica link.

Errors

(5) ACCESS_ERRORS (*copy_volume*, ATOMIC, MODIFY, WRITE_LINKS)

(6) OBJECT_IS_NOT_REPLICATED_ON_VOLUME (*object*, *copy_volume*)

(7) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(8) OBJECT_IS_PREDEFINED_REPLICATED (*object*)

(9) OBJECT_IS_A_REPLICA_SET (*object*)

(10) STATIC_CONTEXT_IS_IN_USE (*object*)

(11) The following implementation-dependent error may be raised:
ACCESS_ERRORS (master of *object*, ATOMIC, CHANGE)

17.2.7 REPLICATED_OBJECT_DUPLICATE

(1)
```
REPLICATED_OBJECT_DUPLICATE (
    object          : Object_designator,
    volume          : Administration_volume_designator,
    copy_volume     : Administration_volume_designator
)
```

(2) If *volume* and *copy_volume* are the same, then REPLICATED_OBJECT_DUPLICATE has no effect.

(3) If *volume* and *copy_volume* are not the same, and a copy of *object* does not already exist in *copy_volume*, a copy is created and a "replica" link, keyed by the exact identifier of *object*, is created from *copy_volume* to the new copy, together with its reverse "replica_on" link.

(4) If *volume* and *copy_volume* are not the same, and a copy of *object* already exists in *copy_volume*, the copy in *copy_volume* is updated as defined below.

(5) On completion, the atomic object of the copy in *copy_volume* is identical to the atomic object of *object* except for the following:

(6) - The volume identifier of the copy object is set to the volume identifier of *copy_volume*.

(7) - The last access time of the copy object is set to the value of the system clock at the time of call.

(8) - The destination of its "replica_on" link is *copy_volume*.

(9) - The replicated state of the copy object is set to COPY.

(10) - Usage designation links are not copied to the copy in *copy_volume*.

(11) - If *object* has contents and there is a copy of *object* in *copy_volume*, the effect is that of CONTENTS_TRUNCATE followed by CONTENTS_WRITE with the contents of *object*.

(12) A write lock of the default mode is obtained on the copy object, and a read lock of the default mode is obtained in *object*. Write locks of the default mode are obtained on the "replica" link and its reverse "replica_on" link, if created.

Errors

(13) ACCESS_ERRORS (copy of *object* on *volume*, ATOMIC, READ)

(14) If a new "replica" link is created:
ACCESS_ERRORS (*copy_volume*, ATOMIC, MODIFY, APPEND_LINKS)

(15) OBJECT_IS_NOT_REPLICATED_ON_VOLUME (*object*, *volume*)

(16) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(17) REPLICATED_COPY_IS_IN_USE (*object*)

(18) STATIC_CONTEXT_IS_IN_USE (*object*)

(19) VOLUME_IS_NOT_MASTER_OR_COPY_VOLUME_OF_REPLICA_SET (*volume*, replica set of *object*)

(20) VOLUME_IS_NOT_COPY_VOLUME_OF_REPLICA_SET (*copy_volume*, replica set of *object*)

(21) VOLUME_IS_MASTER_VOLUME_OF_REPLICA_SET (*copy_volume*, replica set of *object*)

(22) The following implementation-dependent error may be raised:
ACCESS_ERRORS (master of *object*, ATOMIC, CHANGE)

NOTES

(23) 1 REPLICATED_OBJECT_DUPLICATE causes a copy of the atomic object of *object* to exist as the atomic object of the copy object in the volume *copy_volume*. The copy object has the same components as *object*, but the components are not copied.

(24) 2 Updates to copy objects by this operation are subject to transaction rollback.

17.2.8 REPLICATED_OBJECT_REMOVE

(1)
```
REPLICATED_OBJECT_REMOVE (
    object : Object_designator
)
```

(2) REPLICATED_OBJECT_REMOVE removes the master object *object* from the replica set to which it belongs by changing it into a normal object residing on the master volume of this replica set. The replicated state of *object* is set to NORMAL, and the "replica" and "includes_object" links leading to the object are deleted, together with their reverse "replica_on" and "replicated_as_part_of" links.

(3) Write locks of the default mode are obtained on the deleted "replica", "replica_on", "includes_object" and "replicated_as_part_of" links, and on *object*.

Errors

(4) ACCESS_ERRORS (master volume of *object*, ATOMIC, MODIFY, WRITE_LINKS)

(5) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, CONTROL_OBJECT)

(6) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(7) ACCESS_ERRORS (replica set containing *object*, ATOMIC, MODIFY, WRITE_LINKS)

(8) OBJECT_HAS_COPIES (*object*)

(9) OBJECT_IS_A_REPLICA_SET (*object*)

(10) OBJECT_IS_NOT_MASTER_REPLICATED_OBJECT (*object*)

(11) OBJECT_IS_PREDEFINED_REPLICATED (*object*)

(12) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(13) The following implementation-dependent errors may be raised for any object X with a link to *object*:
VOLUME_IS_NOT_MOUNTED (X, ATOMIC)
VOLUME_IS_READ_ONLY (X, ATOMIC)

17.2.9 WORKSTATION_SELECT_REPLICA_SET_VOLUME

(1)
```
WORKSTATION_SELECT_REPLICA_SET_VOLUME (
    station     : Workstation_designator,
    replica_set : Replica_set_designator,
    volume      : Administration_volume_designator
)
```

(2) WORKSTATION_SELECT_REPLICA_SET_VOLUME selects *volume* as the chosen volume for accesses to *replica* set by *station*.

(3) If *station* has a "replica_set_chosen_volume" link whose key is the replica set identifier of *replica_set*, that link is first deleted.

(4) A "replica_set_chosen_volume" link with a key equal to the replica set identifier of *replica_set* is then created from *station* and leading to *volume*.

(5) A write lock of the default mode is created on the "replica_set_chosen_volume" link.

Errors

(6) If *station* already has a chosen volume for accesses to *replica_set*:
ACCESS_ERRORS (*station*, ATOMIC, MODIFY, WRITE_LINKS)

(7) ACCESS_ERRORS (*station*, ATOMIC, MODIFY, APPEND_LINKS)

(8) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(9) REPLICA_SET_IS_NOT_KNOWN (*replica_set*)

(10) VOLUME_IS_NOT_MASTER_OR_COPY_VOLUME_OF_REPLICA_SET (*replica_set*, *volume*)

17.2.10 WORKSTATION_UNSELECT_REPLICA_SET_VOLUME

(1)
```
WORKSTATION_UNSELECT_REPLICA_SET_VOLUME (
    station     : Workstation_designator,
    replica_set : Replica_set_designator
)
```

(2) WORKSTATION_UNSELECT_REPLICA_SET_VOLUME deletes the "replica_set_chosen_volume" link from *station* whose key is the replica set identifier of *replica_set*.

(3) A write lock of the default mode is created on the deleted "replica_set_chosen_volume" link.

Errors

(4) ACCESS_ERRORS (*station*, ATOMIC, MODIFY, WRITE_LINKS)

(5) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(6) REPLICA_SET_IS_NOT_KNOWN (*replica_set*)

(7) WORKSTATION_HAS_NO_CHOICE_OF_VOLUME_FOR_REPLICA_SET (*station*, *replica_set*)

18 Network connection

18.1 Network connection concepts

18.1.1 Execution sites

(1)
```
sds system:
```

(2)
```
execution_site_directory: child type of object with
link
    known_execution_site: non_duplicated existence link (execution_site_identifier) to
        execution_site;
    execution_sites_of: implicit link to common_root reverse execution_sites;
end execution_site_directory;
```

(3)
```
execution_site: child type of object with
link
    running_process: (navigate) non_duplicated designation link (number) to process;
end execution_site;
```

(4)
```
end system;
```

(5) The execution site identifier is assigned to the execution site on creation and uniquely identifies the execution site within the PCTE installation during its existence.

(6) The destinations of the "running_process" links, if any, are the processes running on the workstation (see 13.1.4).

(7) The execution site directory is an administrative object (see 9.1.2).

(8) NOTE - An execution site is either a workstation (see 18.1.2) or a foreign system (see 18.1.3).

18.1.2 Workstations

(1)
```
Work_status = set of Work_status_item
```

(2)
```
Work_status_item = ACTIVITY_REMOTE_LOCKS | ACTIVITY_LOCAL_LOCKS |
    TRANSACTION_REMOTE_LOCKS | TRANSACTION_LOCAL_LOCKS |
    QUEUE_REMOTE | QUEUE_LOCAL | RECEIVE_REMOTE | RECEIVE_LOCAL |
    CHILD_REMOTE | CHILD_LOCAL
```

(3)
```
Requested_connection_status = LOCAL | CLIENT | CONNECTED
```

(4)
```
Connection_status = Requested_connection_status | AVAILABLE
```

(5)
```
Workstation_status = Connection_status * Work_status
```

(6)
```
New_administration_volume ::
    FOREIGN_DEVICE              : String
    ADMINISTRATION_VOLUME       : Volume_identifier
    VOLUME_CHARACTERISTICS      : String
    DEVICE                      : Device_identifier
    DEVICE_CHARACTERISTICS      : String
```

(7) **sds** system:

(8)
```
workstation: child type of execution_site with
attribute
    connection_status: (read) non_duplicated enumeration  (LOCAL, CLIENT,
        AVAILABLE, CONNECTED) := LOCAL;
    PCTE_implementation_name: (read) non_duplicated string;
    PCTE_implementation_release: (read) non_duplicated string;
    PCTE_implementation_version: (read) non_duplicated string;
    node_name: (read) non_duplicated string;
    machine_name: (read) non_duplicated string;
link
    controlled_device: (navigate) non_duplicated existence link (device_identifier:
        natural) to device reverse device_of;
    associated_administration_volume: (navigate) non_duplicated designation link to
        administration_volume;
    initial_process: non_duplicated existence link (number) to process;
    outermost_activity: (navigate) non_duplicated existence link (number) to activity;
end workstation;
```

(9) **end** system;

(10) The work status consists of a number of independent work status items as follows (where *local* means residing on a volume mounted on a device controlled by this workstation, and *remote* means residing on a volume mounted on a device controlled by some other workstation):

(11) - ACTIVITY_REMOTE_LOCKS: at least one non-transaction activity started on the workstation holds locks on remote objects,

(12) - ACTIVITY_LOCAL_LOCKS: at least one non-transaction activity started on another workstation has locks on local objects,

(13) - TRANSACTION_REMOTE_LOCKS: at least one transaction started on the workstation holds locks on remote objects,

(14) - TRANSACTION_LOCAL_LOCKS: at least one transaction started on another workstation has locks on local objects,

(15) - QUEUE_REMOTE: at least one process on the workstation has a remote reserved message queue,

(16) - QUEUE_LOCAL: at least one process on a remote workstation has a message queue reserved on the workstation,

(17) - RECEIVE_REMOTE: at least one process on the workstation is waiting for the reception of a message from a remote message queue,

(18) - RECEIVE_LOCAL: at least one process on another workstation is waiting for the reception of a message from a local message queue,

(19) - CHILD_REMOTE: at least one process on the workstation has one or more unterminated remote child processes,

(20) - CHILD_LOCAL: at least one process on another workstation has one or more unterminated local child processes.

(21) New administration volumes are used in WORKSTATION_CREATE. The meaning of FOREIGN_DEVICE is implementation-defined. It is a string used to designate a new physical resource (i.e. not yet represented by a device object).

(22) A workstation A is a *client* of a workstation B if at least one of the following is true:

(23) - a process running on A has started a child process on B which is not yet terminated;

(24) - a process running on A is accessing (i.e. reading from, writing to, or navigating through) an object residing on a volume mounted on a device controlled by B;

(25) - there is a service designation link to an object residing on a volume mounted on a device controlled by B from a process running on A;

(26) - a process running on A has reserved a message queue whose associated message queue object resides on a volume mounted on a device managed by B.

(27) Conversely, a workstation A is a *server* of a workstation B if B is a client of A.

(28) The connection status denotes the status of the workstation with respect to other workstations of the PCTE installation. The values have the following meanings (for the definitions of client and server see below).

(29) - LOCAL The workstation cannot be a client or a server for another workstation. It does not respond to a call of WORKSTATION_CONNECT from another workstation.

(30) - AVAILABLE The same as LOCAL except that the workstation responds to a connection request from another workstation.

(31) - CLIENT The workstation can be a client but not a server of another workstation. It does not respond to a connection request from another workstation.

(32) - CONNECTED The workstation can be a client or a server of another workstation.

(33) The implementation name is the name of the particular implementation of PCTE running on the workstation; it is implementation-defined.

(34) The implementation release identifies of the release of the PCTE implementation running on the workstation; it is implementation-defined.

(35) The implementation version identifies of the version of the PCTE implementation running on the workstation; it is implementation-defined.

(36) The node name provides the mechanism to communicate to the network, e.g. the local area network address; it is implementation-defined.

(37) The machine name is the name of the particular machine type of the workstation; it is implementation-defined.

(38) The controlled devices are also called the devices *controlled by* the workstation. Each of the devices is identified by a device identifier which is unique within the set of devices controlled by the workstation.

(39) For the administration volume, see 11.1.2. A workstation object resides on the administration volume of the workstation.

(40) For the initial process of the workstation, see 13.1.5.

(41) For the outermost activity of the workstation, see 16.1.1.

(42) For the associated accounting log, see 22.1.2.

(43) A workstation is *busy* if it has connection status CONNECTED and is a server of another workstation, or has connection status CLIENT and is a client of another workstation.

(44) Within an operation, the *local* workstation is the workstation on which the calling process is executed.

NOTES

(45) 1 The normal situation in a PCTE installation is one of the following, though abnormal situations may occur:

(46) - all workstations with connection status LOCAL;

(47) - one workstation with connection status AVAILABLE, all other workstations with connection status LOCAL;

(48) - two or more workstations with connection status CONNECTED or CLIENT, all other workstations with connection status LOCAL;

(49) - all workstations with connection status CONNECTED or CLIENT.

(50) 2 In some implementations a workstation may have more than one "initial_process" or "outermost_activity" link. Only the destinations with the highest key are the initial process and outermost activity of the workstation, respectively. Destinations with other keys are remnants from previous sessions which allow an implementation-dependent tool to perform actions following workstation or system failure.

(51) 3 A workstation may access an administration volume shared with another workstation even if its connection status is LOCAL.

18.1.3 Foreign systems

```
(1)  sds system:

(2)  foreign_system: child type of execution_site with
     attribute
         system_class: enumeration (FOREIGN_DEVICE, BARE_MACHINE,
             HAS_EXECUTIVE_SYSTEM, SUPPORTS_IPC_AND_CONTROL,
             SUPPORTS_MONITOR) := BARE_MACHINE;
     end foreign_system;

(3)  end system;
```

(4) The system class indicates the level of interaction which is supported between PCTE processes and foreign processes started on the foreign system, as follows.

(5) - FOREIGN_DEVICE The foreign system can be used only as the foreign system for operations defined in 18.3.

(6) - BARE_MACHINE The foreign system is a bare machine executing no code other than the software under development. The only permitted operation by a PCTE process is PROCESS_CREATE. Any further communication is prevented by the absence of any communication agent on the foreign system

(7) - HAS_EXECUTIVE_SYSTEM The foreign system is a foreign executive system which accepts the creation, starting, and termination of processes on it, and can signal the end of their execution to the creating host process.

(8) - SUPPORTS_IPC_AND_CONTROL As for HAS_EXECUTIVE_SYSTEM and can also support at least the message queue mechanisms represented by the operations (see clause 14) MESSAGE_RECEIVE_NO_WAIT, MESSAGE_RECEIVE_WAIT, MESSAGE_SEND_NO_WAIT, and MESSAGE_SEND_WAIT, and the process control mechanisms such as process suspension and resumption.

(9) - SUPPORTS_MONITOR As for SUPPORTS_IPC_AND_CONTROL and can also support the monitoring operations of 13.5.

(10) NOTE - On a foreign system of system class, BARE_MACHINE, PROCESS_CREATE is intended to download the process but not to start its execution.

18.1.4 Network partitions

(1) The execution sites of a PCTE installation may be connected together to share resources.

(2) At any time, a set of workstations of a PCTE installation, each of which is running PCTE, and which are connected together, is called a *network partition*. The connection status of each workstation in the network partition controls the use which one workstation may make of resources controlled by another (see below).

(3) An implementation may impose restrictions on the sets of workstations which can form a network partition. In particular, an implementation may or may not allow any single isolated workstation to be a network partition, and an implementation may or may not allow more than one network partition to exist at the same time (in this case the sets of workstations in the network partitions are disjoint since the connectedness relation is transitive).

(4) The specification of the abstract operations must always be understood to be in the context of the calling process's network partition; for example, "an object is not accessible" must always be understood to mean that the object is not accessible within the calling process's network partition; the object may be accessible in some other network partition.

(5) A network partition may rejoin other partitions by implementation-defined means so that workstations in the partition are now accessible to workstations in other partitions. Although the time at which the network failure is detected may be variable, the network failure must be detected in all partitions before the network partitions can be rejoined.

18.1.5 Accessibility

(1) In order for an operation to operate on an entity, that entity must be *accessible*.

(2) The rules for accessibility are as follows; except as given by these rules, no entity in a PCTE installation is accessible:

(3) - the local workstation is accessible;

(4) - if the local workstation has connection status CLIENT or CONNECTED, all workstations in the same network partition which have connection status CONNECTED are accessible;

(5) - processes executing on accessible workstations are accessible;

(6) - devices controlled by accessible workstations are accessible;

(7) - volumes mounted on accessible devices are accessible;

(8) - objects residing on accessible volumes, or which are replicas on accessible administration volumes, and their direct attributes, direct outgoing links, contents, and associated sequences of messages are accessible;

(9) - the atomic object associated with an accessible object is accessible;

(10) - locks on resources residing on accessible volumes are accessible;

(11) - the accessibility of a foreign system is implementation-defined;

(12) - a process on an accessible foreign system is accessible;

(13) - the accessibility of a file residing on a foreign system is implementation-defined.

(14) A PCTE implementation may arrange that an operation succeeds even though some entity which is used is not accessible according to the above rules. Otherwise an error is raised to indicate

that the entity is not accessible, and the above rules indicate what must be done by the user to make the entity accessible.

(15) An object is defined as *unreachable* if operations fail to access it because it is not accessible.

(16) When a workstation A *ceases to be a client* of a workstation B the following actions occur:

(17) - All child processes running on B started by unreachable processes running on A are terminated in the same way as by calling PROCESS_TERMINATE (child process, FORCED_TERMINATION).

(18) - All objects residing on volumes mounted on devices controlled by B and with contents opened by unreachable processes running on A are closed.

(19) - All locks on objects or links residing on volumes mounted on devices controlled by B, or on deleted objects which resided on such volumes, and held by unreachable activities started by processes running on A, are treated as for activity abortion. If such a lock had external mode WTR or DTR then any changes are rolled back.

(20) - Operations being executed by processes running on A accessing unreachable objects residing on volumes mounted on devices controlled by B may fail with the error condition OPERATION_IS_INTERRUPTED.

(21) - All message queues residing on volumes mounted on devices controlled by B and reserved for unreachable processes running on A are unreserved, handlers on those message queues are disabled, and notifiers on those message queues are deleted.

(22) - Usage designation links from objects residing on volumes mounted on devices controlled by B to unreachable processes running on A are deleted.

(23) When a station A *ceases to be a server* of a station B the following actions occur:

(24) - All unreachable objects residing on volumes mounted on devices controlled by A and with contents opened by processes running on B are closed. Any subsequent reads and writes to the contents fail.

(25) - All locks on unreachable objects or links residing on volumes mounted on devices controlled by A held by activities started by processes running on B are released. These activities are not aborted immediately in their own workstations. However, any attempt to end a transaction activity which held such locks with external mode RPR, WTR or DTR results in the error TRANSACTION_CANNOT_BE_COMMITTED.

(26) - Operations being executed by processes running on B accessing unreachable objects which are residing on volumes mounted on devices controlled by A may fail with the error condition OPERATION_IS_INTERRUPTED.

(27) - All unreachable message queues residing on volumes mounted on devices controlled by A and reserved for processes running on other workstations are unreserved.

(28) - All "process_waiting_for" links from processes running on B to unreachable objects residing on volumes mounted on devices controlled by A are deleted, and the corresponding waiting operations in the processes fail with the error condition OBJECT_IS_INACCESSIBLE.

(29) - Service designation links from processes running on B to unreachable objects residing on volumes mounted on devices controlled by A are deleted.

(30) - For each "notifier" link whose origin message queue resides on a volume mounted on a device controlled by B and whose destination resides on a volume mounted on a device controlled by A, a NOT_ACCESSIBLE_MSG message is sent to its origin message queue.

(31) If a network failure is detected by a workstation A then an inaccessible workstation B normally ceases to be a client of A and also ceases to be a server of A, and A ceases to be a client or server of B. However if B fails to detect the network failure (e.g. because it was only transient and connection at the communications protocol level has been restored) then any subsequent attempt by B to act as a server or client of A results in A responding so as to cause B to cease to be a server or client of A.

18.1.6 Workstation closedown

(1) An *orderly closedown* of a workstation occurs only when all the descendant processes of the initial process associated with the workstation and any other processes executing on the workstation, with the exception of the initial process, are terminated. The outermost activity is terminated.

(2) If a workstation is improperly terminated or fails, this is termed *abnormal closedown*. On abnormal closedown of a workstation, the following actions are taken:

(3) - Each process P executing on the workstation at the time of workstation failure is terminated as by PROCESS_TERMINATE (P, SYSTEM_FAILURE). In particular, all activities started by the processes are aborted.

(4) - The outermost activity of the workstation is terminated abnormally.

(5) - Contents of an implementation-dependent set of pipes managed by the workstation are lost. The messages and the values of the "reader_waiting", "writer_waiting", "space_used", "message_count", "last_send_time", and "last_receive_time" attributes of an implementation-dependent set of message queues managed by the workstation are lost.

(6) - All locks on resources, residing on volumes mounted on devices controlled by the failed workstation, and held by activities started by processes executing on other workstations, are released as if the activities were abnormally terminated, and any updates performed under WTR or DTR locks are rolled back.

(7) - All objects residing on volumes mounted on devices controlled by the failed workstation with contents opened by processes running on other workstations are closed.

(8) - All message queues residing on volumes mounted on devices controlled by the failed workstation and reserved for processes running on other workstations are unreserved, handlers on those message queues are disabled, and notifiers on those message queues are deleted.

(9) - Usage designation links from objects residing on volumes mounted on devices controlled by the failed workstation to processes running on other workstations are deleted.

(10) - Updates to objects residing on volumes mounted on devices controlled by the failed workstation which had not been made permanent at the actual time of workstation failure are lost.

(11) - Updates to files not under WTR or DTR locks, and to audit files and accounting logs residing on volumes mounted on devices controlled by the failed workstation are lost to an implementation-defined degree.

(12) - The workstation connection status is set to LOCAL.

(13) The terminated outermost activity and the terminated initial process remain.

(14) Whether a workstation closes down in an abnormal or orderly manner, the contents of pipes, messages in message queues, and audit criteria are lost.

(15) When a workstation is restarted after abnormal or orderly closedown, a new outermost activity and a new initial process are created, and the previous, terminated, outermost activity and initial process are not reused.

NOTES

(16) 1 The actions described above at workstation abnormal closedown are performed by the implementation at some point between the failure and restarting the workstation.

(17) 2 As a consequence of terminating all the processes on abnormal closedown, any active activities are aborted.

(18) 3 After abnormal closedown the previous initial process may have components, i.e. be a tree of process objects.

(19) 4 The previously running initial process and the previously active outermost activity must be deleted explicitly. This means that it is possible that there are several "initial_process" and several "outermost_activity" links emanating from a workstation. However, only one "initial_process" link leads to a running initial process and only one "outermost_activity" link leads to an active outermost activity.

(20) 5 Implementations aiming for high security may wish to take special measures to ensure that workstation failure does not result in any loss of data written to the audit file.

18.2 Network connection operations

18.2.1 WORKSTATION_CONNECT

(1)
```
WORKSTATION_CONNECT (
    status  : Requested_connection_status
)
```

(2) WORKSTATION_CONNECT has no effect if the connection status of the local workstation is already the requested status *status*. Otherwise it connects the local workstation to the PCTE installation, and sets its connection status as follows:

(3) - If *status* is CONNECTED and there is no other workstation in the PCTE installation available for connection, the connection status is set to AVAILABLE.

(4) - If *status* is CLIENT and there is no other workstation in the PCTE installation available for connection, the connection status is unchanged.

(5) - If *status* is CONNECTED or CLIENT and there is another workstation in the PCTE installation available for connection, the connection status is set to *status*. The connection status of all available workstations are automatically changed to CONNECTED.

(6) There may be installation-defined procedures to be carried out before and after calling this operation; such procedures are outside the scope of this ECMA Standard.

Errors

(7) ACCESS_ERRORS (the local workstation, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(8) CONNECTION_IS_DENIED

(9) LAN_ERROR_EXISTS

(10) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(11) STATUS_IS_BAD (*status*)

18.2.2 WORKSTATION_CREATE

(1)
```
WORKSTATION_CREATE (
    execution_site_identifier   : Natural,
    administration_volume       : Volume_designator | New_administration_volume,
    access_mask                 : Atomic_access_rights,
    node_name                   : Text,
    machine_name                : Text
)
```

(2) WORKSTATION_CREATE creates a new workstation in the PCTE installation, as follows.

(3) If *administration_volume* is a volume designator:

(4) - a "workstation" object *new_station* is created on *administration_volume* as destination of a new "known_execution_site" link from the execution site directory keyed by *execution_site_identifier*;

(5) - an "object_on_volume" link is created from *existing_administration_volume* to *new_station*, keyed by the exact identifier of *new_station*.

(6) If *administration_volume* is a new administration volume with foreign device *foreign_device*, administration volume *new_administration_volume*, volume characteristics *volume_characteristics*, device *new_device*, and device characteristics *device_characteristics*:

(7) - *foreign_device* is interpreted in an implementation-defined way to specify a device containing a physical volume that has been prepared in an implementation-defined way to become a new administration volume.

(8) - an "administration_volume" object is created on the specified physical volume, with volume characteristics *volume_characteristics*, as destination of a new "known_volume" link from the volume directory, keyed by *new_administration_volume*;

(9) - a "workstation" object *new_station* is created on the new volume, as destination of a new "known_execution_site" link from the execution site directory keyed by *execution_site_identifier*;

(10) - a "device_supporting_volume" object is created on the new volume, with device characteristics *device_characteristics*, as destination of a new "controlled_device" link from *new_station*, keyed by *new_device*;

(11) - "object_on_volume" links are created from the new administration volume to itself, to *new_station*, and to the new "device_supporting_volume" object, keyed by the exact identifiers of their destinations;

(12) - the labels of the created device and of the created volume are set to the mandatory context of the calling process;

(13) - a "mounted_volume" link is created from the new device to the new administration volume;

(14) - a "copy_volume" link with key *volume_identifier* is created to the newly created administration volume object, and is reversed by a "copy_volume_of" link with key '0' leading to the administration replica set. Copies of the master objects of the administration replica set, the common root, the "system" schema, and the administrative objects are created in the newly created administration volume.

(15) In both cases:

(16) - *access_mask* is used in conjunction with the default atomic ACL and default owner of the calling process to define the atomic ACL and the composite ACL which are to be associated with the created objects (see 19.1.4);

(17) - the labels of the created workstation are set to the mandatory context of the calling process;

(18) - an "associated_administration_volume" link is created from new_station to existing_administration_volume or new_administration_volume;

(19) - an initial process is created for the workstation;

(20) - the attributes of the new workstation are set as follows:

(21) . the connection status is set to LOCAL;

(22) . the PCTE implementation name, PCTE implementation release, and PCTE implementation version are the same as for the local workstation;

(23) . the node name and machine name are set from the parameters *node_name* and *machine_name*.

(24) If the auditing module is supported, there is at least one audit file for the new workstation (see 21.1.1), but auditing is initially disabled.

(25) Write locks are obtained on the execution site directory, the volume directory, the created workstation and (if they are created) the new administration volume and device.

Errors

(26) ACCESS_ERRORS (volume directory, ATOMIC, CHANGE, APPEND_LINKS)

(27) ACCESS_ERRORS (execution site directory, ATOMIC, CHANGE, APPEND_LINKS)

(28) If *administration_volume* is a volume designator:
ACCESS_ERRORS (*administration_volume*, ATOMIC, MODIFY, APPEND_LINKS)
ACCESS_ERRORS (existing device, ATOMIC, CHANGE, APPEND_IMPLICIT)
ACCESS_ERRORS (administration replica set, ATOMIC, MODIFY, APPEND_LINKS)
VOLUME_IS_UNKNOWN (*administration_volume*)

(29) CONTROL_WOULD_NOT_BE_GRANTED (*new_station*)

(30) If *administration_volume* is a new administration volume:
FOREIGN_DEVICE_IS_INVALID (*foreign_device*)
VOLUME_EXISTS (*new_administration_volume*)
VOLUME_IDENTIFIER_IS_INVALID (*new_administration_volume*)

(31) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL

(32) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(33) PROCESS_IS_IN_TRANSACTION

(34) WORKSTATION_EXISTS (*execution_site_identifier*)

(35) WORKSTATION_IDENTIFIER_IS_INVALID (*execution_site_identifier*)

NOTES

(36) 1 The new physical administration volume may need to be initialized by a system tool before this operation is invoked.

(37) 2 For bootstrapping reasons, this operation cannot apply to the first workstation of a PCTE installation.

(38) 3 The new workstation is created but is not yet initialized. It is an implementation-defined procedure which is responsible for starting the initial process of the new created workstation.

(39) 4 The ability to provide an existing administration volume is intended to cater for discless workstations and other cases of shared administration volumes.

18.2.3 WORKSTATION_DELETE

(1)
```
WORKSTATION_DELETE (
    station : Workstation_designator
)
```

(2) WORKSTATION_DELETE deletes a workstation from the PCTE installation.

(3) If the administration volume of *station* is mounted on a device which is controlled by another workstation (which implies that they share the same administration volume), the effect of the workstation deletion is the same as:

OBJECT_DELETE (execution site directory, *execution_site_link*)

where *execution_site_link* is the "known_execution_site" link from the execution site directory to *station*.

(4) If the administration volume of *station* is mounted on a device which is controlled by *station*, the workstation deletion is only possible if and only if:

(5) - no other workstation has the same administration volume, i.e. there is only one "associated_administration_volume" link to the administration volume;

(6) - the only objects residing on or which are replicas on the administration volume are:

(7) . *station*;

(8) . copies of the administration replica set;

(9) . the administration volume;

(10) . the "device_supporting_volume" object which is the destination of a "controlled_device" link from *station* and the origin of the "mounted_volume" link to the administration volume;

(11) . terminated processes and activities;

(12) - there are no reference, composition, or existence links from an object residing on another volume to the objects residing on the administration volume, except the "known_volume" link from the volume directory to the administration volume, the "known_execution_site" link from the execution site directory to *station*, and "audit" links from *station*.

(13) The objects residing on and which are replicas on the administration volume are deleted, the space previously occupied by the volume is freed, the "copy_volume" link from the administration replica set to the administration volume, and the "known_volume" link to the administration volume and the "known_execution_site" link to *station* are deleted. The administration volume is unmounted.

(14) Write locks are obtained on the deleted workstation, the deleted administration volume, the deleted device supporting the administration volume, the administration replica set, and the deleted links. These locks do not prevent the dismounting and deletion of the administration volume.

Errors

(15) ACCESS_ERRORS (execution site directory, ATOMIC, MODIFY, WRITE_LINKS)

(16) ACCESS_ERRORS (volume directory, ATOMIC, MODIFY, WRITE_LINKS)

(17) ACCESS_ERRORS (*station*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(18) ACCESS_ERRORS (*station*, ATOMIC, MODIFY, WRITE_CONTENTS)

(19) If the conditions hold for deletion of the "workstation" object *station*:
ACCESS_ERRORS (*station*, COMPOSITE, MODIFY, DELETE)

(20) OBJECT_IS_IN_USE_FOR_DELETE (*station*)

(21) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(22) PROCESS_IS_IN_TRANSACTION

(23) VOLUME_HAS_OTHER_LINKS (*administration_volume*)

(24) VOLUME_HAS_OTHER_OBJECTS (*administration_volume*)

(25) VOLUME_HAS_OBJECTS_IN_USE (administration volume of *station*)

(26) WORKSTATION_IS_CONNECTED (*station*)

(27) WORKSTATION_IS_UNKNOWN (*station*)

(28) NOTE - Additional implementation-defined restrictions may be defined for this operation.

18.2.4 WORKSTATION_DISCONNECT

(1)
```
WORKSTATION_DISCONNECT (
)
```

(2) WORKSTATION_DISCONNECT changes the connection status of the local workstation to LOCAL, unless the connection status is already LOCAL, in which case it has no effect.

Errors

(3) ACCESS_ERRORS (local workstation, ATOMIC, WRITE)

(4) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(5) WORKSTATION_IS_BUSY (local workstation)

18.2.5 WORKSTATION_GET_STATUS

(1)
```
WORKSTATION_GET_STATUS (
    station : [ Workstation_designator ]
)
    status  : Workstation_status
```

(2) WORKSTATION_GET_STATUS returns the current connection status and work status of *station* in *status*. If *station* is not supplied, the local workstation is assumed.

Errors

(3) ACCESS_ERRORS (*station*, ATOMIC, READ)

(4) WORKSTATION_IS_UNKNOWN (*station*)

18.2.6 WORKSTATION_REDUCE_CONNECTION

(1)
```
WORKSTATION_REDUCE_CONNECTION (
    station : [ Workstation_designator ],
    status  : Requested_connection_status,
    force   : Boolean
)
```

(2) WORKSTATION_REDUCE_CONNECTION reduces the connection status of the workstation *station* to the connection status *status*. If *station* is not supplied, the local workstation is assumed.

(3) If the required change of status is a *disconnection*, i.e. the current status of *station* is CONNECTED and the required status is CLIENT or LOCAL, or the current status is CLIENT and the required status is LOCAL, then *force* has the following effect.

(4) - **true**: The operation is performed whether *station* is busy or not. If the connection status change is from CLIENT to LOCAL, the station ceases to be a client. If the connection status change is from CONNECTED to CLIENT, the station ceases to be a server. If the connection status change is from CONNECTED to LOCAL, the station ceases to be a client or a server.

(5) - **false**: The workstation *station* must not be busy (see 18.1.2).

(6) In other cases *force* has no effect.

Errors

(7) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, CONTROL_OBJECT)

(8) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(9) WORKSTATION_IS_BUSY (*station*)

(10) WORKSTATION_IS_NOT_CONNECTED (*station*)

(11) WORKSTATION_IS_UNKNOWN (*station*)

18.3 Foreign system operations

18.3.1 CONTENTS_COPY_FROM_FOREIGN_SYSTEM

(1)
```
CONTENTS_COPY_FROM_FOREIGN_SYSTEM (
    file                    : File_designator,
    foreign_system          : Foreign_system_designator,
    foreign_name            : String,
    foreign_parameters      : [ String ]
)
```

(2) CONTENTS_COPY_FROM_FOREIGN_SYSTEM copies the file identified by *foreign_name*, residing on the foreign system *foreign_system*, into the file contents of the object *file*, overwriting any previous contents.

(3) The syntax and interpretation of *foreign_name* and *foreign_parameters*, whether *foreign_parameters* is required, and the interpretation of the process's mandatory and discretionary context and the permissions required on *foreign_name*, are all implementation-defined and may depend on *foreign_system*.

(4) A write lock of the default mode is obtained on *file*. A read lock of the default mode is obtained on *foreign_system*.

Errors

(5) ACCESS_ERRORS (*file*, ATOMIC, MODIFY, WRITE_CONTENTS)

(6) ACCESS_ERRORS (*foreign_system*, ATOMIC, READ, NAVIGATE)

(7) FOREIGN_OBJECT_IS_INACCESSIBLE (*foreign_system*, *foreign_name*)

(8) FOREIGN_SYSTEM_IS_INACCESSIBLE (*foreign_system*)

(9) FOREIGN_SYSTEM_IS_UNKNOWN (*foreign_system*)

(10) STATIC_CONTEXT_IS_IN_USE (*file*)

18.3.2 CONTENTS_COPY_TO_FOREIGN_SYSTEM

(1)
```
CONTENTS_COPY_TO_FOREIGN_SYSTEM (
    file                 : File_designator,
    foreign_system       : Foreign_system_designator,
    foreign_name         : String,
    foreign_parameters   : [ String ]
)
```

(2) CONTENTS_COPY_TO_FOREIGN_SYSTEM copies the file contents of the object *object* into a file identified by *foreign_name* on the foreign system *foreign_system*.

(3) The syntax and interpretation of *foreign_name* and *foreign_parameters*, whether *foreign_parameters* is required, and the interpretation of the process's mandatory and discretionary context and the permissions required on *foreign_name*, are all implementation-defined and may depend on *foreign_system*.

(4) A read lock of the default mode is obtained on *file*. A read lock of the default mode is obtained on *foreign_system*.

Errors

(5) ACCESS_ERRORS (*file*, ATOMIC, READ, READ_CONTENTS)

(6) ACCESS_ERRORS (*foreign_system*, ATOMIC, READ, NAVIGATE)

(7) FOREIGN_OBJECT_IS_INACCESSIBLE (*foreign_system*, *foreign_name*)

(8) FOREIGN_EXECUTION_IMAGE_IS_BEING_EXECUTED (*foreign_system*, *foreign_name*)

(9) FOREIGN_SYSTEM_IS_INACCESSIBLE (*foreign_system*)

(10) FOREIGN_SYSTEM_IS_UNKNOWN (*foreign_system*)

(11) LABEL_IS_OUTSIDE_RANGE (*file, foreign_system)*

(12) NOTE - It is implementation-defined whether the contents of *object* overwrites or is appended to the contents of the foreign file; this may depend on the properties of *foreign_system* and on *foreign_parameters*.

18.4 Time operations

18.4.1 TIME_GET

(1)
```
TIME_GET (
)
    time     : Time
```

(2) TIME_GET returns as *time* the current value of the system time.

Errors

(3) None.

18.4.2 TIME_SET

(1)
```
TIME_SET (
    time     : Time
)
```

(2) TIME_SET sets the value of system time to *time*.

Errors

(3) PRIVILEGE_IS_NOT_GRANTED (PCTE_CONFIGURATION)

(4) The following implementation-defined error may be raised:
TIME_CANNOT_BE_CHANGED

ECMA

114 Rue du Rhône
CH-1204 Geneva
Switzerland

This Standard ECMA-149 is available free of charge in printed form and as a file.

See inside cover page for instructions