Standard ECMA-149
4th Edition - December 1997

ECMA

Standardizing Information and Communication Systems

Portable Common Tool Environment (PCTE) - Abstract Specification

VOLUME 2

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

Standard ECMA-149
4th Edition - December 1997

ECMA

Standardizing Information and Communication Systems

Portable Common Tool Environment (PCTE) - Abstract Specification

VOLUME 2

Phone: +41 22 849.60.00 - **Fax:** +41 22 849.60.01 - **URL:** http://www.ecma.ch - **Internet:** helpdesk@ecma.ch

IW E149-V2.DOC 22-02-99 16,16

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

19 Discretionary security

19.1 Discretionary security concepts

19.1.1 Security groups

(1) Group_identifier = Natural

(2) **sds** discretionary_security:

(3) **import object type** system-object, system-static_context, system-process, system-common_root;

(4) **import attribute type** system-name, system-number;

(5) security_group_directory: **child type of** object **with**
link
known_security_group: (**navigate**) **existence link** (group_identifier: **natural**) **to** security_group;
security_groups_of: **implicit link to** common_root **reverse** security_groups;
end security_group_directory;

(6) security_group: **child type of** object;

(7) user: **child type of** security_group **with**
link
user_identity_of: (**navigate**) **non_duplicated designation link** (number) **to** process;
user_member_of: (**navigate**) **reference link** (number) **to** user_group **reverse** has_users;
end user;

(8) user_group: **child type of** security_group **with**
link
has_users: (**navigate**) **reference link** (number) **to** user **reverse** user_member_of;
user_subgroup_of: (**navigate**) **reference link** (number) **to** user_group **reverse** has_user_subgroups;
has_user_subgroups: (**navigate**) **reference link** (number) **to** user_group **reverse** user_subgroup_of;
adopted_user_group_of: (**navigate**) **non_duplicated designation link** (number) **to** process;
end user_group;

(9) program_group: **child type of** security_group **with**
link
has_programs: (**navigate**) **reference link** (number) **to** static_context **reverse** program_member_of;
program_subgroup_of: (**navigate**) **reference link** (number) **to** program_group **reverse** has_program_subgroups;
has_program_subgroups: (**navigate**) **reference link** (number) **to** program_group **reverse** program_subgroup_of;
end program_group;

(10) **extend object type** static_context **with**
link
program_member_of: (**navigate**) **implicit link** (system_key) **to** program_group **reverse** has_programs;
end static_context;

(11)
```
extend object type process with
link
    user_identity: (navigate) designation link to user;
    adopted_user_group: (navigate) designation link to user_group;
    adoptable_user_group: (navigate) designation link (number) to user_group with
    attribute
        adoptable_for_child: (read) boolean := true;
    end adoptable_user_group;
end process;
```

(12)
```
extend object type common_root with
link
    security_groups: (navigate) existence link to security_group_directory reverse
        security_groups_of;
end common_root;
```

(13)
```
end discretionary_security;
```

(14) The security group directory is an administrative object (see 9.1.2).

(15) A user is a *member* of a user group if there is a "has_users" link from the user group to the user.

(16) A static context is a *member* of a program group if there is a "has_programs" link from the program group to the static context.

(17) A user group A is a *user subgroup* of a user group B if there is a "has_user_subgroups" link from the user group B to the user group A. User group B is a *direct user supergroup* of user group A.

(18) An *indirect user supergroup* of a user group is a direct user supergroup of a direct or indirect user supergroup of the user group. A *user supergroup* of a user group is a direct user supergroup or an indirect user supergroup of that user group.

(19) The set of user groups with the user-subgroup/user-supergroup relation forms an acyclic graph with the predefined user group ALL_USERS as root.

(20) A program group consists of a set of static contexts. A program group A is a *program subgroup* of a program group B if there is a "has_program_subgroups" link from the program group B to the program group A. Program group B is a *direct program supergroup* of program group A.

(21) An *indirect program supergroup* of a program group is a direct program supergroup of a direct or indirect program supergroup of the program group. A *program supergroup* of a program group is a direct program supergroup or an indirect program supergroup of that program group.

(22) Where there is no risk of ambiguity, a user subgroup or a program subgroup is called simply a *subgroup*, and a user supergroup or a program supergroup is called simply a *supergroup*.

(23) *Discretionary groups* are security groups used for the purposes of discretionary access control. Each process has the following *effective security groups*:

(24) - One user, the destination of the "user_identity" link from the process, called *the user* of the process.

(25) - One user group, the *adopted user group* of the process, of which the user is a member, and all user supergroups of that user group (including the group ALL_USERS). The adopted user group is the destination of the "adopted_user_group" link from the process.

(26) - All program groups of which a non-interpretable static context run by a process (see 13.1.1) is a member, and all their supergroups; and for an interpretable static context, the program groups of which the interpreter is a member, and all their supergroups.

(27) Each process also has an associated set of user groups called its *adoptable* user groups which are the destination of "adoptable_user_group" links from the process; these are the set of user groups out of which the process may make effective one user group in place of the currently adopted user group. Adoptable user groups must have the user as a member.

(28) When a process creates a child process, its adoptable user groups are inherited except when the "adoptable_for_child" attribute of the "adoptable_user_group" link from the parent process is **false**.

(29) No object type is a descendant type of more than one of the object types "user", "user_group", and "program_group".

(30) The predefined user group ALL_USERS always exists, as do the predefined program groups PCTE_SECURITY, PCTE_AUDIT, PCTE_EXECUTION, PCTE_REPLICATION, PCTE_CONFIGURATION, PCTE_HISTORY, and PCTE_SCHEMA_UPDATE, which are objects in the initial state of the object base linked to the security group directory with predefined values of their group identifiers. Their security group identifiers are as follows:

- ALL_USERS 1
- PCTE_SECURITY 2
- PCTE_AUDIT 3
- PCTE_EXECUTION 4
- PCTE_REPLICATION 5
- PCTE_CONFIGURATION 6
- PCTE_HISTORY 7
- PCTE_SCHEMA_UPDATE 8

(31) Zero is not used as a security group identifier; it is used to denote absence of a security group.

(32) A user must be a member of a user group in order for a process to act on its behalf.

NOTES

(33) 1 Discretionary access to objects is controlled on the basis of the effective security groups of the accessing process. Security groups are of three types: users, user groups and program groups. Each process has one group which represents the user on behalf of whom the process is running. A user may play several different roles while using the PCTE-based environment, and these roles are represented by the user groups to which the user belongs. The role the user is playing at any one time is given by the user group which is currently adopted plus its supergroups recursively. It is an important security requirement that a user adopts at most one role before operations are carried out on its behalf. The subgroup structure is intended to reflect the organization of the project into working groups or teams and team membership.

(34) 2 Rights may also be granted to a program, which the user also obtains when the program executes on the user's behalf provided that the user has the right to execute the program. Program groups may be used to deny as well as to grant access to specific data objects. In this way program groups may be used to model data abstraction and implement information hiding. They also provide a less specific way of restricting access. A process may only act on behalf of a single user and user group at any one time and which must be authenticated. Giving a right to a program means that the right is given to any user who has the right to execute the program when the program is executed on behalf of that user. Program groups also provide a means of expanding the number of effective security groups without violating the constraint of there being only one user role effective at any one time.

(35) 3 A user which is a member of a user group need not be a member of a sub- or supergroup of that group.

(36) 4 The security group structure is intended to be used by tools, such as "login" tools, built on top of PCTE.

(37) 5 The predefined user group ALL_USERS is effective for all processes, as it is the root of the directed acyclic graph of user groups. Access rights which are effective for all users can be given to this user group.

(38) 6 A process may have no effective program group.

(39) 7 The predefined program groups have the following meanings:

(40) - PCTE_AUDIT This program group is required by the following operations for the audit mechanism:
  . AUDIT_SWITCH_ON_SELECTION;
  . AUDIT_SWITCH_OFF_SELECTION;
  . AUDIT_ADD_CRITERION;
  . AUDIT_REMOVE_CRITERION;
  . AUDIT_GET_CRITERIA;
  . AUDIT_SELECTION_CLEAR;
  . AUDITING_STATUS;
  . AUDIT_FILE_COPY_AND_RESET.

(41) - PCTE_CONFIGURATION This program group is required when type identifiers are used to denote invisible types in type references (see 23.1.2.5), and by the following operations which define devices or volumes or which manage workstations or archives:
  . ARCHIVE_RESTORE;
  . ARCHIVE_SAVE;
  . DEVICE_CREATE;
  . DEVICE_REMOVE;
  . VOLUME_CREATE;
  . VOLUME_DELETE;
  . WORKSTATION_REDUCE_CONNECTION;
  . WORKSTATION_CREATE;
  . WORKSTATION_CREATE;
  . WORKSTATION_CONNECT;
  . WORKSTATION_DELETE;

(42) - PCTE_EXECUTION This program group may be required by the following operations for execution mechanisms such as setting the file size limit for a process or changing the priority of a process:
  . PROCESS_SET_FILE_SIZE_LIMIT;
  . PROCESS_INTERRUPT_OPERATION;
  . PROCESS_SET_PRIORITY;
  . TIME_SET.

(43) - PCTE_HISTORY. This program group is required by the following operations to explicitly set the last access time or last modification time of an object, or to manipulate the version graph:
  . OBJECT_SET_TIME_ATTRIBUTES;
  . VERSION_ADD_PREDECESSOR;
  . VERSION_REMOVE;
  . VERSION_REMOVE_PREDECESSOR.

(44) - PCTE_REPLICATION This program group is required by the operations of the replication clause and all the operations which modify the object base when they apply to masters of replicated objects. These are a very large subset of all PCTE operations (see C.3). They are not listed here.

(45) - PCTE_SECURITY This program group is required to use the operations which are critical to either the consistency of the security group structure or to security (or both). These are the three operations:

. GROUP_REMOVE;

. GROUP_RESTORE;

. PROCESS_SET_USER.

(46) - PCTE_SCHEMA_UPDATE. This program group is required by operations which update an SDS, i.e. those defined in 10.2.

19.1.2 Access control lists

(1)
```
Discretionary_access_mode = APPEND_CONTENTS | APPEND_IMPLICIT | APPEND_LINKS |
    CONTROL_DISCRETIONARY | CONTROL_MANDATORY | CONTROL_OBJECT |
    DELETE | EXECUTE | EXPLOIT_CONSUMER_IDENTITY | EXPLOIT_DEVICE |
    EXPLOIT_SCHEMA | NAVIGATE | OWNER | READ_ATTRIBUTES | READ_CONTENTS |
    READ_LINKS | STABILIZE | WRITE_ATTRIBUTES | WRITE_CONTENTS |
    WRITE_IMPLICIT | WRITE_LINKS
```

(2)
```
Discretionary_access_mode_value = GRANTED | UNDEFINED | DENIED | PARTIALLY_DENIED
```

(3)
```
Discretionary_access_modes = set of Discretionary_access_mode
```

(4)
```
Access_rights = map Discretionary_access_mode to Discretionary_access_mode_value
```

(5)
```
Acl = map Group_identifier to Access_rights
```

(6)
```
Atomic_discretionary_access_mode_value = GRANTED | UNDEFINED | DENIED
```

(7)
```
Atomic_access_rights = map Discretionary_access_mode to
    Atomic_discretionary_access_mode_value
```

(8)
```
sds discretionary_security:
```

(9)
```
import object type system-object, system-process;
```

(10)
```
extend object type object with
attribute
    atomic_acl: (protected) non_duplicated string;
    composite_acl: (protected) non_duplicated string;
end object;
```

(11)
```
extend object type process with
attribute
    default_atomic_acl: (protected) string;
    default_object_owner: (protected) natural;
end process;
```

(12)
```
end discretionary_security;
```

(13) Each object has two associated *access control lists* (or *ACLs*): an *atomic ACL* and a *composite ACL*. They are represented by two string attributes, "atomic_acl" and "composite_acl" respectively. The *scope* of an ACL is the set of atomic objects to which it governs access: the scope of the atomic ACL of an object is the atomic object associated with the object; the scope of the composite ACL is the atoms of the object.

(14) Each ACL is a set of ACL entries. Each ACL entry gives the discretionary access mode value of each discretionary access mode for one security group.

(15) In an atomic ACL, the possible discretionary mode values are GRANTED, DENIED, and UNDEFINED. In an composite ACL they are GRANTED, DENIED, UNDEFINED, and PARTIALLY_DENIED.

(16) *Access right evaluation for a group* is defined by the function

```
EVALUATE_GROUP (
    g   : Security_group_designator;
    o   : Object_designator;
    s   : Object_scope;
    m   : Discretionary_access_mode
)
    v   : Discretionary_access_mode_value
```

where *v* is the discretionary access mode value of *m* in the ACL entry for *g* in the atomic ACL (if *s* is ATOMIC) or the composite ACL (if *s* is COMPOSITE) for *o*. The group *g* is said to have the discretionary access mode *m* atomically *granted*, *denied*, or *undefined* to the object *o*, if *s* is ATOMIC and *v* is GRANTED, DENIED, or UNDEFINED respectively; and compositely *granted*, *denied*, *undefined*, or *partially denied* if *s* is COMPOSITE and *v* is GRANTED, DENIED, UNDEFINED, or PARTIALLY_DENIED respectively. *v* is called the *atomic* or *composite m value* for *g* to *o*. If *v* is GRANTED, *g* is said to have the *atomic* or *composite m discretionary access right* to *o*.

(17) For every object *o* there is at least one security group *g* for which EVALUATE_GROUP (*g*, *o*, ATOMIC, CONTROL_DISCRETIONARY) = GRANTED and at least once security group *g'* for which EVALUATE_GROUP(*g'*, *o*, ATOMIC, CONTROL_MANDATORY) = GRANTED.

(18) For every object *o* there is at least one security group which has the atomic CONTROL_DISCRETIONARY right to *o*, and at least once security group which has the atomic CONTROL_MANDATORY right to *o*.

(19) The following constraints apply to the composite ACL for an object *o* and the atomic ACLs of *o* and its components, for any security group *g* and for any discretionary access mode *m* except OWNER and CONTROL_DISCRETIONARY:

(20) EVALUATE_GROUP (*g*, *o*, COMPOSITE, *m*) = GRANTED if and only if EVALUATE_GROUP (*g*, *o*, ATOMIC, *m*) = GRANTED and EVALUATE_GROUP (*g*, *c*, ATOMIC, *m*) = GRANTED for every component *c* of *o*.

(21) EVALUATE_GROUP (*g*, *o*, COMPOSITE, *m*) = DENIED if and only if EVALUATE_GROUP (*g*, *o*, ATOMIC, *m*) = DENIED and EVALUATE_GROUP (*g*, *c*, ATOMIC, *m*) = DENIED for every component *c* of *o*.

(22) EVALUATE_GROUP (*g*, *o*, COMPOSITE, *m*) = PARTIALLY_DENIED if an only if EVALUATE_GROUP (*g*, *o*, ATOMIC, *m*) = DENIED or EVALUATE_GROUP (*g*, *c*, ATOMIC, *m*) = DENIED for some component *c* of *o*, and EVALUATE_GROUP (*g*, *o*, ATOMIC, *m*) ≠ DENIED or EVALUATE_GROUP (*g*, *c*, ATOMIC, *m*) ≠ DENIED for some component *c* of *o*.

(23) EVALUATE_GROUP (*g*, *o*, COMPOSITE, *m*) = UNDEFINED in all other cases.

(24) The following constraints apply to the composite ACL of an object *o* and the atomic ACLs of *o* and its components, for any security group *g* and for the discretionary access modes OWNER and CONTROL_DISCRETIONARY.

(25) - If EVALUATE_GROUP (*g*, *o*, COMPOSITE, OWNER) = GRANTED then EVALUATE_GROUP (*g*, *c*, COMPOSITE, OWNER) = GRANTED for every component *c* of *o*, EVALUATE_GROUP (*g*, *o*, ATOMIC, CONTROL_DISCRETIONARY) = GRANTED, and EVALUATE_GROUP (*g*, *c*, ATOMIC, CONTROL_DISCRETIONARY) = GRANTED for every component *c* of *o*.

(26) - If EVALUATE_GROUP (*g*, *o*, COMPOSITE, OWNER) = DENIED then EVALUATE_GROUP (*g*, *c*, COMPOSITE, OWNER) = DENIED for every component *c* of *o*, EVALUATE_GROUP (*g*, *o*, ATOMIC, CONTROL_DISCRETIONARY) = DENIED and EVALUATE_GROUP (*g*, *c*, ATOMIC, CONTROL_DISCRETIONARY) = DENIED for every component *c* of *o*.

(27) *Access right evaluation for a process* is defined by the function

```
EVALUATE_PROCESS (
    p   : Process_designator;
    o   : Object_designator;
    s   : Object_scope;
    m   : Discretionary_access_mode
)
    a   : Boolean
```

(28) The returned value *a* is defined from the ACLs of *o* in the following way: EVALUATE_PROCESS (*p*, *o*, *s*, *m*) = **true** if and only if there is at least one effective group *g* of *p* for which EVALUATE_GROUP (*g*, *o*, *s*, *m*) = GRANTED, and for every other effective group *g'* of *p* EVALUATE_GROUP (*g'*, *o*, *s*, *m*) = GRANTED or EVALUATE_GROUP (*g'*, *o*, *s*, *m*) = UNDEFINED. If *a* = **true**, *p* is said to have the *atomic* or *composite m discretionary access right* to *o*, according as *s* is ATOMIC or COMPOSITE respectively.

(29) The default atomic ACL and default object owner are used to determine the atomic ACLs and composite ACLs of objects created by the process (see 19.1.4).

NOTES

(30) 1 A composite ACL is computable from the atomic ACLs of the object and its components, except for the discretionary access mode OWNER.

(31) 2 The implementation-defined mapping of access control lists to the string attribute values may economize on space by omitting discretionary access modes with value UNDEFINED, and omitting ACL entries with all values UNDEFINED.

(32) 3 If OWNER is set to UNDEFINED for an object *o* and a group *g*, the OWNER values for *g* to the components of *o*, and the CONTROL_DISCRETIONARY values for *g* to *o* and its components, are unchanged.

19.1.3 Discretionary access modes

(1) The following list describes the meanings of the discretionary access modes, generally in terms of the classes of operations for which they are *needed atomically* or *compositely* on an object *o*, i.e. for which a necessary precondition is that the calling process has the atomic or composite access right *m*, respectively, to *o*. The exact definitions are given by the occurrences of DISCRETIONARY_ACCESS_IS_NOT_GRANTED in the operation descriptions; see 19.2.

(2) - APPEND_CONTENTS. Needed atomically to append to the contents of an object or to send a message to a message queue.

(3) - APPEND_IMPLICIT. Needed atomically to create new implicit links of an object.

(4) - APPEND_LINKS. Needed atomically to create new links, other than implicit links, from an object. (This right is not sufficient to write the non-key attributes of such a link.)

(5) - CONTROL_DISCRETIONARY. Needed atomically on an object to change its atomic ACL (except CONTROL_MANDATORY). CONTROL_DISCRETIONARY occurs only in atomic ACLs.

(6) - CONTROL_MANDATORY. Needed atomically on an object to change its "confidentiality_label" and "integrity_label" attributes and to change the CONTROL_MANDATORY rights of other groups on that object.

(7) - CONTROL_OBJECT. Needed atomically on an object to convert it to a descendant type, to move it to another volume, to create or delete "predecessor" and "successor" links to and from the object, and to convert a normal object to a master object or vice versa,. Needed on a message queue to change the number of storage units allowed by the message queue.

(8) - DELETE. Needed compositely on an object to delete the object (i.e. to delete the last composition or existence link to the object). DELETE has no effect in an atomic ACL.

(9) - EXECUTE. Needed atomically on a static context to execute the associated program. EXECUTE has no effect on objects of other types.

(10) - EXPLOIT_CONSUMER_IDENTITY. Needed atomically on a consumer group to use it as a consumer identity. EXPLOIT_CONSUMER_IDENTITY has no effect on objects of other types.

(11) - EXPLOIT_DEVICE. Needed atomically on a device supporting volume to mount a volume on the device or to unmount a volume from the device. EXPLOIT_DEVICE has no effect on objects of other types.

(12) - EXPLOIT_SCHEMA. Needed atomically on an SDS, a type in SDS, or a type to use it in a working schema or to consult the typing information contained in it. EXPLOIT_SCHEMA has no effect on objects of other types.

(13) - NAVIGATE. Needed atomically on an object to use a link reference of a link of the object in a pathname (see 23.1.2.2); needed atomically on a foreign system to access a file on that foreign system.

(14) - OWNER. OWNER occurs only in composite ACLs. It is needed to modify the composite ACL of an object, except for implicit modification by modification of the atomic ACL of the object or of a component of the object. This modification right includes the OWNER right for any security group on that object, except that CONTROL_DISCRETIONARY rights may apply (see below) if there is no owner of the object. Unlike other discretionary access modes in composite ACLs, modification of OWNER values is not automatic and must be done explicitly using OBJECT_SET_ACL_ENTRY.

(15) An object must always have an atomic ACL such that it is possible that a process could exist with a set of effective groups such that the process has the CONTROL_DISCRETIONARY discretionary access right to that object and that another or the same process could exist with a set of effective groups such that the process has the CONTROL_MANDATORY discretionary access right to that object.

(16) An *owner* of an object is a security group with OWNER right to the object. There may be more than one owner of an object.

(17) For changing OWNER discretionary access values the following rules hold:

(18) . An owner may modify the OWNER discretionary access value to an object for itself, and for another security group except when the other group is also an owner of the object.

(19) . An owner of an object may modify the OWNER discretionary access value for any group to any component of the object.

(20) . The OWNER discretionary access value for a group to an object may not be modified if that object is a component of an object to which that group has OWNER granted or denied.

(21) . If no owner for an object exists, then the OWNER discretionary access value may be modified if OWNER is granted for a group to all components of the object (excluding the object, if it is a component of itself) and CONTROL_DISCRETIONARY is granted for the group to the object and to all its components.

(22) . The constraints defined in 19.1.2 must be maintained.

(23) OWNER when used in connection with discretionary security does not have a meaning in the accounting sense.

(24) - READ_ATTRIBUTES. Needed atomically to read the attribute values of an object and to evaluate a link of an object if the evaluation uses the preferred link type and preferred link key of the object (see 23.1.2.5) For some predefined attributes, e.g. "atomic_acl", the READ_ATTRIBUTES right is not needed, if the attribute is retrieved by an operation especially defined to retrieve that attribute.

(25) The READ_ATTRIBUTES right is not needed to read the attribute values of the links of an object.

(26) - READ_CONTENTS. Needed atomically to read the contents of an object, to save a message queue or a process address space, to save a message queue, or to peek a message in a message queue.

(27) - READ_LINKS. Needed atomically to read the attributes of the links of an object, or to scan sets of links of an object.

(28) - STABILIZE. Needed to change the stability of an object, i.e. to create or delete a stabilizing link to it or a compositely stabilizing link to it or to an object of which it is a component.

(29) - WRITE_ATTRIBUTES. Needed atomically to change the attribute values of an object. It does not control changing the attribute values of the links of an object, nor the time attributes of an object.

(30) - WRITE_CONTENTS. Needed atomically to write to or update the contents of an object or a process address space, to set or remove a breakpoint in a process, to restore a message queue, or to receive or delete a message from a message queue. An object's contents may not be deleted although it may be emptied.

(31) - WRITE_IMPLICIT. Needed atomically to delete implicit links of an object. For this category of link, there are no attributes to change.

(32) - WRITE_LINKS. Needed atomically to delete links, other than implicit links, of an object and to change values of link attributes.

(33) Where EXPLOIT_SCHEMA, EXPLOIT_DEVICE, EXPLOIT_CONSUMER_IDENTITY, CONTROL_OBJECT, CONTROL_DISCRETIONARY, CONTROL_MANDATORY or OWNER discretionary access rights to an object are required of the calling process by an operation which changes the links or attributes of that object, discretionary access rights which would be appropriate for such changes (e.g. APPEND_LINKS, WRITE_ATTRIBUTES) are not also required to that object.

NOTES

(34) 1 OWNER consistency rules demand that an owner may not modify the OWNER discretionary access right to an object for another security group not only when the other security group is an owner of the object, but also when the other security group is the owner of an object of which the object is a component.

(35) 2 The rules for conferring the OWNER discretionary access right on an object for which no owner exists also cover the case where no owner exists for an object of which the object is a component, since if such an owner existed, an owner would exist for the object under consideration.

(36) 3 The OWNER right on an object for a security group can never be PARTIALLY_DENIED. This is achieved by ensuring that when a composition link is created (e.g. by OBJECT_CREATE, SDS_CREATE_OBJECT_TYPE, or LINK_RESTORE) any OWNER rights of the newly enclosing object are propagated to the new component, and that when the OWNER right is set on an object (by OBJECT_SET_ACL_ENTRY) the new value is consistent with rights on enclosing objects.

19.1.4 Access control lists on object creation

(1) When an object is created, its atomic ACL is determined from the default atomic ACL of the creating process as follows. For each ACL entry in the default atomic ACL, with access rights M, an ACL entry in the atomic ACL is created for the same group with access rights M', where the mapping M' is determined for each discretionary access mode *m* by the corresponding discretionary mode values of M and the access mask A:

(2) - M'(*m*) = GRANTED if M(*m*) = GRANTED or A(*m*) = GRANTED, and neither M(*m*) = DENIED nor A(*m*) = DENIED.

(3) - M'(*m*) = DENIED if M(*m*) = DENIED or A(*m*) = DENIED.

(4) - M'(*m*) = UNDEFINED if M(*m*) = UNDEFINED and A(*m*) = UNDEFINED.

(5) The access mask A is a parameter to the operation used to create the object (e.g. OBJECT_CREATE).

(6) The default object owner of the creating process defines the group identifier of a security group. The composite ACL of the created object is derived from the atomic ACLs of the object subject to the constraints given in 19.1.2 for all discretionary access modes except OWNER and CONTROL_DISCRETIONARY. An entry in the composite ACL relates to the default object owner of the creating process, if one exists, and has the OWNER discretionary access mode granted. It is an error if the ACL entry in the created atomic ACL for the default object owner group does not have CONTROL_DISCRETIONARY granted.

(7) When an operation creates an object, any further accesses to that object during that same operation call are not subject to discretionary or mandatory access checks.

19.2 Operations for discretionary access control operation

19.2.1 GROUP_GET_IDENTIFIER

(1)
```
GROUP_GET_IDENTIFIER (
    group       : Security_group_designator
)
    identifier  : Group_identifier
```

(2) GROUP_GET_IDENTIFIER returns in *identifier* the key of the "known_security_group" link from the security group directory to the security group *group*.

Errors

(3) ACCESS_ERRORS (security group determined by *group*, ATOMIC, READ, READ_LINKS)

(4) GROUP_IDENTIFIER_IS_INVALID (*group*)

19.2.2 OBJECT_CHECK_PERMISSION

(1)
```
OBJECT_CHECK_PERMISSION (
    object      : Object_designator,
    modes       : Discretionary_access_modes,
    scope       : Object_scope
)
    accessible  : Boolean
```

(2) OBJECT_CHECK_PERMISSION tests if the calling process has the discretionary and mandatory permission to access the object *object* according to the set of access modes given in *modes* and the scope *scope*. For the discretionary permissions, the operation evaluates EVALUATE_PROCESS (calling process, *object*, *scope*, *mode*) (see 19.1.2) for each discretionary access mode *mode* in *modes*. For the mandatory permissions read and write access is interpreted according to the discretionary access modes:

(3) - Read access is tested if *modes* contains NAVIGATE, READ_ATTRIBUTES, READ_LINKS, READ_CONTENTS, EXECUTE, EXPLOIT_DEVICE, EXPLOIT_SCHEMA or EXPLOIT_CONSUMER_IDENTITY.

(4) - Write access is tested if *modes* contains APPEND_CONTENTS, APPEND_LINKS, APPEND_IMPLICIT, WRITE_ATTRIBUTES, WRITE_CONTENTS, WRITE_LINKS, WRITE_IMPLICIT, DELETE, CONTROL_DISCRETIONARY, CONTROL_ MANDATORY, CONTROL_OBJECT or OWNER.

(5) Testing for mandatory read access permission means checking for confidentiality violation and integrity confinement violation (see 20.1). Testing for mandatory write access permission means checking for confidentiality confinement violation and integrity violation. These checks are defined in terms of label domination between the mandatory labels of *object* and the mandatory context of the process.

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

(7) The return value *accessible* is:

(8) - **false** if for at least one of the discretionary access modes given in *modes* EVALUATE_PROCESS (calling process, *object*, *scope*, *mode*) = **false**.

(9) - **false** if read access is implied by modes and either LABEL_DOMINATES (*confidentiality_context* (*process*), *confidentiality_label* (*object*)) = **false** or LABEL_DOMINATES (*integrity_label* (*object*), *integrity_context* (*process*) = **false** (see 20.1.3).

(10) - **false** if write access is implied by *modes* and either LABEL_DOMINATES (*confidentiality_label* (*object*), *confidentiality_context*(*process*) = **false** or LABEL_DOMINATES (*integrity_context*(*process)*, *integrity_label* (*object*)) = **false**

(11) - **true** otherwise. In this case, for all of the discretionary access modes given in *modes* EVALUATE_PROCESS (calling process, *object*, *scope*, *mode*) = **true**; and:

(12) . if read access is implied by *modes* then LABEL_DOMINATES (*confidentiality_context* (*process*), *confidentiality_label* (*object*)) = **true** and LABEL_DOMINATES (*integrity_label* (*object*), *integrity_context* (*process*)) = **true**;

(13) . if write access is implied by *modes* then LABEL_DOMINATES (*confidentiality_label* (*object*), *confidentiality_context*(*process*)) = **true** and LABEL_DOMINATES (*integrity_context* (*process*), *integrity_label* (*object*)) = **true**.

(14) For the maps confidentiality_label, confidentiality_context, integrity_label, and integrity_context see 20.1.4.

Errors

(15) ACCESS_MODE_IS_INCOMPATIBLE (*scope*, *modes*)

(16) CONFIDENTIALITY_WOULD_BE_VIOLATED (*granule*, *scope*)

(17) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (*granule*, *scope*)

(18) OBJECT_IS_ARCHIVED (*granule*)

(19) OBJECT_IS_INACCESSIBLE (*granule*, *scope*)

(20) NOTE - READ_ATTRIBUTES access right is not necessary to perform this operation. If it were, the operation would lose much of its usefulness, since access checks do not require any access permissions to read mandatory labels or ACLs.

19.2.3 OBJECT_GET_ACL

(1)
```
OBJECT_GET_ACL (
    object      : Object_designator,
    scope       : Object_scope
)
    acl         : Acl
```

(2) OBJECT_GET_ACL returns the atomic or composite ACL of the object *object*, according as *scope* is ATOMIC or COMPOSITE.

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

Errors

(4) ACCESS_ERRORS (*granule*, *scope*, READ, READ_ATTRIBUTES)

(5) NOTE - It is expected that implementations calculate the composite ACL of an object (except for the OWNER modes) from the atomic ACLs of the object and its components.

19.2.4 OBJECT_SET_ACL_ENTRY

(1)
```
OBJECT_SET_ACL_ENTRY (
    object      : Object_designator,
    group       : Group_identifier,
    modes       : Atomic_access_rights,
    scope       : Object_scope
)
```

(2) OBJECT_SET_ACL_ENTRY sets an ACL entry in the atomic or composite ACL of the object *object* for the security group *group*. If the settings of the ACL entry are already as required,

except for setting a composite ACL entry to UNDEFINED, then this operation has no effect. In the case where for *scope* = COMPOSITE, and some mode *m*, *modes*(*m*) is UNDEFINED and EVALUATE_GROUP (*group*, *object*, COMPOSITE, *m*) is previously UNDEFINED, then there is an effect if for one or more components *c* of *object*, EVALUATE_GROUP (*group*, *c*, ATOMIC, *m*) is not already set UNDEFINED. EVALUATE_GROUP (*group*, *c*, ATOMIC, *m*) of such components is changed to UNDEFINED.

(3) If *scope* is ATOMIC, then OBJECT_SET_ACL_ENTRY sets the ACL entry for *group* in the atomic ACL of *object* to *modes*, for all discretionary access modes specified in *modes*. OWNER must not appear in *modes*.

(4) If *scope* is COMPOSITE, then for *object* and all its components, OBJECT_SET_ACL_ENTRY sets the ACL entries for *group* in the composite ACLs and also in the atomic ACLs for all discretionary access modes specified in *modes*, except CONTROL_DISCRETIONARY and OWNER, to *modes*. CONTROL_DISCRETIONARY must not appear in *modes*. OWNER is treated as follows:

(5) - the OWNER discretionary access mode value for *group* in the composite ACL of *object* and the CONTROL_DISCRETIONARY discretionary access mode value for *group* in the atomic ACL of *object* are set to *modes* (OWNER) provided that any outer object of *object* has OWNER undefined for *group*.

(6) - If *modes* (OWNER) = UNDEFINED, then in the components of *object* the discretionary access mode values for OWNER in the composite ACL and the discretionary access mode values for CONTROL_DISCRETIONARY in the atomic ACL of *group* are not changed.

(7) - If *modes* (OWNER) = GRANTED or DENIED, then in the components of *object group* is set to have OWNER granted or denied respectively in the composite ACL, and CONTROL_DISCRETIONARY granted or denied respectively in the atomic ACL; provided that any outer object of *object* has OWNER undefined for *group*.

(8) - If no owner for *object* exists, then the operation can modify OWNER, and OWNER only, if CONTROL_DISCRETIONARY is granted for *group* in the atomic ACL of *object*, and for all components of *object*, OWNER is granted for *group* in the composite ACL.

(9) Whether *scope* is ATOMIC or COMPOSITE, the composite ACLs of all outer objects of *object*, and of *object* itself if *scope* is ATOMIC, are updated, so that all constraints defined for composite ACLs, and the atomic and composite ACLs of their components (see 19.1.2) are maintained. OWNER modes of outer objects of *object* are not updated; if this would be necessary to maintain the constraints then an error is raised.

(10) If *scope* is COMPOSITE, then write locks of the default mode are obtained on *object* and on all its components.

Errors

(11) If *scope* is ATOMIC:
ACCESS_ERRORS (*object*, ATOMIC, CHANGE, CONTROL_DISCRETIONARY)

(12) If *scope* is COMPOSITE:

(13) If there is no owner for *object*, and only OWNER is to be modified:
ACCESS_ERRORS (*object*, ATOMIC, CHANGE, CONTROL_DISCRETIONARY)
ACCESS_ERRORS (component of *object*, COMPOSITE, CHANGE, OWNER)

(14) If there is an owner for *object*, or there is no owner for *object* and modes other than OWNER are required to be modified:
ACCESS_ERRORS (*object*, COMPOSITE, CHANGE, OWNER)

(15) If the CONTROL_MANDATORY access right is changed:
ACCESS_ERRORS (*object*, *scope*, CHANGE, CONTROL_MANDATORY)

(16) If *scope* is COMPOSITE and *modes* (CONTROL_MANDATORY) = UNDEFINED but no change to the composite ACL is required, then for each atom A of *object* where CONTROL_MANDATORY is to be changed from GRANTED:
ACCESS_ERRORS (A, ATOMIC, CHANGE, CONTROL_MANDATORY)

(17) ACCESS_MODE_IS_NOT_ALLOWED (*modes*, *scope*)

(18) CONTROL_WOULD_NOT_BE_GRANTED (*object*)

(19) GROUP_IDENTIFIER_IS_INVALID (*group*)

(20) If *scope* is COMPOSITE:
OBJECT_HAS_GROUP_WHICH_IS_ALREADY_OWNER (*object*, *group*)

(21) OBJECT_OWNER_CONSTRAINT_WOULD_BE_VIOLATED (*object*)

(22) The following implementation-dependent error may be raised:
OBJECT_IS_INACCESSIBLE (outer object of *object*, ATOMIC)

NOTES

(23) 1 If an implementation calculates the composite ACL when retrieving it, it may be so designed that it requires the outer objects to be accessible.

(24) 2 CONTROL_DISCRETIONARY rather than READ_ATTRIBUTES or WRITE_ATTRIBUTES discretionary access right is required to perform this operation. It would be superfluous to require both.

(25) 3 If *object* is an SDS which may be in use in a working schema, then any change to its composite ACL only has effect when the SDS is next included in a working schema by PROCESS_SET_WORKING_SCHEMA, PROCESS_CREATE_AND_START, or PROCESS_START.

19.3 Discretionary security administration operations

19.3.1 GROUP_INITIALIZE

(1)
```
GROUP_INITIALIZE (
    group       : User_designator | User_group_designator | Program_group_designator
)
    identifier  : Group_identifier
```

(2) GROUP_INITIALIZE adds the security group *group* to the security group directory. A "known_security_group" link is created from the master of the security group directory to *group*. The key of this link is set to a system-generated unique value, which is guaranteed never to be re-used as a security group key and is returned as *identifier*.

(3) Write locks of the default mode are obtained on the created links.

Errors

(4) ACCESS_ERRORS (the security group directory, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*group*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(6) SECURITY_GROUP_IS_KNOWN (*group*)

NOTES

(7) 1 The group identifier, which is the same as the key to the "known_security_group" link to the object, may be determined using the GROUP_GET_IDENTIFIER operation.

(8) 2 This operation does not change any copies of the security group directory.

19.3.2 GROUP_REMOVE

(1)
```
GROUP_REMOVE (
    group   : User_designator | User_group_designator | Program_group_designator
)
```

(2) GROUP_REMOVE removes the security group *group* from the set of known groups. The "known_security_group" link from the security group directory is deleted. If there are no remaining existence or composition links to *group*, then *group* is also deleted. In this case, the "object_on_volume" link to *group* is deleted.

(3) The master of the security group directory is always updated by this operation.

(4) Write locks of the default mode are obtained on the deleted links and object.

Errors

(5) ACCESS_ERRORS (the security group directory, ATOMIC, MODIFY, WRITE_LINKS)

(6) ACCESS_ERRORS(*group*, ATOMIC, MODIFY, WRITE_IMPLICIT)

(7) If the conditions hold for deletion of the "security_group" object *group*:
ACCESS_ERRORS (*group*, COMPOSITE, MODIFY, DELETE)

(8) GROUP_IDENTIFIER_IS_INVALID (*group*)

(9) OBJECT_HAS_LINKS_PREVENTING_DELETION (*group*)

(10) OBJECT_IS_IN_USE_FOR_DELETE (*group*)

(11) PRIVILEGE_IS_NOT_GRANTED (PCTE_SECURITY)

(12) SECURITY_GROUP_IS_IN_USE (*group*)

(13) SECURITY_GROUP_IS_PREDEFINED (*group*)

(14) SECURITY_GROUP_IS_REQUIRED_BY_OTHER_GROUPS (*group*)

(15) NOTE - This operation does not change any copies of the security group directory.

19.3.3 GROUP_RESTORE

(1)
```
GROUP_RESTORE (
    group       : User_designator | User_group_designator | Program_group_designator
    identifier  : Group_identifier
)
```

(2) GROUP_RESTORE adds the security group *group* to the security group directory. A "known_security_group" link is created from the master of the security group directory to *group*. The group identifier *identifier* is used as the key for this link. This identifier must be a used group identifier, originally generated when initializing a security group which has since been deleted.

(3) Write locks of the default mode are obtained on the created links.

Errors

(4) ACCESS_ERRORS (the security group directory, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*group*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(6) GROUP_IDENTIFIER_IS_IN_USE (*identifier*)

(7) GROUP_IDENTIFIER_IS_INVALID (*identifier*)

(8) PRIVILEGE_IS_NOT_GRANTED (PCTE_SECURITY)

(9) SECURITY_GROUP_IS_KNOWN (*group*)

(10) NOTE - This operation does not change any copies of the security group directory.

19.3.4 PROGRAM_GROUP_ADD_MEMBER

(1)
```
PROGRAM_GROUP_ADD_MEMBER (
    group     : Program_group_designator,
    program   : Static_context_designator
)
```

(2) PROGRAM_GROUP_ADD_MEMBER adds the program *program* to the program group *group*. A "program_member_of" link is created from *program* to *group*, together with a "has_programs" reverse link. The keys of the created links are implementation-dependent.

(3) Write locks of the default mode are obtained on the created links.

Errors

(4) ACCESS_ERRORS (*group*, ATOMIC,APPEND_LINKS)

(5) ACCESS_ERRORS (*program*, ATOMIC, MODIFY, APPEND_IMPLICIT)

(6) SECURITY_GROUP_IS_UNKNOWN (*group*)

(7) STATIC_CONTEXT_IS_ALREADY_MEMBER (*program*, *group*)

(8) NOTE - Processes which are current executions of *program* do not receive *group* as an addition to their set of effective security groups.

19.3.5 PROGRAM_GROUP_ADD_SUBGROUP

(1)
```
PROGRAM_GROUP_ADD_SUBGROUP (
    group     : Program_group_designator,
    subgroup  : Program_group_designator
)
```

(2) PROGRAM_GROUP_ADD_SUBGROUP adds the program group *subgroup* to the program group *group*. A "program_subgroup_of" link is created from *subgroup* to *group*, together with a "has_program_subgroups" reverse link. The keys of the created links are implementation-dependent (see 23.1.2.5).

(3) Write locks of the default mode are obtained on the created links.

Errors

(4) ACCESS_ERRORS (*group*, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*subgroup*, ATOMIC, MODIFY, APPEND_LINKS)

(6) MASTER_IS_INACCESSIBLE (some object of the graph of security groups, ATOMIC)

(7) SECURITY_GROUP_ALREADY_HAS_THIS_SUBGROUP (*subgroup*, *group*)

(8) SECURITY_GROUP_IS_IN_USE (*subgroup*)

(9) SECURITY_GROUP_IS_UNKNOWN (*group*)

(10) SECURITY_GROUP_IS_UNKNOWN (*subgroup*)

(11) SECURITY_GROUP_WOULD_BE_IN_INVALID_GRAPH (*subgroup*, *group*)

19.3.6 PROGRAM_GROUP_REMOVE_MEMBER

(1)
```
PROGRAM_GROUP_REMOVE_MEMBER (
    group       : Program_group_designator,
    program     : Static_context_designator
)
```

(2) PROGRAM_GROUP_REMOVE_MEMBER removes the static context *program* from the group *group*. The "program_member_of" link from *program* to *group* and its "has_programs" reverse link are deleted.

(3) Write locks of the default mode are obtained on the deleted links.

Errors

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

(5) ACCESS_ERRORS (*program*, ATOMIC, MODIFY, WRITE_IMPLICIT)

(6) SECURITY_GROUP_IS_UNKNOWN (*group*)

(7) STATIC_CONTEXT_IS_IN_USE (*program*)

(8) STATIC_CONTEXT_IS_NOT_MEMBER (*program*, *group*)

19.3.7 PROGRAM_GROUP_REMOVE_SUBGROUP

(1)
```
PROGRAM_GROUP_REMOVE_SUBGROUP (
    group       : Program_group_designator,
    subgroup    : Program_group_designator
)
```

(2) PROGRAM_GROUP_REMOVE_SUBGROUP removes the program group *subgroup* from the program group *group*. The "program_subgroup_of" link from *subgroup* to *group* and its "has_program_subgroups" reverse link are deleted.

(3) Write locks of the default mode are obtained on the deleted links.

Errors

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

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

(6) PROGRAM_GROUP_IS_NOT_EMPTY (*subgroup*)

(7) SECURITY_GROUP_IS_IN_USE (*subgroup*)

(8) SECURITY_GROUP_IS_NOT_A_SUBGROUP (*subgroup*, *group*)

(9) SECURITY_GROUP_IS_UNKNOWN (*group*)

(10) SECURITY_GROUP_IS_UNKNOWN (*subgroup*)

19.3.8 USER_GROUP_ADD_MEMBER

(1)
```
USER_GROUP_ADD_MEMBER (
    group   : User_group_designator,
    user    : User_designator
)
```

(2) USER_GROUP_ADD_MEMBER adds the user *user* to the user group *group*. A "user_member_of" link from *user* to *group* and a "has_users" reverse link are created. The keys of the created links are implementation-dependent.

(3) Write locks of the default mode are obtained on the created links.

Errors

(4) ACCESS_ERRORS (*group*, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*user*, ATOMIC, MODIFY, APPEND_LINKS)

(6) SECURITY_GROUP_IS_UNKNOWN (*group*)

(7) SECURITY_GROUP_IS_UNKNOWN (*user*)

(8) USER_GROUP_LACKS_ALL_USERS_AS_SUPERGROUP (*group*)

(9) USER_IS_ALREADY_MEMBER (*user*, *group*)

(10) NOTE - This operation does not cause *group* to become an adoptable group of a process running on behalf of *user*.

19.3.9 USER_GROUP_ADD_SUBGROUP

(1)
```
USER_GROUP_ADD_SUBGROUP (
    group       : User_group_designator,
    subgroup    : User_group_designator
)
```

(2) USER_GROUP_ADD_SUBGROUP adds the user group *subgroup* to the user group *group*. A "user_subgroup_of" link from *subgroup* to *group* and a "has_user_subgroups" reverse link are created. The keys of the created links are implementation-dependent.

(3) Write locks of the default mode are obtained on the created links.

Errors

(4) ACCESS_ERRORS (*group*, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*subgroup*, ATOMIC, MODIFY, APPEND_LINKS)

(6) MASTER_IS_INACCESSIBLE (some object of the graph of security groups, ATOMIC)

(7) SECURITY_GROUP_ALREADY_HAS_THIS_SUBGROUP (*subgroup*, *group*)

(8) SECURITY_GROUP_IS_IN_USE (*subgroup*)

(9) SECURITY_GROUP_IS_UNKNOWN (*group*)

(10) SECURITY_GROUP_IS_UNKNOWN (*subgroup*)

(11) SECURITY_GROUP_WOULD_BE_IN_INVALID_GRAPH (*subgroup*, *group*)

19.3.10 USER_GROUP_REMOVE_MEMBER

(1)
```
USER_GROUP_REMOVE_MEMBER (
    group   : User_group_designator,
    user    : User_designator
)
```

(2) USER_GROUP_REMOVE_MEMBER removes the user *user* from the group *group*. The "user_member_of" link from *user* to *group* and its "has_users" reverse link are deleted.

(3) Write locks of the default mode are obtained on the deleted links.

Errors

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

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

(6) SECURITY_GROUP_IS_UNKNOWN (*group*)

(7) SECURITY_GROUP_IS_UNKNOWN (*user*)

(8) USER_GROUP_IS_IN_USE (*user, group*)

(9) USER_IS_NOT_MEMBER (*user, group*)

(10) NOTE - The "adoptable_user_group" link from a process executed on behalf of *user* to *group* is not deleted (see PROCESS_ADOPT_USER_GROUP).

19.3.11 USER_GROUP_REMOVE_SUBGROUP

(1)
```
USER_GROUP_REMOVE_SUBGROUP (
    group     : User_group_designator,
    subgroup  : User_group_designator
)
```

(2) USER_GROUP_REMOVE_SUBGROUP removes the user group *subgroup* from the user group *group*. The "user_subgroup_of" link from *subgroup* to *group* and its "has_user_subgroups" reverse link are deleted.

(3) *subgroup* must not be the effective group for a running process.

(4) Write locks of the default mode are obtained on the deleted links.

Errors

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

(6) ACCESS_ERRORS (*subgroup*, ATOMIC, MODIFY, WRITE_LINKS)

(7) SECURITY_GROUP_IS_IN_USE (*subgroup*)

(8) SECURITY_GROUP_IS_NOT_A_SUBGROUP (*subgroup, group*)

(9) SECURITY_GROUP_IS_UNKNOWN (*group*)

(10) SECURITY_GROUP_IS_UNKNOWN (*subgroup*)

(11) USER_GROUP_WOULD_NOT_HAVE_ALL_USERS_AS_SUPERGROUP (*subgroup*)

20 Mandatory security

20.1 Mandatory security concepts

20.1.1 Mandatory classes

(1)
```
sds mandatory_security:
```

(2)
```
import object type system-object, system-volume, system-device, system-common_root;
```

(3)
```
import attribute type system-name, system-number;
```

(4)
```
import object type discretionary_security-security_group, discretionary_security-user;
```

(5)
```
import attribute type discretionary_security-group_identifier;
```

(6)
```
extend object type security_group with
link
    may_downgrade: (navigate) reference link (name) to confidentiality_class reverse
        downgradable_by;
    may_upgrade: (navigate) reference link (name) to integrity_class reverse
        upgradable_by;
end security_group;
```

(7)
extend object type user **with**
link
cleared_for: (**navigate**) **reference link** (name) **to** mandatory_class **reverse**
having_clearance;
end user;

(8)
mandatory_directory: **child type of** object **with**
link
known_mandatory_class: (**navigate**) **existence link** (name) **to** mandatory_class;
mandatory_classes_of: **implicit link to** common_root **reverse** mandatory_classes;
end mandatory_directory;

(9)
mandatory_class: **child type of** object **with**
link
having_clearance: (**navigate**) **reference link** (group_identifier) **to** user **reverse**
cleared_for;
end mandatory_class;

(10)
confidentiality_class: **child type of** mandatory_class **with**
link
dominates_in_confidentiality: (**navigate**) **reference link to** confidentiality_class **reverse**
confidentiality_dominator;
confidentiality_dominator: (**navigate**) **reference link to** confidentiality_class **reverse**
dominates_in_confidentiality;
downgradable_by: (**navigate**) **reference link** (group_identifier) **to** security_group
reverse may_downgrade;
end confidentiality_class;

(11)
integrity_class: **child type of** mandatory_class **with**
link
dominates_in_integrity: (**navigate**) **reference link to** integrity_class **reverse**
integrity_dominator;
integrity_dominator: (**navigate**) **reference link to** integrity_class **reverse**
dominates_in_integrity;
upgradable_by: (**navigate**) **reference link** (group_identifier) **to** security_group **reverse**
may_upgrade;
end integrity_class;

(12)
extend object type object **with**
attribute
confidentiality_label: (**read**) **string**;
integrity_label: (**read**) **string**;
end object;

(13)
extend object type common_root **with**
link
mandatory_classes: (**navigate**) **existence link to** mandatory_directory **reverse**
mandatory_classes_of;
end common_root;

(14)
end mandatory_security;

(15) The "mandatory_class" object type represents the mandatory classes defined for the PCTE installation. The name of a class, the *class name*, is the key attribute of the "known_mandatory_class" link from the mandatory directory to the mandatory class object. The destinations of the "having_clearance" links represent users which are *cleared* to this mandatory class. The concept of user clearance is elaborated in 20.1.4.

(16) The "confidentiality_class" object type represents the subset of mandatory classes which are confidentiality classes:

(17) - the destination of the "dominates_in_confidentiality" link represents the confidentiality class which this confidentiality class dominates;

(18) - the destination of the "confidentiality_dominator" link represents the confidentiality class which dominates this confidentiality class;

(19) - the destinations of the "downgradable_by" links represent the security groups which have authority to downgrade from that confidentiality class (see 20.1.4).

(20) The "integrity_class" object type represents the subset of mandatory classes which are integrity classes:

(21) - the destination of the "dominates_in_integrity" link represents the integrity class which this integrity class dominates;

(22) - the destination of the "integrity_dominator" link represents the integrity class which dominates this integrity class;

(23) - the destinations of the "upgradable_by" links represent the security groups which have authority to upgrade from that integrity class.

(24) The mandatory directory is an administrative object (see 9.1.2).

20.1.2 The mandatory class structure

(1) Confidentiality_tower = **seq1 of** Confidentiality_class_designator

(2) Integrity_tower = **seq1 of** Integrity_class_designator

(3) Each mandatory class participates in exactly one of the confidentiality towers and integrity towers which define the dominance relationships between these classes. No class may appear more than once in a tower.

(4) The "dominates_in_confidentiality" and "confidentiality_dominator" links of a mandatory class represent the sequence of confidentiality classes in a confidentiality tower. The destination of a "dominates_in_confidentiality" link is the member of the sequence which immediately precedes the origin of the link. The destination of "confidentiality_dominator" is the member of the sequence which is the immediate successor of the origin of the link.

(5) The "dominates_in_integrity" and "integrity_dominator" links represent the sequence of integrity classes in an integrity tower. The destination of a "dominates_in_integrity" link is the member of the sequence which immediately precedes the origin of the link. The destination of "integrity_dominator" is the member of the sequence which is the immediate successor of the origin of the link.

(6) The predicates CLASS_DOMINATES and CLASS_STRICTLY_DOMINATES are defined in terms of the relative positions of the mandatory classes within a confidentiality tower or an integrity tower. A class *left_class dominates* a class *right_class* if CLASS_DOMINATES (*left_class*, *right_class*) = **true**. A class *left_class strictly dominates* a class *right_class* if CLASS_STRICTLY_DOMINATES (*left_class*, *right_class*) = **true**.

(7)
```
CLASS_DOMINATES (
    left_class  : Mandatory_class_designator,
    right_class : Mandatory_class_designator
)
    result      : Boolean
```

(8) The result is **false** if *left_class* and *right_class* do not occur in the same confidentiality tower or integrity tower.

(9) If *left_class* and *right_class* occur in a confidentiality tower or an integrity tower T, and *left_class* = T(I) and *right_class* = T(J), then the result is **true** if I ≥ J, otherwise **false**.

(10)
```
CLASS_STRICTLY_DOMINATES (
    left_class  : Mandatory_class_designator,
    right_class : Mandatory_class_designator
)
    result      : Boolean
```

(11) The result is **false** if *left_class* and *right_class* do not occur in the same confidentiality tower or integrity tower.

(12) If *left_class* and *right_class* occur in a confidentiality tower or an integrity tower T, and *left_class* = T(I) and *right_class* = T(J), then the result is **true** if I > J, otherwise **false**.

20.1.3 Labels and the concept of dominance

(1)
```
Security_label = [ Mandatory_class_designator ] | Conjunction | Disjunction
```

(2)
```
Conjunction :: UNITS: set of Security_label
```

(3)
```
Disjunction :: UNITS: set of Security_label
```

(4) A security label is either a confidentiality label or an integrity label; the structure is the same in either case.

(5) A class name is a confidentiality or integrity class name. Confidentiality class names may occur only in confidentiality labels. Integrity class names may occur only in integrity labels. Conjunctions and disjunctions must contain at least 2 units. A security label of the first kind in which the optional unit is not supplied is called a *null label.*

(6) The concept of mandatory security permissions depends on the concept of a label dominating another.

(7) The predicates LABEL_DOMINATES and LABEL_STRICTLY_DOMINATES are defined in terms of the possible forms of the labels and the domination relationships between the mandatory classes. A label *left_label dominates* a label *right_label* if LABEL_DOMINATES (*left_label*, *right_label*) = **true**. A label *left_label strictly dominates* a label *right_label* if LABEL_STRICTLY_DOMINATES (*left_label*, *right_label*) = **true**.

(8)
```
LABEL_DOMINATES (
    left_label  : Security_label,
    right_label : Security_label
)
    dominates   : Boolean
```

(9) If *right_label* is null then *dominates* = **true**.

(10) If *left_label* is null and *right_label* is not null then *dominates* = **false**.

(11) If *left_label* and *right_label* are both mandatory class designators then *dominates* = CLASS_DOMINATES (*left_label*, *right_label*).

(12) If *left_label* is a mandatory class designator and *right_label* is a disjunction of mandatory class designators r_1, r_2, ... then *dominates* = **true** if CLASS_DOMINATES (*left_label*, r_j) is **true** for some r_j, and **false** otherwise.

(13) If *left_label* is a conjunction of mandatory class designators l_1, l_2, ... and *right_label* is a mandatory class designator then *dominates* = **true** if CLASS_DOMINATES (l_i, *right_label*) is **true** for some l_i, and **false** otherwise.

(14) If *left_label* is a conjunction of mandatory class designators l_1, l_2, ... and *right_label* is a disjunction of mandatory class designators r_1, r_2, ... then *dominates* = **true** if CLASS_DOMINATES(l_i, r_j) is **true** for some l_i and r_j, and **false** otherwise.

(15) If *right_label* is a disjunction of security labels r_1, r_2, ... and some r_k is a disjunction of security labels r_1', r_2', ... then *dominates* = LABEL_DOMINATES (*left_label*, *r*') where *r*' is the disjunction of all the r_j' and all the r_i except r_k.

(16) If *left_label* is a conjunction of security labels l_1, l_2, ... and some l_k is a conjunction of security labels l_1', l_2', ... then *dominates* = LABEL_DOMINATES (*l*', *right_label*) where *l*' is the conjunction of all the l_j' and all the l_i except l_k.

(17) If *right_label* is a conjunction of security labels r_1, r_2, ... then *dominates* = **true** if LABEL_DOMINATES (*left_label*, r_j) is **true** for all r_j, and **false** otherwise.

(18) If *left_label* is a disjunction of security labels l_1, l_2, ... then *dominates* = **true** if LABEL_DOMINATES (l_i, *right_label*) is **true** for all l_i, and **false** otherwise.

(19) If *right_label* is a disjunction of security labels r_1, r_2, ... and some r_k is a conjunction of security labels r_1', r_2', ... then *dominates* = **true** if LABEL_DOMINATES (*left_label*, *r*') is **true** for all r_j', where *r*' is *right_label* with r_k replaced by r_j', and **false** otherwise.

(20) If *left_label* is a conjunction of security labels l_1, l_2, ... and some l_k is a disjunction of security labels l_1', l_2', ... then *dominates* = **true** if LABEL_DOMINATES (*l*', *right_label*) is **true** for all l_i' where *l*' is *left_label* with l_k replaced by l_i', and **false** otherwise.

(21)
```
LABEL_STRICTLY_DOMINATES (
    left_label      : Security_label,
    right_label     : Security_label
)
    dominates       : Boolean
```

(22) The definition of this predicate is the same as for LABEL_DOMINATES except that:

(23) - If *left_label* and *right_label* are both null, then *dominates* is **false**.

(24) - CLASS_DOMINATES is replaced by CLASS_STRICTLY_DOMINATES.

(25) - LABEL_DOMINATES is replaced by LABEL_STRICTLY_DOMINATES.

NOTES

(26) 1 It is possible for label A to dominate label B and B to dominate A, and for label C not to dominate label D while D does not dominate C.

(27) 2 For the mapping of security labels to language bindings see 23.1.3.1.

20.1.4 Mandatory rules for information flow

(1) A *user's confidentiality clearance* is a security label derived from the confidentiality classes to which that user is cleared by forming a conjunction of the confidentiality class names.

(2) A *user's integrity clearance* is a security label derived from the integrity classes to which that user is cleared by forming a conjunction of the integrity class names.

(3) A process has a *mandatory context* associated with it which is used to control the flow of information to and from the process. This mandatory context consists of a confidentiality component called a *confidentiality context* and an integrity component called an *integrity context* .

(4) The confidentiality context and integrity context are represented by the "confidentiality_label" and "integrity_label" attributes respectively of the process as inherited from the parent type "object" (see 20.1.1).

(5) A process may change its confidentiality context during execution so that the new confidentiality context dominates the previous value.

(6) A process may change its integrity context during execution so that the new integrity context is dominated by the previous value.

(7) A process may only change its confidentiality context so that the result is dominated by the user's confidentiality clearance.

(8) When a confidentiality context is changed, it must remain within the confidentiality label range of the workstation on which the process is executing. When an integrity context is changed, it must remain within the integrity label range of the workstation on which the process is executing (see 20.1.5).

(9) An object has a confidentiality label and an integrity label which control the flow of information into and out of its associated atomic object. The following rules apply to a process' mandatory context or an object's mandatory labels after any change due to the floating of labels (see 20.1.6).

(10) For the purposes of these rules *read* and *write* are defined as follows in terms of information flow. If information flows from an object to a process, the process reads from the object. If information flows from a process to an object, even if it is only erasure, the process writes to the object. Reading and writing refer to any property of an object (attributes, links, link attributes, contents) which can contain (or embody) information. Deletion of an object is therefore considered writing to the object, although deletion of an object is only achieved by deleting a link.

(11) - The simple confidentiality rule: A process P may only read from the atomic object associated with an object A if LABEL_DOMINATES (confidentiality context of P, confidentiality label of A).

(12) - The confidentiality confinement rule: A process P may only write to the atomic object associated with an object A if LABEL_DOMINATES (confidentiality label of A, confidentiality context of P).

(13) - The simple integrity rule: A process P may only write to the atomic object associated with an object A if LABEL_DOMINATES (integrity context of P, integrity label of A).

(14) - The integrity confinement rule: A process P may only read from the atomic object associated with an object A if LABEL_DOMINATES (integrity label of A, integrity context of P).

(15) - The communication rule: A process P may transmit information to another process Q (by PROCESS_PEEK or PROCESS_POKE) if LABEL_DOMINATES (confidentiality context

of Q, confidentiality context of P) and LABEL_DOMINATES (integrity context of P, integrity context of Q).

(16) A process may change the confidentiality and integrity labels of an object if and only if it has the atomic CONTROL_MANDATORY right to that object. Under this condition, a confidentiality label may be changed to a value which dominates the previous value and an integrity label may be changed to a value which is dominated by the previous value.

(17) When a confidentiality label of an object is changed, it must remain within the confidentiality label range of the volume on which the object is residing. When an integrity label of an object is changed, it must remain within the integrity label range of the volume on which the object is residing (see 20.1.5). This is true even with the downgrade or upgrade privileges, described below, effective.

(18) If an effective security group of the calling process has additional downgrade or upgrade privileges, these object mandatory labels may be changed so that the new value of the confidentiality label does not dominate the previous value and the new value of the integrity label is not dominated by the previous value, according to the rules defined below:

(19) A process is defined to be *acting with downgrade authority from a confidentiality class* C if the process has an effective security group which has downgrade authority from C, i.e. there is a "downgradable_by" link from C to the security group. This is represented by the predicate DOWNGRADE_AUTHORITY:

(20)
```
DOWNGRADE_AUTHORITY (
    process     : Process_designator,
    class       : Confidentiality_class_designator
)
    authority   : Boolean
```

(21) A process is defined to be *acting with upgrade authority to an integrity class* C if the process has an effective group which has upgrade authority to C, i.e. there is an "upgradable_by" link from C to the security group. This is represented by the predicate UPGRADE_AUTHORITY:

(22)
```
UPGRADE_AUTHORITY (
    process     : Process_designator,
    class       : Integrity_class_designator
)
    authority   : Boolean
```

(23) A process is permitted to change a confidentiality label from *right_label* to *left_label* providing that *right_label* is *dominated in confidentiality by left_label relative to the process.* This concept is defined by the following predicates for classes and labels:

(24)
```
RELATIVE_CLASS_DOMINATES_IN_CONFIDENTIALITY (
    process     : Process_designator,
    left_class  : Mandatory_class_designator,
    right_class : Mandatory_class_designator
)
    dominates   : Boolean
```

(25) This is the same as CLASS_DOMINATES except that if DOWNGRADE_AUTHORITY (*process*, *right_class*) is **true**, *dominates* is always **true**.

(26)
```
RELATIVE_LABEL_DOMINATES_IN_CONFIDENTIALITY (
    process     : Process_designator,
    left_label  : Security_label,
    right_label : Security_label
)
    dominates : Boolean
```

(27) This is the same as LABEL_DOMINATES except that:

(28) - The rule beginning 'If *left_label* is null' (i.e. the second rule) is replaced by the rule: If *left_label* is *null* and *right_label* is a class name C, then *dominates* = DOWNGRADE_AUTHORITY (*process*, C).

(29) - RELATIVE_CLASS_DOMINATES_IN_CONFIDENTIALITY replaces CLASS_ DOMINATES.

(30) - RELATIVE_LABEL_DOMINATES_IN_CONFIDENTIALITY replaces LABEL_ DOMINATES.

(31) A process is permitted to change an integrity label from *left_label* to *right_label* providing that *left_label dominates right_label in integrity relative to the process.* This concept is defined by the following predicates for classes and labels:

(32)
```
RELATIVE_CLASS_DOMINATES_IN_INTEGRITY (
    process     : Process_designator,
    left_class  : Mandatory_class_designator,
    right_class : Mandatory_class_designator
)
    dominates : Boolean
```

(33) This is the same as CLASS_DOMINATES except that if UPGRADE_AUTHORITY (*process*, *right_class*) is **true**, *dominates* is always **true**.

(34)
```
RELATIVE_LABEL_DOMINATES_IN_INTEGRITY (
    process     : Process_designator,
    left_label  : Security_label,
    right_label : Security_label
)
    dominates : Boolean
```

(35) This is the same as LABEL_DOMINATES except that:

(36) - The rule beginning 'If *left_label* is null' (i.e. the second rule) is replaced by the rule: If *left_label* is *null* and *right_label* is a class name C, then *dominates* = UPGRADE_AUTHORITY (*process*, C).

(37) - RELATIVE_CLASS_DOMINATES_IN_INTEGRITY replaces CLASS_DOMINATES.

(38) - RELATIVE_LABEL_DOMINATES_IN_INTEGRITY replaces LABEL_DOMINATES.

(39) The confidentiality context of a process is always dominated by the user's confidentiality clearance, and the integrity clearance of a process is always dominated by the user's integrity clearance.

NOTES

(40) 1 Read and write for mandatory access control are defined in the operations in terms of information flow. If information flows from an object to the process (i.e. access errors may occur with permission READ), it is a read. If information flows from the process to an object (i.e. access errors may occur with permission CHANGE or MODIFY), even if it is only erasure, it is a write. Reading and writing refer to any property of an object (attributes, links, link attributes, contents) which can contain (or embody) information. Deletion of an object is therefore considered a write, although for PCTE, deletion of an object is only achieved by deleting a link.

(41) 2 A restriction on a process's integrity context with reference to the user's integrity clearance is unnecessary because a change is always a downgrade.

(42) 3 The restrictions to changes to a process's confidentiality context or integrity context above apply to the operations PROCESS_SET_CONFIDENTIALITY_LABEL and PROCESS_SET_INTEGRITY_LABEL, and to the floating security labels facility (see 20.1.6) when objects, whose labels would normally prevent access, are read by the process.

(43) 4 The restrictions to changes to an object's confidentiality or integrity labels above apply to the operations OBJECT_SET_CONFIDENTIALITY_LABEL and OBJECT_SET_INTEGRITY_LABEL, and to the floating security labels facility when objects, whose labels would normally prevent access, are written to by a process.

(44) 5 The predicates RELATIVE_LABEL_DOMINATES_IN_CONFIDENTIALITY and RELATIVE_LABEL_ DOMINATES_IN_INTEGRITY are used in operations OBJECT_SET_CONFIDENTIALITY_LABEL and OBJECT_SET_INTEGRITY_LABEL to define some of these checks.

(45) 6 In specifying which accesses are read and which write for mandatory access control, the intention is that the rules should be as follows.

(46) - Each object, its attributes, its contents, its outgoing links (except system-managed designation links representing the use of the object by a process and those representing locks) and their attributes, and its preferred link type and key may be treated as a separate security object.

(47) - Every access to one of those security objects that depends on data from it may be treated as a read, except that the audit selection criteria are accessible, for the purposes of determining whether the event is auditable, without a mandatory read check, and reading security labels for mandatory access checks does not count as a read.

(48) - Every access to one of those security objects that writes data to it is treated as a write, with the following exceptions:

(49) . the last_access_time attribute shall be updatable without mandatory write checks;

(50) . records shall be written to the audit file and accounting log without write mandatory checks;

(51) . updates arising as a result of process failure or abnormal closedown of a workstation shall be possible without mandatory checks.

20.1.5 Multi-level security labels

(1)
```
Multi_level_device_designator = Volume_designator | Device_designator |
    Execution_site_designator
```

(2)
```
sds mandatory_security:
```

(3)
```
import object type system-volume, system-device, system-execution_site;
```

(4)
```
extend object type volume with
attribute
    confidentiality_high_label: (read) non_duplicated string;
    confidentiality_low_label: (read) non_duplicated string;
    integrity_high_label: (read) non_duplicated string;
    integrity_low_label: (read) non_duplicated string;
end volume;
```

(5)
```
extend object type device with
attribute
    confidentiality_high_label;
    confidentiality_low_label;
    integrity_high_label;
    integrity_low_label;
    contents_confidentiality_label: (read) non_duplicated string;
    contents_integrity_label: (read) non_duplicated string;
end device;
```

(6)
```
extend object type execution_site with
attribute
    confidentiality_high_label;
    confidentiality_low_label;
    integrity_high_label;
    integrity_low_label;
end execution_site;
```

(7)
```
end mandatory_security;
```

(8) *Multi-level secure devices* are volumes, devices, and execution sites; they allow data with a fixed range of mandatory labels for confidentiality and for integrity to be stored on them. The fixed ranges of labels required for a multi-level secure device are expressed as two labels, a high label and a low label. In each range the high label must dominate the low label.

(9) A MAXIMUM_LABEL high end of range means that there is no ceiling on the labels of objects contained within the device.

(10) For it to be permissible for an object A to be stored on a multi-level secure device M, CONFIDENTIALITY_LABEL_WITHIN_RANGE (A, M) and INTEGRITY_LABEL_ WITHIN_RANGE (A, M) must be **true**, where:

(11)
```
CONFIDENTIALITY_LABEL_WITHIN_RANGE (
    object          : Object_designator,
    device          : Multi_level_device_designator
)
    inside_range    : Boolean
```

(12) *inside_range* is **true** if the confidentiality low label of *device* does not strictly dominate the confidentiality label of *object*, and the confidentiality high label of *device* either is MAXIMUM_LABEL or dominates the confidentiality label of *object*; and is otherwise **false**.

(13)
```
INTEGRITY_LABEL_WITHIN_RANGE (
    object          : Object_designator,
    device          : Multi_level_device_designator
)
    inside_range    : Boolean
```

(14) *inside_range* is **true** if the integrity low label of *device* does not strictly dominate the integrity label of *object*, and the integrity high label of *device* either is MAXIMUM_LABEL or dominates the integrity label of *object*; and is otherwise **false**.

(15) Similar checks are made when multi-level secure devices are put on other multi-level secure devices. For it to be permissible for a multi-level secure device A to reside on another multi-level secure device B, CONFIDENTIALITY_RANGE_WITHIN_RANGE(A, B) must be true, where:

(16)
```
CONFIDENTIALITY_RANGE_WITHIN_RANGE (
    inner_device    : Multi_level_device_designator,
    outer_device    : Multi_level_device_designator
)
    inside_range    : Boolean
```

(17) *inside_range* is **true** if the confidentiality low label of *outer_device* does not strictly dominate the confidentiality low label of *inner_device* , and the confidentiality high label of *outer_device* either is MAXIMUM_LABEL or dominates the confidentiality high label of *inner_device*; and is otherwise **false**.

(18)
```
INTEGRITY_RANGE_WITHIN_RANGE (
    inner_device   : Multi_level_device_designator,
    outer_device   : Multi_level_device_designator
)
    inside_range   : Boolean
```

(19) *inside_range* is **true** if the integrity low label of *outer_device* does not strictly dominate the integrity low label of *inner_device* , and the integrity high label of *outer_device* either is MAXIMUM_LABEL or dominates the integrity high label of *inner_device*; and is otherwise **false**.

(20) The confidentiality or integrity label of an object A *lies within the confidentiality* or *integrity range* of a multi-level secure device B if CONFIDENTIALITY_LABEL_WITHIN_RANGE (A, B) or INTEGRITY_LABEL_WITHIN_RANGE (A, B) respectively is **true**.

(21) The confidentiality or integrity range of a multi-level secure device A *lies within the confidentiality* or *integrity range* of a multi-level secure device B if CONFIDENTIALITY_ RANGE_WITHIN_RANGE (A, B) or INTEGRITY_RANGE_WITHIN_RANGE (A, B) respectively is **true**.

(22) In addition to its mandatory labels, a device object is associated with two other labels (one confidentiality label and one integrity label), termed *labels of contents*, which govern access to its contents through the device contents operations (see clause 12).

(23) The labels of contents are evaluated in accordance with the characteristics of the physical device each time the contents of the device is accessed. If the device is open for reading or writing, the label associated with the contents of the physical device is implementation-defined. If the label cannot be identified then the confidentiality label C of the contents is set to the confidentiality context of the accessing process, and the integrity label I of the contents is set to the integrity context of the accessing process. If *device_contents* is defined as a pseudo-object representing the contents which have the labels of contents, the process is denied access to the contents if any of the following are **false**:

(24) - CONFIDENTIALITY_LABEL_WITHIN_RANGE (*device_contents*, *device*)

(25) - INTEGRITY_LABEL_WITHIN_RANGE (*device_contents*, *device*)

(26) - LABEL_DOMINATES (confidentiality context of *process*, C)

(27) - LABEL_DOMINATES (I, integrity context of *process*)

NOTES

(28) 1 Checks are made that the constraints are obeyed on an object stored on a multi-level secure device whenever:

(29) - objects have their labels changed;

(30) - processes have their mandatory context changed;

(31) - objects are put on to multi-level secure devices:

(32) . objects are created on a volume;

(33) . objects are moved to a volume;

(34) . processes are started or called on a workstation;

(35) - copying files to foreign system.

(36) 2 The checks made that the constraints are obeyed when multi-level secure devices are put on other multi-level secure devices apply in the following situations:

(37) - volumes are created on devices;

(38) - volumes are mounted on devices;

(39) - devices are created on workstations;

(40) - security ranges on multi-level devices are changed (see 20.2.9 and 20.2.10).

20.1.6 Floating security levels

(1)
```
Floating_level = NO_FLOAT | FLOAT_IN | FLOAT_OUT | FLOAT_IN_OUT
```

(2)
```
sds mandatory_security:
```

(3)
```
import object type system-process;
```

(4)
```
floating_level: NO_FLOAT, FLOAT_IN, FLOAT_OUT, FLOAT_IN_OUT;
```

(5)
```
extend object type process with
attribute
    floating_confidentiality_level: (read) non_duplicated enumeration (floating_level) :=
        NO_FLOAT;
    floating_integrity_level: (read) non_duplicated enumeration (floating_level) :=
        NO_FLOAT;
end process;
```

(6)
```
end mandatory_security;
```

(7) The floating security levels mechanism enables a process to select either or both of the two facilities:

(8) - The mandatory context of a process may float up (confidentiality) or down (integrity) when information is read from an object.

(9) - The mandatory labels of an object may float up (confidentiality) or down (integrity) when information is written to its associated atomic object.

(10) This is specified using the "floating_confidentiality_level" and "floating_integrity_level" attributes of the executing process, which have four possible values:

(11) - NO_FLOAT: switches off the floating mechanism;

(12) - FLOAT_IN: enables the process's mandatory context to float;

(13) - FLOAT_OUT: enables the object's mandatory labels to float;

(14) - FLOAT_IN_OUT: enables both to float.

(15) If the floating of the mandatory context of a process P is enabled (FLOAT_IN and FLOAT_IN_OUT), then when information is read from the atomic object associated with an object A:

(16) - if LABEL_DOMINATES (confidentiality context of P, confidentiality label of A) is **false** then the new confidentiality context is given by FLOAT_UPGRADE (confidentiality context of P, confidentiality label of A);

(17) - if LABEL_DOMINATES (integrity label of A, integrity context of P) is **false**, then the new integrity context is given by FLOAT_DOWNGRADE (integrity context of P, integrity label of A).

(18) If the floating of an object's mandatory labels is enabled (FLOAT_OUT and FLOAT_IN_OUT), then when the atomic object associated with an object A is written to:

(19) - if LABEL_DOMINATES (confidentiality label of A, confidentiality context of the calling process) is **false** then the new confidentiality label is given by FLOAT_UPGRADE (confidentiality label of A, confidentiality context of the calling process);

(20) - if LABEL_DOMINATES (integrity context of the calling process, *integrity* label of A) is **false** then the new integrity label is given by FLOAT_DOWNGRADE (integrity label of A, integrity context of the calling process).

(21) FLOAT_UPGRADE and FLOAT_DOWNGRADE are defined as follows:

(22)
```
FLOAT_UPGRADE (
    upgradable_label  : Security_label,
    higher_label      : Security_label
)
    upgraded_label    : Security_label
```

(23) *upgraded_label* is the conjunction of *upgradable_label* and *higher_label* unless *upgradable_label* is null in which case *upgraded_label* is *higher_label*.

(24)
```
FLOAT_DOWNGRADE (
    downgradable_label  : Security_label,
    lower_label         : Security_label
)
    downgraded_label    : Security_label
```

(25) *downgraded_label* is the disjunction of *downgradable_label* and *lower_label* unless *lower_label* is null in which case *downgraded_label* is also null.

(26) The floating of mandatory labels requires the process to have the CONTROL_MANDATORY right to the object.

(27) The confidentiality context of a process *process* is subject to the constraints:

(28) - It must be dominated by the user confidentiality clearance.

(29) - It must lie within the confidentiality range of the workstation i.e. CONFIDENTIALITY_LABEL_WITHIN_RANGE (*process*, *station*) must be **true**.

(30) The confidentiality label of an object A must lie within the confidentiality range of the volume V in which it resides, i.e. CONFIDENTIALITY_LABEL_WITHIN_RANGE (A, V) must be **true**.

(31) The integrity context must continue to lie within the integrity range of the workstation on which the process is running i.e. INTEGRITY_LABEL_WITHIN_RANGE (*process*, *station*) must be **true**.

(32) The integrity label of an object A must lie within the integrity range of the volume V in which it resides, i.e. INTEGRITY_LABEL_WITHIN_RANGE (A, V) must be **true**.

NOTES

(33) 1 If any of the above conditions results in the process's mandatory context or the object's mandatory label not being changed, then reading and writing of the object are forbidden, as defined in 20.1.4.

(34) 2 CONTROL_MANDATORY right is required for label changes to be effected either explicitly using OBJECT_SET_CONFIDENTIALITY_LABEL and OBJECT_SET_INTEGRITY_LABEL or implicitly using floating security labels.

(35) 3 In order to determine whether these constraints have been violated, access must be made to the objects involved i.e. the user, the station and the volume. These accesses are not also subject to mandatory access control, which

could lead to the further floating of the mandatory context of the current process. These accesses constitute additional bitwise read accesses which are intrinsic covert channels to PCTE (see 20.1.8.2) and are permitted.

(36) 4 An object of type "process" (or a descendant type) cannot have its mandatory labels changed by output floating, regardless of the process status. An operation which tries to write to such an object and would cause floating fails with the relevant confinement violation error.

20.1.7 Implementation restrictions

(1) A trusted implementation of PCTE may have implementation-defined restrictions on various aspects of the security model. In particular there may be implementation-defined restrictions of the following kinds:

(2) - restrictions on the number of confidentiality classes (0 or more);

(3) - restrictions on the number of integrity classes (0 or more);

(4) - restrictions on the form of the confidentiality labels, e.g. may not allow a disjunction;

(5) - restrictions on the form of the integrity labels, e.g. may not allow a conjunction;

(6) - restrictions on creation of links between levels (e.g. may not allow any links to cross differently labelled objects for designated information classes).

(7) In some implementations there may be predefined classes. These predefined classes may be protected using particular implementation-defined techniques.

20.1.8 Built-in policy aspects

(1) Some aspects of the security policy of any PCTE environment are enforced by the PCTE interfaces. Any attempt to violate the built-in policy aspect raises the error condition SECURITY_POLICY_WOULD_BE_VIOLATED.

20.1.8.1 Protection of predefined SDSs

(1) The predefined SDSs "system", "discretionary_security", "mandatory_security", "metasds" and "accounting" have to be protected against any modification.

(2) Thus, for all these SDSs, the atomic and composite ACLs contain an entry corresponding to the predefined security group ALL_USERS - which is automatically set in the discretionary context of all processes - with WRITE_ATTRIBUTES, WRITE_LINKS, APPEND_LINKS and DELETE access DENIED. The other access rights are set to UNDEFINED.

(3) NOTE – Any attempt by clause 10 operations to change a predefined SDS is forbidden.

20.1.8.2 Covert channels

(1) A *covert channel* is a communication channel that allows a process to transfer information in a manner which violates the system's security policy. The mandatory and discretionary security conditions defined in previous clauses are enforced throughout PCTE. An appropriate error condition is raised whenever a given operation would result in a violation of such rules and of the other aspects of the built-in policy.

(2) Two kinds of access are identified for the purposes of mandatory security:

(3) - *data access*: accesses of this kind are implied when data items are explicitly transferred between a process and an object.

(4) - *bitwise access*: accesses of this kind are implied when the status of an object (or of a process) is modified or queried as a side effect of an operation. The term "status" is used here as opposed to the data values held in the object and which can be manipulated via the data accesses defined above.

(5) A bitwise read access is:

(6) - an integrity covert channel where the process strictly dominates the object in integrity;

(7) - a confidentiality covert channel where the object strictly dominates the process in confidentiality.

(8) A bitwise write access is:

(9) - a confidentiality covert channel where the process strictly dominates the object in confidentiality;

(10) - an integrity covert channel where the object strictly dominates the process in integrity.

(11) Both kinds of access imply transfer of information between processes and objects (or other processes). However, in the built-in policy, control of information flow is dealt with differently for the two kinds of access:

(12) - all "data accesses" must conform to the mandatory security rules as defined earlier in this major clause;

(13) - a certain number of "bitwise accesses" are allowed which would otherwise violate the security rules. These are classified as intrinsic covert channels. PCTE implementations can restrict information flow through covert channels. The events leading to intrinsic covert channels are all those associated with bitwise write accesses.

(14) The following operations imply bitwise read access:

(15) - LOCK_SET_OBJECT, LOCK_UNSET_OBJECT, LOCK_SET_INTERNAL_MODE, LOCK_RESET_INTERNAL_MODE;

(16) - any access to an object which implies a check on access rights.

(17) The following operations imply bitwise write access:

(18) - LOCK_SET_OBJECT, LOCK_UNSET_OBJECT, LOCK_SET_INTERNAL_MODE, LOCK_RESET_INTERNAL_MODE;

(19) - ACTIVITY_START, ACTIVITY_ABORT, ACTIVITY_END;

(20) - MESSAGE_RECEIVE_NO_WAIT, MESSAGE_RECEIVE_WAIT, MESSAGE_ PEEK if the message queue is full;

(21) - LINK_CREATE (creation of an implicit link);

(22) - any write to the audit file;

(23) - any write to the accounting log;

(24) - any implicit creation or deletion of a usage designation link;

(25) - any operation which creates or deletes an object (creation or deletion of an "object_on_volume" link);

(26) - OBJECT_MOVE, on the destinations of external non-designation links of the object (if moved) and each moved component.

20.2 Operations for mandatory security operation

20.2.1 DEVICE_SET_CONFIDENTIALITY_RANGE

(1)
```
DEVICE_SET_CONFIDENTIALITY_RANGE (
    device      : Device_designator,
    high_label  : Security_label,
    low_label   : Security_label
)
```

(2) DEVICE_SET_CONFIDENTIALITY_RANGE sets the confidentiality range high label and confidentiality range low label of *device* to *high_label* and *low_label* respectively, subject to the following conditions, where *device'* is *device* with the confidentiality range so changed, *station* is the workstation controlling *device*, *simply_enlarged* is CONFIDENTIALITY_RANGE_WITHIN_RANGE (*device*, *device'*), and *simply_reduced* is CONFIDENTIALITY_RANGE_WITHIN_RANGE (*device'*, *device*):

(3) - If *simply_enlarged* or not *simply_reduced*, then CONFIDENTIALITY_RANGE_WITHIN_RANGE (*device'*, *station*) must be **true**.

(4) - If *simply_reduced* or not *simply_enlarged*, and there is a volume *volume* mounted on *device*, then CONFIDENTIALITY_RANGE_WITHIN_RANGE (*volume*, *device'*) must be **true**.

(5) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

(6) A write lock of the default mode is obtained on *device*.

Errors

(7) ACCESS_ERRORS (*device*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(8) DEVICE_IS_UNKNOWN (*device*)

(9) CONFIDENTIALITY_LABEL_IS_INVALID (*high_label*)

(10) CONFIDENTIALITY_LABEL_IS_INVALID (*low_label*)

(11) LABEL_RANGE_IS_BAD (*high_label*, *low_label*)

(12) PROCESS_IS_IN_TRANSACTION

(13) RANGE_IS_OUTSIDE_RANGE (*object*, *station*)

(14) If there is a volume *volume* mounted on the device:
RANGE_IS_OUTSIDE_RANGE (*volume*, *object*)

(15) NOTE - It is possible that the range is being enlarged and reduced at the same time, e.g. if both *high_label* and *low_label* are upgrades, in which case all relevant constraints must be applied.

20.2.2 DEVICE_SET_INTEGRITY_RANGE

(1)
```
DEVICE_SET_INTEGRITY_RANGE (
    device      : Device_designator,
    high_label  : Security_label,
    low_label   : Security_label
)
```

(2) DEVICE_SET_INTEGRITY_RANGE sets the integrity range high label and integrity range low label of *device* to *high_label* and *low_label* respectively, subject to the following conditions, where *device'* is *device* with the integrity range so changed, *station* is the workstation controlling *device*, *simply_enlarged* is INTEGRITY_RANGE_WITHIN_RANGE

(*device*, *device'*) and *simply_reduced* is INTEGRITY_RANGE_WITHIN_RANGE (*device'*, *device*):

(3) - If *simply_enlarged* or not *simply_reduced*, then INTEGRITY_RANGE_WITHIN_RANGE (*device'*, *station*) must be **true**.

(4) - If *simply_reduced* or not *simply_enlarged*, and there is a volume *volume* mounted on *device*, then INTEGRITY_RANGE_WITHIN_RANGE (*volume*, *device'*) must be **true**.

(5) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

(6) A write lock of the default mode is obtained on *device*.

Errors

(7) ACCESS_ERRORS (*device*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(8) DEVICE_IS_UNKNOWN (*device*)

(9) INTEGRITY_LABEL_IS_INVALID (*high_label*)

(10) INTEGRITY_LABEL_IS_INVALID (*low_label*)

(11) LABEL_RANGE_IS_BAD (*high_label*, *low_label*)

(12) PROCESS_IS_IN_TRANSACTION

(13) RANGE_IS_OUTSIDE_RANGE (*object*, *station*)

(14) If there is a volume *volume* mounted on the device:
RANGE_IS_OUTSIDE_RANGE (*volume*, *object*)

(15) NOTE - It is possible that the range is being enlarged and reduced at the same time, e.g. if both *high_label* and *low_label* are upgrades, in which case all relevant constraints must be applied.

20.2.3 EXECUTION_SITE_SET_CONFIDENTIALITY_RANGE

(1)
```
EXECUTION_SITE_SET_CONFIDENTIALITY_RANGE (
    execution_site : Execution_site_designator,
    high_label     : Security_label,
    low_label      : Security_label
)
```

(2) EXECUTION_SITE_SET_CONFIDENTIALITY_RANGE sets the confidentiality range high label and confidentiality range low label of *execution_site* to *high_label* and *low_label* respectively, subject to the following conditions, where *execution_site'* is *execution_site* with the confidentiality range so changed.

(3) If CONFIDENTIALITY_RANGE_WITHIN_RANGE (*execution_site'*, *execution_site*) or not CONFIDENTIALITY_RANGE_WITHIN_RANGE (*execution_site*, *execution_site'*):

(4) - for each device D controlled by *execution_site*, CONFIDENTIALITY_RANGE_WITHIN_RANGE (D, *execution_site'*) is **true**.

(5) - for each process P executing on *execution_site*, CONFIDENTIALITY_LABEL_WITHIN_RANGE (P, *execution_site'*) is **true**.

(6) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

(7) A write lock of the default mode is established on *execution_site*.

Errors

(8) ACCESS_ERRORS (*execution_site*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(9) DEVICE_IS_UNKNOWN (execution_site)

(10) CONFIDENTIALITY_LABEL_IS_INVALID (high_label)

(11) CONFIDENTIALITY_LABEL_IS_INVALID (low_label)

(12) LABEL_IS_OUTSIDE_RANGE (D, execution_site)

(13) LABEL_IS_OUTSIDE_RANGE (P, *execution_site*)

(14) LABEL_RANGE_IS_BAD (*high_label*, *low_label*)

(15) PROCESS_IS_IN_TRANSACTION

20.2.4 EXECUTION_SITE_SET_INTEGRITY_RANGE

(1)
```
EXECUTION_SITE_SET_INTEGRITY_RANGE (
    execution_site : Execution_site_designator,
    high_label     : Security_label,
    low_label      : Security_label
)
```

(2) EXECUTION_SITE_SET_INTEGRITY_RANGE sets the integrity range high label and integrity range low label of *execution_site* to *high_label* and *low_label* respectively, subject to the following conditions, where *execution_site'* is *execution_site* with the integrity range so changed.

(3) If INTEGRITY_RANGE_WITHIN_RANGE (*execution_site'*, *execution_site*) or not INTEGRITY_RANGE_WITHIN_RANGE (*execution_site*, *execution_site'*):

(4) - for each device D controlled by *execution_site*, INTEGRITY_RANGE_WITHIN_RANGE (D, *execution_site'*) is **true**.

(5) - for each process P executing on *execution_site*, INTEGRITY_LABEL_WITHIN_RANGE (P, *execution_site'*) is **true**.

(6) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

(7) A write lock of the default mode is established on *execution_site*.

Errors

(8) ACCESS_ERRORS (*execution_site*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(9) DEVICE_IS_UNKNOWN (*execution_site*)

(10) INTEGRITY_LABEL_IS_INVALID (*high_label*)

(11) INTEGRITY_LABEL_IS_INVALID (*low_label*)

(12) LABEL_IS_OUTSIDE_RANGE (D, *execution_site*)

(13) LABEL_IS_OUTSIDE_RANGE (P, *execution_site*)

(14) LABEL_RANGE_IS_BAD (*high_label*, *low_label*)

(15) PROCESS_IS_IN_TRANSACTION

20.2.5 OBJECT_SET_CONFIDENTIALITY_LABEL

(1)
```
OBJECT_SET_CONFIDENTIALITY_LABEL (
    object  : Object_designator,
    label   : Security_label
)
```

(2) OBJECT_SET_CONFIDENTIALITY_LABEL sets the confidentiality label of *object* to *label*.

(3) If the previous value of the confidentiality label of *object* is L, then RELATIVE_LABEL_DOMINATES_IN_CONFIDENTIALITY (calling process, *label*, L) must be **true**.

(4) CONFIDENTIALITY_LABEL_WITHIN_RANGE (*object*, *volume*) must remain **true**, where *volume* is the volume on which the *object* resides.

(5) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

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

Errors

(7) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(8) CONFIDENTIALITY_CONFINEMENT_WOULD_BE_VIOLATED (*object*, ATOMIC)

(9) CONFIDENTIALITY_LABEL_IS_INVALID (*label*)

(10) LABEL_IS_OUTSIDE_RANGE (*object*, *volume*)

(11) OBJECT_IS_A_PROCESS (*object*)

(12) OBJECT_LABEL_CANNOT_BE_CHANGED_IN_TRANSACTION (*object*)

20.2.6 OBJECT_SET_INTEGRITY_LABEL

(1)
```
OBJECT_SET_INTEGRITY_LABEL (
    object  : Object_designator,
    label   : Security_label
)
```

(2) OBJECT_SET_INTEGRITY_LABEL sets the integrity label of *object* to *label*.

(3) If the previous value of the integrity label of *object* is L, then RELATIVE_LABEL_DOMINATES_IN_INTEGRITY (calling process, L, label) must be true.

(4) INTEGRITY_LABEL_WITHIN_RANGE (*object*, *volume*) must remain **true**, where *volume* is the volume on which *object* resides.

(5) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

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

Errors

(7) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(8) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (*object*, ATOMIC)

(9) INTEGRITY_LABEL_IS_INVALID (*label*)

(10) LABEL_IS_OUTSIDE_RANGE (*object*,*volume*)

(11) OBJECT_IS_A_PROCESS (*object*)

(12) OBJECT_LABEL_CANNOT_BE_CHANGED_IN_TRANSACTION (*object*)

20.2.7 VOLUME_SET_CONFIDENTIALITY_RANGE

(1)
```
VOLUME_SET_CONFIDENTIALITY_RANGE (
    volume     : Volume_designator,
    high_label : Security_label,
    low_label  : Security_label
    )
```

(2) VOLUME_SET_CONFIDENTIALITY_RANGE sets the confidentiality high label and confidentiality low label of *volume* to *high_label* and *low_label* respectively subject to the following conditions, where *volume'* is *volume* with its confidentiality range so changed, *simply_enlarged* is CONFIDENTIALITY_RANGE_WITHIN_RANGE (*volume*, *volume'*), and *simply_reduced* is CONFIDENTIALITY_RANGE_WITHIN_RANGE (*volume'*, *volume*).

(3) - If *simply_enlarged* or not *simply_reduced*, let *device* be the device on which the *volume* is mounted, then CONFIDENTIALITY_RANGE_WITHIN_RANGE (*volume'*, *device*) must be **true**.

(4) - If *simply_reduced* or not *simply_enlarged*, then for each object G residing on the volume, CONFIDENTIALITY_LABEL_WITHIN_RANGE (G, *volume'*) must be **true**.

(5) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

(6) A write lock of the default mode is obtained on *volume*.

Errors

(7) ACCESS_ERRORS (*volume*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(8) CONFIDENTIALITY_LABEL_IS_INVALID (*high_label*)

(9) CONFIDENTIALITY_LABEL_IS_INVALID (*low_label*)

(10) For each object G residing on *volume*:
LABEL_IS_OUTSIDE_RANGE (G, *device*)

(11) LABEL_RANGE_IS_BAD (*high_label*, *low_label*)

(12) RANGE_IS_OUTSIDE_RANGE (*volume*, *device*)

(13) PROCESS_IS_IN_TRANSACTION

(14) VOLUME_HAS_OBJECT_OUTSIDE_RANGE (*volume*, *high_label*, *low_label*)

(15) VOLUME_IS_UNKNOWN (*volume*)

(16) NOTE - It is possible that the range is being enlarged and reduced at the same time, e.g. if both *high_label* and *low_label* are upgrades, in which case both constraints must be applied.

20.2.8 VOLUME_SET_INTEGRITY_RANGE

(1)
```
VOLUME_SET_INTEGRITY_RANGE (
    volume     : Volume_designator,
    high_label : Security_label,
    low_label  : Security_label
)
```

(2) VOLUME_SET_INTEGRITY_RANGE sets the integrity range high label and integrity range low label of *volume* to *high_label* and *low_label* respectively subject to the following conditions, where *volume'* is *volume* with its integrity range so changed, *simply_enlarged* is

INTEGRITY_RANGE_WITHIN_RANGE (*volume, volume'*), and *simply_reduced* is INTEGRITY_RANGE_WITHIN_RANGE (*volume', volume*):

(3) - If *simply_enlarged* or not *simply_reduced*, let *device* be the device on which the *volume* is mounted, then INTEGRITY_RANGE_WITHIN_RANGE (*volume', device*) must be **true**.

(4) - If *simply_reduced* or not *simply_enlarged*, then for each object G residing on the volume, INTEGRITY_LABEL_WITHIN_RANGE (G, *volume'*) must be **true**.

(5) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

(6) A write lock of the default mode is obtained on *volume*.

Errors

(7) ACCESS_ERRORS (*volume*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(8) INTEGRITY_LABEL_IS_INVALID (*high_label*)

(9) INTEGRITY_LABEL_IS_INVALID (*low_label*)

(10) For each object G residing on *volume*:
LABEL_IS_OUTSIDE_RANGE (G, *device*)

(11) LABEL_RANGE_IS_BAD (*high_label*, *low_label*)

(12) RANGE_IS_OUTSIDE_RANGE (*volume*, *device*)

(13) PROCESS_IS_IN_TRANSACTION

(14) VOLUME_HAS_OBJECT_OUTSIDE_RANGE (*volume*, *high_label*, *low_label*)

(15) VOLUME_IS_UNKNOWN (*volume*)

(16) NOTE - It is possible that the range is being enlarged and reduced at the same time, e.g. if both *high_label* and *low_label* are upgrades, in which case both constraints must be applied.

20.3 Mandatory security administration operations

20.3.1 CONFIDENTIALITY_CLASS_INITIALIZE

(1)
```
CONFIDENTIALITY_CLASS_INITIALIZE (
    object           : Confidentiality_class_designator,
    class_name       : Name,
    to_be_dominated  : [ Confidentiality_class_designator ]
)
```

(2) CONFIDENTIALITY_CLASS_INITIALIZE initializes *object* as a confidentiality class. A "known_mandatory_class" link keyed by *class_name* is created from the master of the mandatory directory to *object*. If *to_be_dominated* is supplied, a "dominates_in_confidentiality" link is created from *object* to *to_be_dominated*, and a "confidentiality_dominator" link is created from *to_be_dominated* to *object*.

(3) If *to_be_dominated* is not supplied, the operation creates a new confidentiality tower consisting of the one confidentiality class *object*. If *to_be_dominated* is supplied, the operation adds *object* to the tail (the 'top') of an existing confidentiality tower.

(4) Write locks of the default mode are obtained on the created links.

Errors

(5) ACCESS_ERRORS (the mandatory directory, ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(7) If *to_be_dominated* is supplied:
ACCESS_ERRORS (*object*, ATOMIC, MODIFY, APPEND_LINKS)
ACCESS_ERRORS (*to_be_dominated*, ATOMIC, MODIFY, APPEND_LINKS)

(8) MANDATORY_CLASS_IS_ALREADY_DOMINATED (*to_be_dominated*)

(9) MANDATORY_CLASS_IS_KNOWN(*object*)

(10) MANDATORY_CLASS_IS_UNKNOWN (*to_be_dominated*)

(11) MANDATORY_CLASS_NAME_IS_IN_USE (*class_name*)

(12) PROCESS_IS_IN_TRANSACTION

(13) NOTE - This operation does not change any copies of the mandatory directory.

20.3.2 GROUP_DISABLE_FOR_CONFIDENTIALITY_DOWNGRADE

(1)
```
GROUP_DISABLE_FOR_CONFIDENTIALITY_DOWNGRADE (
    group                 : User_designator | User_group_designator |
                                Program_group_designator,
    confidentiality_class : Confidentiality_class_designator
)
```

(2) GROUP_DISABLE_FOR_CONFIDENTIALITY_DOWNGRADE deletes a "may_downgrade" link from *group* to *confidentiality_class* and a "downgradable_by" link from *confidentiality_class* to *group*.

(3) Write locks of the default mode are obtained on the deleted links.

Errors

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

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

(6) MANDATORY_CLASS_IS_UNKNOWN (*confidentiality_class*)

(7) SECURITY_GROUP_IS_NOT_ENABLED (*group, confidentiality_class*)

(8) SECURITY_GROUP_IS_UNKNOWN (*group*)

20.3.3 GROUP_DISABLE_FOR_INTEGRITY_UPGRADE

(1)
```
GROUP_DISABLE_FOR_INTEGRITY_UPGRADE (
    group           : User_designator | User_group_designator | Program_group_designator,
    integrity_class : Confidentiality_class_designator
)
```

(2) GROUP_DISABLE_FOR_INTEGRITY_UPGRADE deletes a "may_upgrade" link from *group* to *integrity_class* and an "upgradable_by" link from *integrity_class* to *group*.

(3) Write locks of the default mode are obtained on the links so deleted.

Errors

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

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

(6) MANDATORY_CLASS_IS_UNKNOWN (*integrity_class*)

(7) SECURITY_GROUP_IS_NOT_ENABLED (*group*, *integrity_class*)

(8) SECURITY_GROUP_IS_UNKNOWN (*group*)

20.3.4 GROUP_ENABLE_FOR_CONFIDENTIALITY_DOWNGRADE

(1)
```
GROUP_ENABLE_FOR_CONFIDENTIALITY_DOWNGRADE (
    group                 : User_designator | User_group_designator |
                                Program_group_designator,
    confidentiality_class : Confidentiality_class_designator
)
```

(2) GROUP_ENABLE_FOR_CONFIDENTIALITY_DOWNGRADE creates a "may_downgrade" link, keyed by the confidentiality class name of *confidentiality_class*, from *group* to *confidentiality_class* and a "downgradable_by" link, keyed by the group identifier, from *confidentiality_class* to *group*.

(3) Write locks of the default mode are obtained on the links so created.

Errors

(4) ACCESS_ERRORS (*group*, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*confidentiality_class*, ATOMIC, MODIFY, APPEND_LINKS)

(6) MANDATORY_CLASS_IS_UNKNOWN (*confidentiality_class*)

(7) SECURITY_GROUP_IS_ALREADY_ENABLED (*group*, *confidentiality_class*)

(8) SECURITY_GROUP_IS_UNKNOWN (*group*)

20.3.5 GROUP_ENABLE_FOR_INTEGRITY_UPGRADE

(1)
```
GROUP_ENABLE_FOR_INTEGRITY_UPGRADE (
    group           : User_designator | User_group_designator | Program_group_designator,
    integrity_class : Confidentiality_class_designator
)
```

(2) GROUP_ENABLE_FOR_INTEGRITY_UPGRADE creates a "may_upgrade" link, keyed by the integrity class name of *integrity_class*, from *group* to *integrity_class* and an "upgradable_by" link, keyed by the group identifier, from *integrity_class* to *group*.

(3) Write locks of the default mode are obtained on the links so created.

Errors

(4) ACCESS_ERRORS (*group*, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*integrity_class*, ATOMIC, MODIFY, APPEND_LINKS)

(6) MANDATORY_CLASS_IS_UNKNOWN (*integrity_class*)

(7) SECURITY_GROUP_IS_ALREADY_ENABLED (*group*, *integrity_class*)

(8) SECURITY_GROUP_IS_UNKNOWN (*group*)

20.3.6 INTEGRITY_CLASS_INITIALIZE

(1)
```
INTEGRITY_CLASS_INITIALIZE (
    object          : Integrity_class_designator,
    class_name      : Name,
    to_be_dominated : [ Integrity_class_designator ]
)
```

(2) INTEGRITY_CLASS_INITIALIZE initializes *object* as an integrity class. A "mandatory_class" link keyed by *class_name* is created from the master of the mandatory directory to *object*. If *to_be_dominated* is supplied, a "dominates_in_integrity" link is created

from *object* to *to_be_dominated*, and a "integrity_dominator" link is created from *to_be_dominated* to *object*.

(3) If *to_be_dominated* is not supplied, the operation creates a new integrity tower consisting of the one integrity class *object*. If *to_be_dominated* is supplied, the operation adds *object* to the tail (the "top") of an existing integrity tower.

(4) Write locks of the default mode are obtained on the created links.

Errors

(5) ACCESS_ERRORS (the mandatory directory, ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*object*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(7) If *to_be_dominated* is supplied:
ACCESS_ERRORS (*object*, ATOMIC, MODIFY, APPEND_LINKS)
ACCESS_ERRORS (*to_be_dominated*, ATOMIC, MODIFY, APPEND_LINKS)

(8) MANDATORY_CLASS_IS_ALREADY_DOMINATED (*to_be_dominated*)

(9) MANDATORY_CLASS_IS_KNOWN(*object*)

(10) MANDATORY_CLASS_IS_UNKNOWN (*to_be_dominated*)

(11) MANDATORY_CLASS_NAME_IS_IN_USE (*class_name*)

(12) PROCESS_IS_IN_TRANSACTION

(13) NOTE - This operation does not change any copies of the mandatory directory.

20.3.7 USER_EXTEND_CONFIDENTIALITY_CLEARANCE

(1)
```
USER_EXTEND_CONFIDENTIALITY_CLEARANCE (
    user                    : User_designator,
    confidentiality_class   : Confidentiality_class_designator
)
```

(2) USER_EXTEND_CONFIDENTIALITY_CLEARANCE creates a "cleared_for" link, keyed by the name of the confidentiality class *confidentiality_class*, from *user* to *confidentiality_class* and a "having_clearance" link, keyed by the group identifier, from *confidentiality_class* to *user*.

(3) Write locks of the default mode are obtained on the links so created.

Errors

(4) ACCESS_ERRORS (*user*, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*confidentiality_class*, ATOMIC, MODIFY, APPEND_LINKS)

(6) MANDATORY_CLASS_IS_UNKNOWN (*confidentiality_class*)

(7) SECURITY_GROUP_IS_UNKNOWN (*user*)

(8) USER_IS_ALREADY_CLEARED_TO_CLASS (*user*, *confidentiality_class*)

(9) USER_IS_IN_USE (*user*)

20.3.8 USER_EXTEND_INTEGRITY_CLEARANCE

(1)
```
USER_EXTEND_INTEGRITY_CLEARANCE (
    user            : User_designator,
    integrity_class : Integrity_class_designator
)
```

(2) USER_EXTEND_INTEGRITY_CLEARANCE creates a "cleared_for" link, keyed by the name of the integrity class *integrity_class*, from *user* to *integrity_class*, and a "having_clearance" link, keyed by the group identifier, from *integrity_class* to *user*.

(3) Write locks of the default mode are obtained on the links so created.

Errors

(4) ACCESS_ERRORS (*user*, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*integrity_class*, ATOMIC, MODIFY, APPEND_LINKS)

(6) MANDATORY_CLASS_IS_UNKNOWN (*integrity_class*)

(7) SECURITY_GROUP_IS_UNKNOWN (*user*)

(8) USER_IS_ALREADY_CLEARED_TO_CLASS (*user*, *integrity_class*)

(9) USER_IS_IN_USE (*user*)

20.3.9 USER_REDUCE_CONFIDENTIALITY_CLEARANCE

(1)
```
USER_REDUCE_CONFIDENTIALITY_CLEARANCE (
    user                    : User_designator,
    confidentiality_class   : Confidentiality_class_designator
)
```

(2) USER_REDUCE_CONFIDENTIALITY_CLEARANCE deletes a "cleared_for" link from *user* to *confidentiality_class* or to a confidentiality class which dominates *confidentiality_class* and a "having_clearance" link from that confidentiality class to *user*.

(3) Write locks of the default mode are obtained on the links so deleted.

Errors

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

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

(6) MANDATORY_CLASS_IS_UNKNOWN (*confidentiality_class*)

(7) SECURITY_GROUP_IS_UNKNOWN (*user*)

(8) USER_IS_NOT_CLEARED_TO_CLASS (*user*, *confidentiality_class*)

(9) USER_IS_IN_USE (*user*)

(10) NOTE - There is at most one link that satisfies the conditions above for deletion.

20.3.10 USER_REDUCE_INTEGRITY_CLEARANCE

(1)
```
USER_REDUCE_INTEGRITY_CLEARANCE (
    user            : User_designator,
    integrity_class : Integrity_class_designator
)
```

(2) USER_REDUCE_INTEGRITY_CLEARANCE deletes a "cleared_for" link from *user* to *integrity_class* or to an integrity class which dominates *integrity_class* and a "having_clearance" link from that integrity class to *user*.

(3) Write locks of the default mode are obtained on the deleted links.

Errors

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

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

(6) MANDATORY_CLASS_IS_UNKNOWN (*integrity_class*)

(7) SECURITY_GROUP_IS_UNKNOWN (*user*)

(8) USER_IS_NOT_CLEARED_TO_CLASS (*user*, *integrity_class*)

(9) USER_IS_IN_USE (*user*)

20.4 Mandatory security operations for processes

20.4.1 PROCESS_SET_CONFIDENTIALITY_LABEL

(1)
```
PROCESS_SET_CONFIDENTIALITY_LABEL (
    process                 : [ Process_designator ],
    confidentiality_label   : Security_label
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_CONFIDENTIALITY_LABEL sets the confidentiality label of *process* to *confidentiality_label.*

(4) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

Errors

(5) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(6) CONFIDENTIALITY_LABEL_IS_INVALID (*confidentiality_label*)

(7) If *process* is the calling process:
LABEL_IS_OUTSIDE_RANGE (*process*, execution site of *process*)

(8) LABEL_IS_OUTSIDE_RANGE (*process*, volume on which *process* resides)

(9) PROCESS_CONFIDENTIALITY_IS_NOT_DOMINATED (*confidentiality_label*, *process*)

(10) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(11) PROCESS_IS_UNKNOWN (*process*)

(12) USER_IS_NOT_CLEARED (*process*, *confidentiality_label*)

20.4.2 PROCESS_SET_FLOATING_CONFIDENTIALITY_LEVEL

(1)
```
PROCESS_SET_FLOATING_CONFIDENTIALITY_LEVEL (
    process         : [ Process_designator ],
    floating_mode   : Floating_level
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_FLOATING_CONFIDENTIALITY_LEVEL sets the floating confidentiality level of *process* to *floating_mode.*

Errors

(4) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(5) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(6) PROCESS_IS_UNKNOWN (*process*)

20.4.3 PROCESS_SET_FLOATING_INTEGRITY_LEVEL

(1)
```
PROCESS_SET_FLOATING_INTEGRITY_LEVEL (
    process          : [ Process_designator ],
    floating_mode    : Floating_level
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_FLOATING_INTEGRITY_LEVEL sets the floating integrity level of *process* to *floating_mode*.

Errors

(4) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, MODIFY, WRITE_ATTRIBUTES)

(5) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(6) PROCESS_IS_UNKNOWN (*process*)

20.4.4 PROCESS_SET_INTEGRITY_LABEL

(1)
```
PROCESS_SET_INTEGRITY_LABEL (
    process          : [ Process_designator ],
    integrity_label  : Security_label
)
```

(2) If no value is supplied for *process*, *process* designates the calling process.

(3) PROCESS_SET_INTEGRITY_LABEL sets the integrity label of *process* to *integrity_label*.

(4) If floating of security labels is switched on for the calling process, the facility is ignored for this operation.

Errors

(5) If *process* is not the calling process:
ACCESS_ERRORS (*process*, ATOMIC, CHANGE, CONTROL_MANDATORY)

(6) PROCESS_INTEGRITY_DOES_NOT_DOMINATE (*integrity_label*, *process*)

(7) INTEGRITY_LABEL_IS_INVALID (*integrity_label*)

(8) If *process* is the calling process:
LABEL_IS_OUTSIDE_RANGE (*process*, execution site of *process*)

(9) LABEL_IS_OUTSIDE_RANGE (*process*, volume on which *process* resides)

(10) If *process* is not the calling process:
PROCESS_LACKS_REQUIRED_STATUS (*process*, READY)

(11) PROCESS_IS_UNKNOWN (*process*)

21 Auditing

21.1 Auditing concepts

21.1.1 Audit files

(1) Selectable_event_type = WRITE | READ | COPY | ACCESS_CONTENTS | EXPLOIT
| CHANGE_ACCESS_CONTROL_LIST | CHANGE_LABEL
| USE_PREDEFINED_GROUP | SET_USER_IDENTITY
| WRITE_CONFIDENTIALITY_VIOLATION | READ_CONFIDENTIALITY_VIOLATION
| WRITE_INTEGRITY_VIOLATION | READ_INTEGRITY_VIOLATION | COVERT_CHANNEL
| INFORMATION

(2) Mandatory_event_type = CHANGE_IDENTIFICATION | SELECT_AUDIT_EVENT
| SECURITY_ADMINISTRATION

(3) Auditing_record = Object_auditing_record
| Exploit_auditing_record
| Information_auditing_record
| Copy_auditing_record
| Security_auditing_record

(4) Basic_auditing_record ::
USER : Group_identifier
TIME : Time
WORKSTATION : Execution_site_identifier
EVENT_TYPE : Selectable_event_type | Mandatory_event_type
RETURN_CODE : Return_code
PROCESS : Exact_identifier

(5) Object_auditing_record :: Basic_auditing_record &&
OBJECT : Exact_identifier

(6) Exploit_auditing_record :: Basic_auditing_record &&
NEW_PROCESS : Exact_identifier
EXPLOITED_OBJECT : Exact_identifier

(7) Information_auditing_record :: Basic_auditing_record &&
INFORMATION : String

(8) Copy_auditing_record :: Basic_auditing_record &&
SOURCE : Exact_identifier
DESTINATION : Exact_identifier

(9) Security_auditing_record :: Basic_auditing_record &&
GROUP : Exact_identifier

(10) Exact_identifier = Text

(11) Audit_file = **seq of** Auditing_record

(12) Return_code = FAILURE | SUCCESS

(13) **sds** discretionary_security:

(14) **import object type** system-object, system-workstation;

(15) audit_file: **child type of** object **with**
contents audit_file;
link
audit_of: **reference link** (number) **to** workstation **reverse** audit;
end audit_file;

(16)
```
extend object type workstation with
link
    audit: (navigate) existence link to audit_file reverse audit_of;
end workstation;
```

(17)
```
end discretionary_security;
```

(18) An audit file is an object which stores data associated with events that occur on one or more workstations. It may be associated with one or more workstations which share the same administration volume. The audit file associated with a workstation is the destination of an "audit" link from the workstation.

(19) The audit file contains auditing records, each of which records information concerning one event on the workstation. An auditing record has a general part and a part that depends on the event type of the event being audited.

(20) The general part, represented by the fields of the basic auditing record, is defined as follows:

(21) - USER: the identity of the user invoking the operation giving rise to the event;

(22) - TIME: the system time of the event;

(23) - WORKSTATION: the workstation on which the event takes place;

(24) - EVENT_TYPE: the event type of the event;

(25) - RETURN_CODE: FAILURE if the operation giving rise to the event terminates in an error, SUCCESS otherwise;

(26) - PROCESS: the process performing the operation giving rise to the event.

(27) Event-type-specific fields are defined as follows:

(28) - Events of type SELECT_AUDIT_EVENT are represented by basic auditing records;

(29) - For object auditing records, representing events of types WRITE, READ, ACCESS_CONTENTS, CHANGE_ACCESS_CONTROL_LIST, CHANGE_LABEL, WRITE_CONFIDENTIALITY_VIOLATION, WRITE_INTEGRITY_VIOLATION, READ_CONFIDENTIALITY_VIOLATION, READ_INTEGRITY_VIOLATION, SECURITY_ADMINISTRATION, and COVERT_CHANNEL:

(30) . OBJECT: the object on which the operation takes place.

(31) - For exploit auditing records, representing events of type EXPLOIT:

(32) . NEW_PROCESS: the process resulting from the exploitation of the object, e.g. if the operation has started execution of a program;

(33) . EXPLOITED_OBJECT: the object being exploited.

(34) - For information auditing records, representing events of type INFORMATION:

(35) . INFORMATION: the message associated with the event.

(36) - For copy auditing records, representing events of types COPY and CHANGE_IDENTIFICATION:

(37) . SOURCE: the object being copied from, or the old identification of the object;

(38) . DESTINATION: the object being copied to, or the new identification of the object.

(39) - For security auditing records, representing events of types USE_PREDEFINED_GROUP, and SET_USER_IDENTITY:

(40) . GROUP: the group being used, the user identifier being set or the user performing the audit selection.

(41) If, when writing to the audit file, the write fails because the audit file is unavailable for some reason, then the operation which caused the auditable event to occur waits until an audit file is made available, unless the calling process is acting with the predefined group PCTE_AUDIT. The means by which the audit file unavailability is notified to the operators of the PCTE installation is implementation-defined.

NOTES

(42) 1 The usage mode of the "audit" link type prevents any create or delete accesses. It is the role of an implementation-dependent bootstrap procedure to ensure that the audit file exists on a workstation when it is brought up. The audit data must be protected so that access to it is limited to users who are authorized for audit data.

(43) 2 No constraints on the label of the audit file are enforced by the system when the system writes to the audit file (i.e. it is up to the auditor to define it). When the system writes to the audit file, a bitwise write occurs but even in the case where this bitwise write results in a covert channel, it is not audited.

21.1.2 Audit selection criteria

(1) General_criterion = Selectable_event_type * Selected_return_code

(2) User_criterion = Selectable_event_type * Group_identifier

(3) Confidentiality_criterion = Selectable_event_type * Security_label

(4) Integrity_criterion = Selectable_event_type * Security_label

(5) Object_criterion = Selectable_event_type * Object_designator

(6) Audit_status = ENABLED | DISABLED

(7) Selection_criterion = General_criterion | Specific_criterion

(8) Specific_criterion = User_criterion | Confidentiality_criterion | Integrity_criterion | Object_criterion

(9) Removed_criterion = Selectable_event_type | Specific_criterion

(10) Selected_return_code = Return_code | ANY_CODE

(11) Criterion_type = GENERAL | USER_DEPENDENT | CONFIDENTIALITY_DEPENDENT | INTEGRITY_DEPENDENT | OBJECT_DEPENDENT

(12) General_criteria = **set of** General_criterion

(13) User_criteria = **set of** User_criterion

(14) Confidentiality_criteria = **set of** Confidentiality_criterion

(15) Integrity_criteria = **set of** Integrity_criterion

(16) Object_criteria = **set of** Object_criterion

(17) Criteria = General_criteria | User_criteria | Confidentiality_criteria | Integrity_criteria | Object_criteria

(18) Event types may be *selected* for auditing on a per workstation basis. When a selected event occurs, audit data is written to the audit file associated with the workstation where the event occurred. The event types CHANGE_IDENTIFICATION, SELECT_AUDIT_EVENT and SECURITY_ADMINISTRATION are always audited, regardless of the current selection criteria. A list of event types is in annex E.

(19) Selected events are only audited when auditing is *enabled* on the workstation. When auditing is *disabled*, only the event types that are always audited are audited.

(20) Events are selected on the basis of their types and either a return code, a user, an object, or a label. Each workstation maintains a set of *audit selection criteria*. The set of audit selection criteria is not persistent across workstation failure.

(21) Criteria of each type select events as follows:

(22) - General criterion: all events of the specified type and with the specified return code are selected for auditing, or if the specified return code is ANY_CODE then all events of that type are selected.

(23) - User-dependent criterion: all events of the specified type and being performed on behalf of the user identified by the group identifier are selected for auditing.

(24) - Confidentiality-dependent criterion: all events of the specified type that are performed on objects of the specified confidentiality label are selected for auditing.

(25) - Integrity-dependent criterion: all events of the specified type that are performed on objects of the specified integrity label are selected for auditing.

(26) - Object-dependent criterion: all events of the specified type that are performed on the specified object are selected for auditing.

21.2 Auditing operations

21.2.1 AUDIT_ADD_CRITERION

(1)
```
AUDIT_ADD_CRITERION (
    station     : Workstation_designator,
    criterion   : Selection_criterion
)
```

(2) AUDIT_ADD_CRITERION adds the criterion *criterion* to the audit selection criteria for the workstation *station*. Events of the type specified in *criterion* will then be audited on *station*, dependent on the type of *criterion* specified:

(3) - General criterion: The events are recorded on the basis of the return code of the operation generating the event. If the event type is already selected with the same return code, then the operation has no effect.

(4) - Confidentiality-dependent criterion: Events performed on objects of the specified confidentiality label are audited on *station*. If the event type and confidentiality label are already selected then the operation has no effect.

(5) - Integrity-dependent criterion: Events performed on objects of the specified integrity label are be audited on *station*. If the event type and integrity label are already selected then the operation has no effect.

(6) - Object-dependent criterion: Events performed on the specified object are audited on *station*. The object specified by *criterion* must be accessible. If the event type and object are already selected then the operation has no effect.

(7) - User-dependent criterion: Events performed by the specified user are audited on *station*. If the event type and user are already selected then the operation has no effect.

Errors

(8) For confidentiality-dependent criterion:
CONFIDENTIALITY_LABEL_IS_INVALID (security label specified by *criterion*)

(9) For object-dependent criterion:
DISCRETIONARY_ACCESS_IS_NOT_GRANTED (specified object, ATOMIC)

(10) For user-dependent criterion:
GROUP_IDENTIFIER_IS_INVALID (group identifier of *criterion*)
USER_IS_UNKNOWN (user specified by *criterion*)

(11) For integrity-dependent criterion:
INTEGRITY_LABEL_IS_INVALID (security label specified by *criterion*)

(12) OBJECT_IS_INACCESSIBLE (*station*, ATOMIC)

(13) PRIVILEGE_IS_NOT_GRANTED (PCTE_AUDIT)

(14) WORKSTATION_IS_UNKNOWN (*station*)

21.2.2 AUDIT_FILE_COPY_AND_RESET

(1)
```
AUDIT_FILE_COPY_AND_RESET (
    source          : Audit_file_designator,
    destination     : Audit_file_designator
)
```

(2) AUDIT_FILE_COPY_AND_RESET copies the audit file *source* into the audit file *destination*. The contents of *source* is cleared. No audit records are lost.

(3) This operation may not be invoked from within a transaction.

(4) Write locks of the default mode are obtained on *source*, on *destination*, and on the created and deleted links.

Errors

(5) ACCESS_ERRORS (*source*, ATOMIC, MODIFY, (READ_CONTENTS, WRITE_CONTENTS))

(6) ACCESS_ERRORS (*destination*, ATOMIC, MODIFY, WRITE_CONTENTS)

(7) OBJECT_IS_IN_USE_FOR_MOVE (*destination*)

(8) AUDIT_FILE_IS_NOT_ACTIVE (*source*)

(9) PRIVILEGE_IS_NOT_GRANTED (PCTE_AUDIT)

(10) PROCESS_IS_IN_TRANSACTION

21.2.3 AUDIT_FILE_READ

(1)
```
AUDIT_FILE_READ (
    audit_file    : Audit_file_designator
)
    records       : Audit_file
```

(2) AUDIT_FILE_READ reads the contents of the audit file *audit_file*, returning the result as a sequence of auditing records in *records*.

Errors

(3) ACCESS_ERRORS (*audit_file*, ATOMIC, READ, READ_CONTENTS)

21.2.4 AUDIT_GET_CRITERIA

(1)
```
AUDIT_GET_CRITERIA (
    station          : Workstation_designator,
    criterion_type   : Criterion_type
)
    criteria         : Criteria
```

(2) AUDIT_GET_CRITERIA returns the set of criteria of the type given by *criterion_type* that have been set for the workstation *station*. The returned set contains the event types that have been selected mapped to the return codes, mandatory labels, object designators, or user designators (depending on the *criterion_type)* associated with each event type.

(3) The set of criteria returned depends on the value of *criterion_type*:

(4) - GENERAL: the set of general criteria is returned.

(5) - CONFIDENTIALITY_DEPENDENT the set of confidentiality-dependent criteria is returned.

(6) - INTEGRITY_DEPENDENT: the set of integrity-dependent criteria is returned.

(7) - OBJECT_DEPENDENT: the set of object-dependent criteria is returned.

(8) - USER_DEPENDENT: the set of user-dependent criteria is returned.

Errors

(9) OBJECT_IS_INACCESSIBLE (*station,* ATOMIC)

(10) PRIVILEGE_IS_NOT_GRANTED (PCTE_AUDIT)

(11) WORKSTATION_IS_UNKNOWN (*station*)

21.2.5 AUDIT_RECORD_WRITE

(1)
```
AUDIT_RECORD_WRITE (
    text     : String
)
```

(2) AUDIT_RECORD_WRITE writes an information auditing record in the audit file *audit_file* of the local workstation. The INFORMATION field of the auditing record is specified by *text*.

Errors

(3) ACCESS_ERRORS (*audit_file,* ATOMIC, MODIFY, APPEND_CONTENTS)

(4) AUDIT_FILE_IS_NOT_ACTIVE (*audit_file*)

(5) LIMIT_WOULD_BE_EXCEEDED (MAX_AUDIT_INFORMATION_LENGTH)

21.2.6 AUDIT_REMOVE_CRITERION

(1)
```
AUDIT_REMOVE_CRITERION (
    station      : Workstation_designator,
    criterion    : Removed_criterion
)
```

(2) AUDIT_REMOVE_CRITERION removes the criterion *criterion* from the audit criteria of the workstation *station*.

(3) For a selectable event type, all general selection criteria with that event type are removed regardless of the return code specified.

(4) For a confidentiality-dependent criterion, events of the selected type performed on objects with the selected confidentiality label are no longer audited.

(5) For an integrity-dependent criterion, events of the selected type performed on objects with the selected integrity label are no longer audited.

(6) For an object-dependent criterion, events of the selected type performed on the selected object are no longer audited.

(7) For a user-dependent criterion, events of the selected type performed on behalf of the selected user are no longer audited.

Errors

(8) For confidentiality-dependent criterion:
CONFIDENTIALITY_CRITERION_IS_NOT_SELECTED (*criterion*)
CONFIDENTIALITY_LABEL_IS_INVALID (security label specified by *criterion*)

(9) For event type:
EVENT_TYPE_IS_NOT_SELECTED (*criterion*)

(10) For integrity-dependent criterion:
INTEGRITY_CRITERION_IS_NOT_SELECTED (*criterion*)
INTEGRITY_LABEL_IS_INVALID (security label specified by *criterion*)

(11) For object-dependent criterion:
DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*object*, ATOMIC)
OBJECT_CRITERION_IS_NOT_SELECTED (*criterion*)

(12) For user-dependent criterion:
GROUP_IDENTIFIER_IS_INVALID (group identifier of *criterion*)
USER_IS_UNKNOWN (user specified by *criterion*)
USER_CRITERION_IS_NOT_SELECTED (*criterion*)

(13) OBJECT_IS_INACCESSIBLE (*station*, ATOMIC)

(14) PRIVILEGE_IS_NOT_GRANTED (PCTE_AUDIT)

(15) WORKSTATION_IS_UNKNOWN (*station*)

21.2.7 AUDIT_SELECTION_CLEAR

(1)
```
AUDIT_SELECTION_CLEAR (
    station : Workstation_designator
)
```

(2) AUDIT_SELECTION_CLEAR removes all selected audit criteria from the workstation *station*.

Errors

(3) OBJECT_IS_INACCESSIBLE (*station*, ATOMIC)

(4) PRIVILEGE_IS_NOT_GRANTED (PCTE_AUDIT)

(5) WORKSTATION_IS_UNKNOWN (*station*)

21.2.8 AUDIT_SWITCH_OFF_SELECTION

(1)
```
AUDIT_SWITCH_OFF_SELECTION (
    station : Workstation_designator
)
```

(2) AUDIT_SWITCH_OFF_SELECTION disables auditing on the workstation *station*. Events on *station* will no longer be audited, except for the event types that are always audited.

(3) The current auditing selection criteria are maintained.

Errors

(4) OBJECT_IS_INACCESSIBLE (*station*, ATOMIC)

(5) PRIVILEGE_IS_NOT_GRANTED (PCTE_AUDIT)

(6) WORKSTATION_IS_UNKNOWN (*station*)

21.2.9 AUDIT_SWITCH_ON_SELECTION

(1)
```
AUDIT_SWITCH_ON_SELECTION (
    station : Workstation_designator
)
```

(2) AUDIT_SWITCH_ON_SELECTION enables auditing on the workstation *station*. Events on *station* will then be audited according to the current selection criteria. If auditing is already enabled, then the operation has no effect.

Errors

(3) OBJECT_IS_INACCESSIBLE (*station*, ATOMIC)

(4) PRIVILEGE_IS_NOT_GRANTED (PCTE_AUDIT)

(5) WORKSTATION_IS_UNKNOWN (*station*)

21.2.10 AUDITING_GET_STATUS

(1)
```
AUDITING_GET_STATUS (
    station : Workstation_designator
)
    status  : Audit_status
```

(2) AUDITING_GET_STATUS returns ENABLED if auditing is currently enabled on the workstation *station*, and DISABLED otherwise.

Errors

(3) OBJECT_IS_INACCESSIBLE (*station*, ATOMIC)

(4) PRIVILEGE_IS_NOT_GRANTED (PCTE_AUDIT)

(5) WORKSTATION_IS_UNKNOWN (*station*)

22 Accounting

22.1 Accounting concepts

22.1.1 Consumers and accountable resources

(1) Consumer_identifier = Natural

(2) Resource_identifier = Natural

(3) **sds** accounting:

(4) **import object type** system-object, system-process, system-common_root;

(5) **import attribute type** system-number;

(6) accounting_directory: **child type of** object **with**
link
known_consumer_group: (**navigate**) **existence link** (consumer_identifier: **natural**) **to** consumer_group;
known_resource_group: (**navigate**) **existence link** (resource_identifier: **natural**) **to** resource_group;
accounts_of: **implicit link to** common_root **reverse** accounts;
end accounting_directory;

(7) consumer_group: **child type of** object **with**
link
consumer_process: (**navigate**) **non_duplicated designation link** (number) **to** process;
end consumer_group;

(8) resource_group: **child type of** object **with**
link
resource_group_of: (**navigate**) **reference link** (number) **to** object **reverse** in_resource_group;
end resource_group;

(9) **extend object type** process **with**
link
consumer_identity: (**navigate**) **designation link to** consumer_group;
end process;

(10) **extend object type** object **with**
link
in_resource_group: (**navigate**) **reference link to** resource_group **reverse** resource_group_of;
end object;

(11) **extend object type** common_root **with**
link
accounts: (**navigate**) **existence link to** accounting_directory **reverse** accounts_of;
end common_root;

(12) **end** accounting;

(13) A consumer group is a group of consumer processes which are accounted together for their usage of accountable resources.

(14) A resource group is a group of *accountable resources*, the usage of which are accounted together. *Accountable resources* are files, pipes, devices, static contexts, workstations, SDSs, and message queues. There is a "resource_group_of" link from the resource group to each of its accountable resources.

(15) The accounting directory is an administrative object (see 9.1.2).

(16) Each consumer group and each resource group has an associated consumer identifier or resource identifier respectively which identifies it uniquely within the PCTE installation and is used in the construction of accounting records. These identifiers are key attributes of the links from the accounting directory to the consumer group and resource group respectively.

(17) A process may be associated with a consumer group, which is the destination of the "consumer_identity" link. If a process is not associated with a consumer group, accounting is still effective for that process.

22.1.2 Accounting logs and accounting records

(1) Accounting_log ::
RECORDS : **seq of** Accounting_record
represented by accounting_log

(2) Accounting_record = Workstation_accounting_record
| Static_context_accounting_record
| Sds_accounting_record
| Device_accounting_record
| File_accounting_record
| Pipe_accounting_record
| Message_queue_accounting_record
| Information_accounting_record

(3) Basic_accounting_record ::
SECURITY_USER : Group_identifier
ADOPTED_USER_GROUP : Group_identifier
CONSUMER_GROUP : [Consumer_identifier]
RESOURCE_GROUP : [Resource_identifier]
START_TIME : Time
KIND : Resource_kind

(4) Resource_kind = WORKSTATION | STATIC_CONTEXT | SDS | DEVICE | FILE | PIPE | MESSAGE_QUEUE | INFORMATION

(5) Workstation_accounting_record :: Basic_accounting_record && - - KIND = WORKSTATION
DURATION : Float
CPU_TIME : Float
SYS_TIME : Float

(6) Static_context_accounting_record :: Basic_accounting_record && - - KIND = STATIC_CONTEXT
DURATION : Float
CPU_TIME : Float
SYS_TIME : Float

(7) Sds_accounting_record = Basic_accounting_record - - KIND = SDS

(8) Device_accounting_record :: Basic_accounting_record && - - KIND = DEVICE
DURATION : Float
READ_COUNT : Natural
WRITE_COUNT : Natural
READ_SIZE : Natural
WRITE_SIZE : Natural

(9) File_accounting_record :: Basic_accounting_record && - - KIND = FILE
DURATION : Float
READ_COUNT : Natural
WRITE_COUNT : Natural
READ_SIZE : Natural
WRITE_SIZE : Natural

(10) Pipe_accounting_record :: Basic_accounting_record && - - KIND = PIPE
DURATION : Float
READ_COUNT : Natural
WRITE_COUNT : Natural
READ_SIZE : Natural
WRITE_SIZE : Natural

```
(11)  Message_queue_accounting_record :: Basic_accounting_record &&
                                                        - - KIND = MESSAGE_QUEUE
          OPERATION       : SEND | RECEIVE
          MESSAGE_SIZE    : Natural
(12)  Information_accounting_record :: Basic_accounting_record &&   - - KIND = INFORMATION
          INFORMATION     : String
(13)  sds accounting:
(14)  import object type system-workstation;
(15)  extend object type workstation with
      link
          has_log: (navigate) reference link to accounting_log reverse is_log_for;
      end workstation;
(16)  accounting_log: child type of object with
      contents accounting_log;
      link
          is_log_for: (navigate) reference link (number) to workstation reverse has_log;
      end accounting_log;
(17)  end accounting;
```

(18) An accounting log is an object associated with a workstation which is a server (see below). It has an "is_log_for" link to each associated workstation.

(19) An accounting record is a record of accountable resource usage by a process. Each usage has a *start event* when the usage is deemed to start and an *end event* when it is deemed to be complete. The accounting record is written to the accounting log associated with the workstation which is a server for the accountable resource at the end event. The accountable resource usages are as follows.

(20) - Use of the contents of a file, pipe, or device (KIND is FILE, PIPE or DEVICE respectively). The start event is when the process opens the contents (CONTENTS_OPEN); the end event is when the process next closes the contents (CONTENTS_CLOSE) or when the process terminates (PROCESS_TERMINATE).

(21) - Use of a static context or workstation associated with the process (KIND is STATIC_CONTEXT or WORKSTATION respectively). The start event is when the process is started (PROCESS_START or PROCESS_CREATE_AND_START); the end event is when the process terminates (PROCESS_TERMINATE).

(22) - Use of an SDS in the working schema of the process (KIND is SDS). The start event is when the process is started (PROCESS_START or PROCESS_CREATE_AND_START) or when a working schema containing the SDS is set (PROCESS_SET_WORKING_SCHEMA); the end event is when a new working schema is set (PROCESS_SET_WORKING_SCHEMA) or the process terminates (PROCESS_TERMINATE).

(23) - Sending a message to a message queue or receiving a message from a message queue (KIND is MESSAGE_QUEUE). The start and end events are the same: the sending or receiving of the message (MESSAGE_SEND_WAIT, MESSAGE_SEND_NO_WAIT, MESSAGE_RECEIVE_ WAIT, MESSAGE_RECEIVE_NO_WAIT).

(24) - Certain operations act as an end event followed by a start event for all started accounting resource usages by the calling process; they are PROCESS_SET_CONSUMER_

IDENTITY, PROCESS_UNSET_CONSUMER_IDENTITY, PROCESS_SET_USER, and PROCESS_ADOPT_GROUP.

(25) - Certain operations act as an end event for certain started accounting resource usages by the calling process; they are WORKSTATION_DISCONNECT for accountable resources on volumes of the workstation; VOLUME_UNMOUNT for accountable resources on the volume; PROCESS_TERMINATE and ACTIVITY_ABORT for started accounting resource usages by the process. ACCOUNTING_OFF acts as an end event for all accountable resources on volumes of the workstation, for all processes.

(26) - Certain events may be end events for started accounting resource usages by the calling process; they are failure of the execution site of the process, and the volume on which the accountable resource resides becoming inaccessible. In the case of static context and SDS resources on inaccessible volumes, whether such events are end events or not is implementation-defined.

(27) When a resource is made accountable after its usage has started, or is removed from being accountable before its usage has ended, if such usage is recorded, and if so how, are implementation-defined.

(28) If an accountable resource becomes inaccessible to the process, this counts as an end event for the usage of that resource.

(29) A process may also write accounting records via ACCOUNTING_RECORD_WRITE (KIND is INFORMATION).

(30) A workstation is a *server* for an accountable resource if the accountable resource resides on a volume mounted on a device controlled by the workstation, and the workstation is associated with an accounting log and accounting is enabled on the workstation.

(31) The information in an accounting record depends on the kind of accountable resource involved. Each accounting record has a fixed part and a resource specific part. The fields of the accounting record are set as follows.

(32) - Basic accounting record (fixed part):

(33) . SECURITY_USER: the group identifier of the user identity of the process;

(34) . ADOPTED_USER_GROUP: the group identifier of the adopted user group of the process;

(35) . CONSUMER_GROUP: the exact identifier of the consumer group of the process;

(36) . RESOURCE_GROUP: the exact identifier of the resource group of the accountable resource;

(37) . START_TIME: the time by the system clock at the start event of the usage of the accountable resource;

(38) . DURATION: the duration of the usage of the accountable resource, in seconds, from the start event to the end event;

(39) . INFORMATION: free for use by tools writing accounting records into the accounting log via ACCOUNTING_RECORD_WRITE; absent in other accounting records.

(40) - Workstation accounting record and static context accounting record:

(41) . CPU_TIME: the consumption of processor time in seconds by the process during the usage of the workstation or static context;

(42) . SYS_TIME: the consumption of system time in seconds by the process during the usage of the workstation or static context.

(43) - Device accounting record, File accounting record, or Pipe accounting record:

(44) . READ_COUNT: number of read operations by the process from the device, file, or pipe during the usage;

(45) . WRITE_COUNT: number of write operations by the process to the device, file, or pipe during the usage;

(46) . READ_SIZE: total size in octets of data read by the process from the device, file, or pipe during the usage;

(47) . WRITE_SIZE: total size in octets of data written by the process to the device, file, or pipe during the usage.

(48) A read operation for device accounting purposes is a CONTENTS_READ only. A write operation for device accounting purposes is CONTENTS_WRITE or CONTENTS_TRUNCATE.

(49) - Message queue accounting record:

(50) . OPERATION: whether the usage was to send or to receive a message;

(51) . MESSAGE_SIZE: the size in octets of the message sent or received.

(52) The structure of the accounting log is implementation-defined.

(53) If, when writing to an accounting log, the write fails because the accounting log is unavailable, then the operation which caused the accountable event to occur waits until an accounting log is made available. The means by which the accounting log unavailability is notified to the operators of the PCTE installation is implementation-defined. When an end event occurs, the completed accounting record is not lost. If there is an abnormal closedown of the workstation before the end event of the usage of accountable resources, the extent to which the completion of such usages are recorded in further accounting records is implementation-defined.

NOTES

(54) 1 It is intended that accounting logs are persistent across workstation failures, and that modifications to an accounting log are not subject to the transaction rollback mechanism (i.e. updates to an accounting log can never be discarded).

(55) 2 The start of accountable usage for a resource may be when it is defined as an accountable resource, and the end of accountable usage may be when it is defined to be no longer an accountable resource. While the optimum implementation is to use the start and end of accountability of the resource as the start and end events of usage for resources which are in use at the time of change, it is recognized that this may cause unnecessary complication and inefficiency. An implementation must therefore specify how this situation is handled.

(56) 3 No constraints on the label of the accounting log are enforced by the system when it writes accounting records, although a bitwise write occurs. Whether this bitwise write, when it gives rise to a covert channel, is audited or not, is implementation-defined.

22.2 Accounting administration operations

22.2.1 ACCOUNTING_LOG_COPY_AND_RESET

(1)
```
ACCOUNTING_LOG_COPY_AND_RESET (
    source_log          : Accounting_log_designator,
    destination_log     : Accounting_log_designator
)
```

(2) ACCOUNTING_LOG_COPY_AND_RESET appends the contents of the accounting log *source_log* to the contents of the accounting log *destination_log*. The contents of *source_log* is then reset, i.e. the contents of *source_log* is now empty.

(3) There is no loss of accounting records.

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

Errors

(5) ACCESS_ERRORS (*source_log*, ATOMIC, MODIFY, (READ_CONTENTS, WRITE_CONTENTS))

(6) ACCESS_ERRORS (*destination_log*, ATOMIC, MODIFY, APPEND_CONTENTS)

(7) ACCOUNTING_LOG_IS_NOT_ACTIVE (*source_log*)

(8) OBJECT_IS_IN_USE_FOR_MOVE (*destination_log*)

(9) PROCESS_IS_IN_TRANSACTION

22.2.2 ACCOUNTING_LOG_READ

(1)
```
ACCOUNTING_LOG_READ (
    log          : Accounting_log_designator
)
    records      : Accounting_log
```

(2) ACCOUNTING_LOG_READ returns the sequence of accounting records in the accounting log *log*.

Errors

(3) ACCESS_ERRORS (*log*, ATOMIC, READ, READ_CONTENTS)

22.2.3 ACCOUNTING_OFF

(1)
```
ACCOUNTING_OFF (
    station : Workstation_designator
)
```

(2) ACCOUNTING_OFF disables the accounting mechanism on the workstation *station*.

(3) The "has_log" link from the object *station* (if there is one) and its reverse "is_log_for" link are deleted.

(4) Write locks of the default mode are obtained on the deleted link.

Errors

(5) ACCESS_ERRORS (accounting log of *station*, ATOMIC, MODIFY, WRITE_LINKS)

(6) ACCESS_ERRORS (*station*, ATOMIC, MODIFY, WRITE_LINKS)

(7) WORKSTATION_IS_UNKNOWN (*station*)

22.2.4 ACCOUNTING_ON

(1)
```
ACCOUNTING_ON (
    log      : Accounting_log_designator,
    station  : Workstation_designator
)
```

(2) ACCOUNTING_ON enables the accounting mechanism on the workstation *station* with accounting log *log*.

(3) A "has_log" link, reversed by an "is_log_for" link, is created from *station* to *log*.

(4) Write locks of the default mode are obtained on the created links.

Errors

(5) ACCESS_ERRORS (*log*, ATOMIC, MODIFY, APPEND_CONTENTS)

(6) ACCESS_ERRORS (*station*, ATOMIC, MODIFY, APPEND_LINKS)

(7) LINK_EXISTS (*station*, "has_log" link)

(8) OBJECT_IS_NOT_ON_ADMINISTRATION_VOLUME (*log*, *station*)

(9) WORKSTATION_IS_UNKNOWN (*station*)

(10) NOTE - The error LINK_EXISTS indicates that accounting was already enabled.

22.2.5 ACCOUNTING_RECORD_WRITE

(1)
```
ACCOUNTING_RECORD_WRITE (
    log         : Accounting_log_designator,
    information : String
)
```

(2) ACCOUNTING_RECORD_WRITE appends a basic accounting record to the accounting log *log*.

(3) The fields of the accounting record are set as follows:

(4) - SECURITY_USER is set to the group identifier of the destination of the "user_identity" link of the calling process;

(5) - ADOPTED_USER_GROUP is set to the group identifier of the adopted user group of the calling process;

(6) - CONSUMER_GROUP is set to the exact identifier of the destination of the "consumer_identity" link of the calling process;

(7) - RESOURCE_GROUP is set to null;

(8) - KIND is set to INFORMATION;

(9) - START_TIME is set to the current system time;

(10) - INFORMATION is set to the parameter *information*.

Errors

(11) ACCESS_ERRORS (*log*, ATOMIC, MODIFY, APPEND_CONTENTS)

(12) ACCOUNTING_LOG_IS_NOT_ACTIVE (*log*)

(13) LIMIT_WOULD_BE_EXCEEDED (MAX_ACCOUNT_INFORMATION_LENGTH)

22.2.6 CONSUMER_GROUP_INITIALIZE

(1)
```
CONSUMER_GROUP_INITIALIZE (
    group      : Consumer_group_designator
)
    identifier : Consumer_identifier
```

(2) CONSUMER_GROUP_INITIALIZE establishes the object *group* as a known consumer group. A "consumer_group" link is created from the accounting directory to *group*. The key of this link is set to a unique value, which is guaranteed never to be re-used as a consumer group identifier, and is returned in *identifier*.

(3) Write locks of the default mode are obtained on the created links.

Errors

(4) ACCESS_ERRORS (*group*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(5) ACCESS_ERRORS (the accounting directory, ATOMIC, MODIFY, APPEND_LINKS)

(6) CONSUMER_GROUP_IS_KNOWN (*group*)

22.2.7 CONSUMER_GROUP_REMOVE

(1)
```
CONSUMER_GROUP_REMOVE (
    group   : Consumer_group_designator
)
```

(2) CONSUMER_GROUP_REMOVE removes the consumer group *group* from the set of known consumer groups.

(3) No process must currently use the "consumer_identity" associated with that consumer group (i.e. the object *group* must not have any "consumer_process" links).

(4) The "known_consumer_group" link from the accounting directory to *group* is deleted.

(5) If it is the only existence link to the object *group* and if there are no composition links to *group*, then *group* is also deleted. In that case, the "object_on_volume" link to *group* from the volume on which *group* was residing is also deleted.

(6) Write locks of the default mode are obtained on the deleted links and on *group* if it is deleted.

Errors

(7) ACCESS_ERRORS (*group*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(8) ACCESS_ERRORS (the accounting directory, ATOMIC, MODIFY, WRITE_LINKS)

(9) If conditions hold for the deletion of the *group:*
ACCESS_ERRORS (*group*, COMPOSITE, MODIFY, DELETE)

(10) CONSUMER_GROUP_IS_IN_USE (*group*)

(11) CONSUMER_GROUP_IS_UNKNOWN (*group*)

(12) OBJECT_HAS_LINKS_PREVENTING_DELETION (*group*)

(13) OBJECT_IS_IN_USE_FOR_DELETE (*group*)

22.2.8 RESOURCE_GROUP_ADD_OBJECT

(1)
```
RESOURCE_GROUP_ADD_OBJECT (
    object  : Object_designator,
    group   : Resource_group_designator
)
```

(2) RESOURCE_GROUP_ADD_OBJECT defines the object *object* to be an accountable resource.

(3) An "in_resource_group" link is created from *object* to *group*. Its reverse "resource_group_of" link is keyed by the next unused value (see 23.1.2.7).

(4) Write locks of the default mode are obtained on the created links.

Errors

(5) ACCESS_ERRORS (*object*, ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*group*, ATOMIC, MODIFY, APPEND_LINKS)

(7) OBJECT_IS_ALREADY_IN_RESOURCE_GROUP (*object*, *group*)

(8) OBJECT_IS_NOT_ACCOUNTABLE_RESOURCE (*object*)

(9) RESOURCE_GROUP_IS_UNKNOWN (*group*)

22.2.9 RESOURCE_GROUP_INITIALIZE

(1)
```
RESOURCE_GROUP_INITIALIZE (
    group       : Resource_group_designator
)
    identifier  : Resource_identifier
```

(2) RESOURCE_GROUP_INITIALIZE establishes the object *group* as a known resource group. A "resource_group" link is created from the accounting directory to *group*. The key of this link is set to a unique value, which is guaranteed never to be re-used as a resource group identifier, and is returned in *identifier*.

(3) Write locks of the default mode are obtained on the created links.

Errors

(4) ACCESS_ERRORS (*group*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(5) ACCESS_ERRORS (the accounting directory, ATOMIC, MODIFY, APPEND_LINKS)

(6) RESOURCE_GROUP_IS_KNOWN (*group*)

22.2.10 RESOURCE_GROUP_REMOVE

(1)
```
RESOURCE_GROUP_REMOVE (
    group   : Resource_group_designator
)
```

(2) RESOURCE_GROUP_REMOVE removes the resource group *group* from the set of known resource groups.

(3) *group* must have no "resource_group_of" links to accountable resources.

(4) The "known_resource_group" link from the accounting directory to *group* is deleted. If it is the only existence link to *group* and if there are no composition links to *group* then *group* is also deleted. In that case, the "object_on_volume" link to *group* from the volume on which the *group* was residing is also deleted.

(5) Write locks of the default mode are obtained on the deleted links and on *group* if it is deleted.

Errors

(6) ACCESS_ERRORS (*group*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(7) ACCESS_ERRORS (the accounting directory, ATOMIC, MODIFY, WRITE_LINKS)

(8) If conditions hold for the deletion of the *group:*
ACCESS_ERRORS (*group*, COMPOSITE, MODIFY, DELETE)

(9) OBJECT_HAS_LINKS_PREVENTING_DELETION (*group*)

(10) OBJECT_IS_IN_USE_FOR_DELETE (*group*)

(11) RESOURCE_GROUP_IS_UNKNOWN (*group*)

22.2.11 RESOURCE_GROUP_REMOVE_OBJECT

(1)
```
RESOURCE_GROUP_REMOVE_OBJECT (
    object  : Object_designator,
    group   : Resource_group_designator
)
```

(2) RESOURCE_GROUP_REMOVE_OBJECT removes the object *object* as an accountable resource from the resource group *group*.

(3) The "resource_group_of" link and the "in_resource_group" reverse link between *object* and *group* are deleted.

(4) Write locks of the default mode are obtained on the deleted links.

Errors

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

(6) ACCESS_ERRORS (*group*, ATOMIC, MODIFY, WRITE_LINKS)

(7) OBJECT_IS_NOT_IN_RESOURCE_GROUP (*object*, *group*)

(8) RESOURCE_GROUP_IS_UNKNOWN (*group*)

22.3 Consumer identity operations

22.3.1 PROCESS_SET_CONSUMER_IDENTITY

(1)
```
PROCESS_SET_CONSUMER_IDENTITY (
    group   : Consumer_group_designator
)
```

(2) PROCESS_SET_CONSUMER_IDENTITY sets the consumer identity of the calling process, by creating a "consumer_identity" link from the calling process to *group* and a complementary "consumer_process" link from *group* to the calling process.

(3) If the calling process already has a "consumer_identity" link, that link and its complementary "consumer_process" link are deleted.

Errors

(4) ACCESS_ERRORS (*group*, ATOMIC, SYSTEM_ACCESS)

(5) CONSUMER_GROUP_IS_UNKNOWN (*group*)

(6) DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*group*, ATOMIC, EXPLOIT_CONSUMER_IDENTITY)

(7) If *process* is the calling process:
VOLUME_IS_FULL (calling process)

22.3.2 PROCESS_UNSET_CONSUMER_IDENTITY

(1)
```
PROCESS_UNSET_CONSUMER_IDENTITY (
)
```

(2) PROCESS_UNSET_CONSUMER_IDENTITY suppresses the consumer identity of the calling process by deleting the "consumer_identity" link, if any, from the calling process and a complementary "consumer_process" link. If the calling process has no "consumer_identity" link, the operation has no effect.

Errors

(3) None.

23 Common binding features

23.1 Mapping of types

23.1.1 Mapping of predefined PCTE datatypes

(1) *Predefined PCTE datatypes* are the datatypes used as or to form the types of parameters and results of operations defined in this ECMA Standard which are not constructed from other PCTE datatypes. They are: Boolean, Natural, Integer, Float, Time, Text, and Octet. In order to define the possible values of these PCTE datatypes in a way which allows sensible binding decisions to be made, use is made of ISO/IEC 11404 which defines a comprehensive set of *LI datatypes* in abstract terms as sets of *values* and of associated *characterizing operations*. A language binding must define a mapping from these LI datatypes to binding language datatypes, which must ensure that all the characterizing operations are supported. In most cases the binding language supports them, but if not then the binding must supply them separately.

23.1.1.1 Boolean values

(1) The PCTE datatype Boolean is mapped to the primitive LI datatype boolean. This has 2 values, true and false; it is unordered. The characterizing operations are Equal, Not, And, Or.

23.1.1.2 Integer values

(1) The PCTE datatype Integer is mapped to the primitive LI datatype integer or to a generated LI datatype integer range (lowerbound .. upperbound) with appropriate bounds.

(2) integer is a primitive LI datatype comprising the mathematical integers (positive, negative, and zero); its characterizing operations are Equal, Add, Multiply, Negate, NonNegative, and InOrder.

(3) range is a subtype generator which creates a subtype of an ordered LI datatype within given bounds. The characterizing operations of the subtype are the same as those of the parent type.

(4) The bounds are required to satisfy:

(5) - upperbound = MAX_INTEGER_ATTRIBUTE;

(6) - lowerbound = MIN_INTEGER_ATTRIBUTE.

23.1.1.3 Natural values

(1) The PCTE datatype Natural is mapped to a generated LI datatype integer range (0 .. upperbound) with appropriate upper bound.

(2) The characterizing operations are as for the LI datatype integer except for Negate (and NonNegative which is not required as it is always true).

(3) The upper bound is required to satisfy:

(4) - upperbound = MAX_NATURAL_ATTRIBUTE.

23.1.1.4 Float values

(1) The PCTE datatype Float is mapped to a primitive LI datatype real (radix, factor), where radix and factor are integers with radix > 1, or to a generated LI datatype real (radix, factor) range (lowerbound .. upperbound). float (radix, factor) is a subset of the mathematical datatype of real numbers with precision of at least $radix^{-factor}$. The characterizing operations are Equal, Add, Multiply, Negate, Reciprocal, and InOrder (and Promote which is not required as there is no PCTE datatype corresponding to the LI datatype rational).

(2) The values radix, factor, lowerbound, and upperbound must satisfy the following:

(3) - upperbound = MAX_FLOAT_ATTRIBUTE;

(4) - lowerbound = MIN_FLOAT_ATTRIBUTE;

(5) - $radix^{-factor} \leq 10^{-MAX\_DIGITS\_FLOAT\_ATTRIBUTE}$;

(6) - the smallest positive and negative numbers representable are ± SMALLEST_FLOAT_ATTRIBUTE.

23.1.1.5 Time values

(1) The PCTE datatype Time is mapped to a primitive LI datatype time (second) or time (second, radix, factor), where radix and factor are integers with radix > 1, or to a subtype time (second) range (lowerbound .. upperbound) or time (second, radix, factor) range (lowerbound .. upperbound) with appropriate bounds. This is a datatype representing moments in time to a resolution of 1 second, or to a fraction of a second defined by $radix^{-factor}$. The characterizing operations are Equal, InOrder, Difference, Extend (to a more precise resolution), and Round (to a less precise resolution).

(2) The binding mapping must respect the limits:

(3) - factor ≥ 0 (resolution at worst 1 second);

(4) - upperbound = MAX_TIME_ATTRIBUTE;

(5) - lowerbound = MIN_TIME_ATTRIBUTE.

23.1.1.6 Octet, character, and text values

(1) The PCTE datatype Octet is mapped to the LI datatype octet = new integer range (0 .. 255). The characterizing operation is Equal (and Select and Replace (from array) which are not required as there is no PCTE datatype corresponding to bit).

(2) Octet values have no intrinsic graphical representation. When a graphical representation is required, the graphical representation of the PCTE datatype Character is used.

(3) The PCTE datatype Character comprises the human-readable characters of one or more character sets selected by the PCTE implementation. In a character set, a single character may be represented by a single byte or by more than one byte.

(4) The PCTE Datatype Text comprises sequences of characters. Characters of more than one character set may exist in a single text value. The method of identifying the character set of a given character is implementation-defined.

NOTES

(5) 1 By the definition of the PCTE datatype Character, an octet may be part of a character of a multi-byte character set. Therefore, it may occur that an octet which is identical to an octet associated with a character of a single-byte character set is part of a character of a multi-byte character set. Even if such an octet is identical to an octet associated with a special character (e.g. '/', '$', '#', '.' in a pathname), a PCTE implementation should not interpret the octet as such a special character.

(6) 2 For the definitions of the terms 'octet' and 'character set' see ISO/IEC 10646-1. For the definition of the term 'byte' see ISO/IEC 2022.

23.1.1.7 Token values

(1) The PCTE datatype Token is used only as the field type of a single anonymous field of a record type, called a *private PCTE datatype*, as e.g. in:

```
User_defined_message_type :: Token
```

This ensures that the values of type User_defined_message_type are distinct from the values of all other types.

(2) Except for designators and nominators (for which see 23.1.2), each private PCTE datatype is mapped to a distinct LI datatype new private (*length*), where private is an LI datatype defined by

```
type private (length: NaturalNumber) = new array (1 .. length) of (bit)
```

For each such type *length* is a binding-defined positive integer.

23.1.1.8 Enumeration values

(1) *Enumerated PCTE datatypes* are unions of VDM-SL enumeration types, each comprising a single enumerated value. An enumerated PCTE datatype:

```
VALUE1 | VALUE2 | ...
```

is mapped to an LI enumerated datatype enumerated (value1, value2, ...) with corresponding values. The characterizing operation is Equal (and InOrder and Successor which are not required as an enumerated PCTE datatype is unordered).

23.1.2 Mapping of designators and nominators

(1) This clause defines the constraints on the PCTE datatypes used in bindings for referring to objects, attributes, links, and types, and types in SDS.

(2) These datatypes are object designators, attribute designators, link designators, type nominators, type nominators in SDS, and actual keys, when used as or in parameters or results of operations defined in clauses 9 to 22; the corresponding binding types are called object references, attribute references, link references, type references, type names in SDS, and keys respectively. Object references, attribute references, link references, and type references are binding-defined datatypes supported by operations defined in 23.2 to 23.4. Type names in SDS and keys are text values with an internal syntax, defined in the BSI metasyntactic notation.

(3) For all designators and nominators, except link designators used in the creation of links, the entity designated must always exist. The process of identifying the entity from a reference is

called *evaluation* of the reference; this is implicitly performed by all operations in clauses 9 to 22 for unevaluated references. This process can give rise to error conditions, which are defined in 23.1.2.1 to 23.1.2.4.

(4) NOTE - The mapping of the PCTE datatypes used in the bindings is summarized in table 11, with the operations used to create values of the types. Text creation is binding-defined.

Table 11 - Mapping of PCTE datatypes to common binding datatypes

PCTE datatype	Binding datatype	Created by
object designator	object reference	OBJECT_REFERENCE_SET_ABSOLUTE, OBJECT_REFERENCE_SET_RELATIVE
attribute designator	attribute reference	TYPE_REFERENCE_SET
link designator	link reference	LINK_REFERENCE_SET
type nominator	type reference	TYPE_REFERENCE_SET
type nominator in SDS	type name in SDS	text creation
actual key	key	text creation

23.1.2.1 References

(1)
```
X_reference = Internal_X_reference | External_X_reference
```

(2)
```
External_X_reference ::
    NAME          : X_name
    EVALUABILITY  : Boolean
```

(3)
```
Evaluation_point = NOW | FIRST_USE | EVERY_USE
```

(4)
```
Evaluation_status = INTERNAL | EXTERNAL
```

(5)
```
Reference_equality = EQUAL_REFS | UNEQUAL_REFS | EXTERNAL_REFS
```

(6) References are an abstract datatype characterized by the operations of 23.2 to 23.4; the above VDM-SL type definition of an X_reference, where 'X' is 'object', 'attribute', 'link', or 'type', is for expository purposes only and need not be mapped explicitly in a binding.

(7) References provide two ways of designating an object, attribute, link, or type: by a name (an *external reference*), and by a handle (an *internal reference*). The *evaluation status* of a reference is external or internal accordingly. An object, attribute, link, or type is accessed from a reference by *evaluating* it. The syntax of external references and the structure of internal references are defined in 23.1.2.2 to 23.1.2.5 inclusive. An external reference is evaluated by first *pre-evaluating* it to give an internal reference; pre-evaluation in general evaluates the reference as far as possible in the absence of any other information. The internal reference is then evaluated in the context of any other required information to complete the process.

(8) The evaluability applies only to external references; it is **true** if the reference is to be converted to an internal reference when next pre-evaluated by an operation of clauses 9 to 22 and **false** otherwise. Evaluation points are used as parameters of operations returning references to indicate the evaluation status and evaluability required: NOW indicates an internal reference, FIRST_USE an external reference with evaluability **true**; and EVERY_USE an external reference with evaluability **false**.

(9) The evaluation of a reference takes place during the successful execution of an operation of clauses 9 to 22 of which the reference is a parameter. Operations in clause 23 do not as a rule evaluate their reference parameters, though in some cases (where stated) external reference parameters are pre-evaluated.

(10) References returned by any of the operations in clauses 9 to 22 are always internal.

23.1.2.2 Object references

(1)
```
Internal_object_reference :: Token
```

(2) An object reference identifies an object. The syntax of object names (also called *pathnames*) is as follows:

(3)
```
pathname = referenced object name, [ '/', relative pathname ] | ['$current_object', '/' ],
    relative pathname;
```

(4)
```
relative pathname = link name, {'/', link name};
```

(5)
```
referenced object name = '$', key string value | alias;
```

(6) For link references see 23.1.2.4.

(7) Pre-evaluation of an external object reference is the same as evaluation, and evaluation of an internal object reference is a null process.

(8) A relative pathname specifies a chain of links starting from a given origin object; the first link is specified by the origin object and the first link name; the second by the destination of the first link and the second link name, and so on. Finally the relative pathname specifies the destination object of the final link.

(9) A pathname with no relative pathname specifies the same object as the referenced object name. A pathname with a relative pathname specifies the final destination object given by the object specified by the referenced object name and the relative pathname, as just described.

(10) A pathname which consists only of a relative pathname is equivalent to a pathname starting from the current object and following the specified relative pathname, as just described.

(11) A referenced object name of the first form specifies the destination of the "referenced_object" link from the calling process with the key given by the key string value. The key is a string which is the referenced object name. If the link to reference object is omitted from an external object designator, the default reference object is ".".

(12) For practical purposes, *aliases* are provided for the most commonly used referenced objects; see table 12.

Table 12 - Aliases of referenced objects

Alias	Key of referenced object	Meaning
"$"	"self"	The current process object
"#"	"static_context"	The static context of the current process
"_"	"common_root"	The common root of the PCTE installation
"~"	"home_object"	An object conventionally associated with each user, called "home". The type of home is not predefined.
"."	"current_object"	An object conventionally chosen for the interpretation of a pathname without a starting referenced object name (see above) and providing the conventional notion of a current directory.

(13) When an object reference is evaluated, the constituent pathname, if any, is evaluated. The evaluation of the pathname involves the evaluation of the link names in the pathname.

(14) The visible types are those that are visible at the time of calling an operation. Thus even if an internal object reference is used in an operation, the visible types are those that are visible when the operation is called rather than when the object reference is evaluated.

(15) Evaluation of an external object reference *reference* may give rise to the following errors, which can therefore occur in any operation which has an object designator as parameter or result.

(16) ACCESS_ERRORS (object identified by *reference*, ATOMIC, READ, NAVIGATE)

(17) For each link name *link* in the relative pathname:
ACCESS_ERRORS (origin object of *link*, ATOMIC, SYSTEM_ACCESS)
LINK_DESTINATION_DOES_NOT_EXIST (*link*)
LINK_DESTINATION_IS_NOT_VISIBLE (*link*)
USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (origin object of *link*, *link*, NAVIGATE)
Errors arising from evaluation of *link* (see 23.1.2.4)

(18) LINK_DESTINATION_DOES_NOT_EXIST ("referenced_object" link from the calling process identified by the referenced object name of *reference*)

(19) REFERENCE_CANNOT_BE_ALLOCATED

(20) REFERENCE_NAME_IS_INVALID (*reference*)

(21) REFERENCED_OBJECT_IS_UNSET (*reference*)

(22) Any use of an object reference *reference* as parameter of an operation may additionally raise the following errors, whatever its evaluation status.

(23) OBJECT_IS_OF_WRONG_TYPE (*reference*)

(24) OBJECT_REFERENCE_IS_INVALID (*reference*)

(25) OBJECT_REFERENCE_IS_UNSET (*reference*)

23.1.2.3 Attribute references

(1) Internal_attribute_reference :: Internal_type_reference

(2) An attribute reference identifies an attribute relative to a given object or link. It may therefore identify different attributes depending on the object or link. The syntax of attribute names is as follows:

(3) attribute name = type name;

(4) Pre-evaluation of an external attribute reference consists in evaluating the type name (see 23.1.2.5) as an attribute type name to give an internal type reference. Evaluation of an internal attribute reference consists in identifying the attribute of the given object or link with that attribute type; as no two attributes of an object or link may have the same attribute type, evaluation is unambiguous.

(5) Evaluation of an attribute reference *reference* may give rise to the following errors:

(6) ATTRIBUTE_TYPE_IS_NOT_VISIBLE (*reference*)

(7) If the operation does not delete the attribute
ATTRIBUTE_TYPE_OF_LINK_TYPE_IS_NOT_APPLIED (*reference*)
ATTRIBUTE_TYPE_OF_OBJECT_TYPE_IS_NOT_APPLIED (*reference*)

(8) Errors arising from evaluation of the attribute type reference (see 23.1.2.5).

23.1.2.4 Link references

(1)
```
Internal_link_reference ::
    ACTUAL_LINK_TYPE    : [Internal_type_reference]
    KEY                 : [Key]
```

(2) An internal link type reference must contain an actual link type or a key (or both). A key is a text value obeying the syntax of a key (see 23.1.2.7).

(3) A link reference identifies a link in the context of its origin. It may therefore identify different links depending on the origin. The syntax of link names is as follows:

(4) link name = cardinality one link name | cardinality many link name;

(5) cardinality one link name = '.', type name;

(6) cardinality many link name = key, '.', [type name] | key;

(7) The *canonical form* of a link name is with a type identifier for link type name, and no '+', '++', or '-' key attribute values in the key.

(8) The second form of cardinality many link name is allowed only if the key consists of a single key attribute value, or if the rightmost key attribute value of the key does not obey the syntax of a type name.

(9) The pre-evaluation of an external link reference is as follows. If the link name contains a type name, that type name is evaluated as a link type name (see 23.1.2.5) to give the actual link type. If not, the internal link reference has no actual link type and the determination of the actual link type is deferred. For an external reference with a cardinality one link name, the internal link reference has no key; for an external link reference with a cardinality many link name, the key of the internal link reference is the same as the key of the link name.

(10) The evaluation of an internal link reference is as follows:

(11) - If the link reference does not contain an actual link type, the preferred link type of the origin is used as actual link type.

(12) - If there is no key, the link reference identifies the cardinality one link with the given origin and the actual link type.

(13) - If there is a key, it is evaluated in the context of the actual link type to yield an actual key, as described in 23.1.2.7. The identified link is the cardinality many link with the given origin, the actual link type, and the actual key.

(14) Pre-evaluation of a link reference *reference* may give rise to the following errors.

(15) LIMIT_WOULD_BE_EXCEEDED (MAX_KEY_VALUE)

(16) LIMIT_WOULD_BE_EXCEEDED (MAX_KEY_SIZE)

(17) LIMIT_WOULD_BE_EXCEEDED (MAX_LINK_NAME_SIZE)

(18) Errors arising from evaluation of the link type reference (see 23.1.2.5)

(19) Evaluation of a link reference *reference* in the context of the origin *origin* may additionally give rise to the following errors, whatever the evaluation status of *reference*.

(20) If any '+' or '++' key attribute values are evaluated:
DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*origin*, ATOMIC, READ_LINKS)

(21) If any '-' key attribute values are evaluated:
DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*origin*, ATOMIC, READ_ATTRIBUTES)

(22) KEY_IS_BAD (*origin, reference*)

(23) LINK_DOES_NOT_EXIST (*origin, reference*)

(24) LINK_TYPE_IS_NOT_APPLIED_TO_OBJECT_TYPE (object type of *origin*, link type reference of *reference*)

(25) If the link type name is not supplied:
PREFERENCE_DOES_NOT_EXIST (*origin, reference*)

(26) Errors arising from evaluation of the preferred link type reference (see 23.1.2.5).

23.1.2.5 Type references

(1)
```
Internal_type_reference :: Token
```

(2) A type reference identifies a type. The syntax of a type name is as follows:

(3)
```
type name = local name | full type name | type identifier;
```

(4)
```
local name = name;
```

(5)
```
full type name = sds name, '-', local name;
```

(6)
```
sds name = name;
```

(7)
```
type identifier = '_', string;
```

(8) Pre-evaluation of an external type reference is the same as evaluation, and evaluation of an internal type reference is a null process.

(9) A full type name identifies a type with the given local name in the SDS specified by the SDS name which is a member of the sequence of SDS names in the current working schema. A type

name which is just a local name identifies the type in the current working schema. If derived from a local name, the evaluation of the type reference yields the first type in working schema (in the sequence of SDS names in the current working schema) with that local name as its local name.

(10) A type identifier is a string with first character '_'; the syntax and interpretation of the rest of the string are implementation-defined. The value of the "type_identifier" attribute of a "type" object (see 10.1.2) is the corresponding type identifier without the initial '_' character.

(11) If a type name or a link name is returned by an operation rather than a type reference or a link reference, the type name or link name is returned as a type name if the type is visible and named in the current working schema, and as a type identifier otherwise. When a type name is returned it is the local name of the first named associated type in SDS in the sequence of SDS names in the working schema, provided that this local name does not occur earlier in the sequence of SDS names for another type. In the latter case, the full type name, i.e. prefixed by an SDS name, of the first associated type in SDS in the sequence of SDS names in the working schema is returned.

(12) The use of a type identifier as or as part of an input parameter is allowed if the type is visible, or if the predefined program group PCTE_CONFIGURATION is effective for the calling process; creating objects and links, getting, setting, and resetting attributes (including the working schema operations of 10.4), and converting objects by means of types which are not visible are invalid even for the PCTE_CONFIGURATION program group.

(13) Evaluation of a type reference *reference* may give rise to the following errors:

(14) If PCTE_CONFIGURATION is not effective for the calling process:
ATTRIBUTE_TYPE_IS_NOT_VISIBLE (*reference*)
ENUMERAL_TYPE_IS_NOT_VISIBLE (*reference*)
LINK_TYPE_IS_NOT_VISIBLE (*reference*)
OBJECT_TYPE_IS_NOT_VISIBLE (*reference*)

(15) If PCTE_CONFIGURATION is effective for the calling process:
OPERATION_IS_NOT_ALLOWED_ON_TYPE (*reference*)

(16) If *reference* contains a type identifier *identifier*, the following implementation-defined error may be raised:
TYPE_IDENTIFIER_IS_INVALID (*identifier*)

(17) If *reference* is a full type name:
SDS_IS_NOT_IN_WORKING_SCHEMA (SDS name of *reference*)
SDS_IS_UNKNOWN (SDS name of *reference*)
TYPE_IS_UNKNOWN_IN_SDS (SDS name of *reference*)

(18) TYPE_IS_NOT_DESCENDANT (*reference*, expected type)

(19) TYPE_IS_OF_WRONG_KIND (*reference*)

(20) Any use of an internal type reference *reference* as a parameter of an operation may additionally raise the following error:
TYPE_REFERENCE_IS_INVALID (*reference*)

NOTES

(21) 1 The ability of the program group PCTE_CONFIGURATION to identify types which are not part of the working schema of the calling process, is intended to be used to remove garbage from the object base, e.g. "type" objects associated with types which are no longer in existence.

(22) 2 Internal type references are independent of the working schema, and remain valid over changes to the working schema.

23.1.2.6 Type names in SDS

(1) A type nominator in SDS in clauses 9 to 22 corresponds to a type name in SDS in a language binding. A type name in SDS is a text value with the following syntax.

(2) type name in sds = local name | type identifier;

(3) A type name in SDS identifies a type in SDS in a given SDS. A type nominator in SDS returned by an operation is returned as a local name if the type is named in the relevant SDS, otherwise as a type identifier.

(4) Evaluation of a type name in SDS *name* may give rise to the following errors:

(5) TYPE_IDENTIFIER_USAGE_IS_INVALID (*name*)

(6) TYPE_IS_UNKNOWN_IN_SDS (given SDS, *name*)

23.1.2.7 Keys

(1) A value of type Actual_key in clauses 9 to 22 corresponds to a key in a language binding. A key is a text value with the following syntax.

(2) key = key attribute value, {'.', key attribute value};

(3) key attribute value = key natural value | key string value | key nil value;

(4) key string value = key first character, {key character};

(5) key first character = key character - ('$' | '#' | '~' | '_' | '+');

(6) key character = character - ('.' | '-' | '/');

(7) key natural value = '0' | nonzero digit, {digit} | highest used value | next unused value;

(8) nonzero digit = '1'| '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

(9) digit = '0' | nonzero digit;

(10) highest used value = '+';

(11) next unused value = '++';

(12) key nil value = '-';

(13) A key identifies a link in the context of a given origin object and an actual link type. It is evaluated to give an *actual key* according to the rules given below; the actual key is a sequence of key values (strings or naturals), and identifies the link with given type and origin object, and with that sequence of values of its key attributes. The length of a key is limited to MAX_KEY_SIZE and the values of key attribute values are limited to MAX_KEY_VALUE and MAX_KEY_SIZE (see 24.1).

(14) The key attribute values in the key are evaluated in order giving key values of the actual key as follows.

(15) - A key string value gives that string.

(16) - A key natural value of the first or second form gives the natural number which it represents in the usual decimal representation.

(17) - The highest used value '+' gives the greatest key attribute value, if any, in that position in the sequence of key values among all links which have the same origin and the same key prefix (the preceding sequence of key values evaluated according to these rules). If there are no such links, the value of '+' is zero.

(18) - The next unused value '++' gives the value of '+' plus one when the actual key is to be used as the key of a link created by the operation, and the value of '+' in other cases.

(19) - The key nil value '-' is undefined if the origin object of the link has no preferred link key, or if its preferred link type is not the given actual link type. Otherwise it gives the value of the key attribute in the same position in the preferred link key of the origin object. If the preferred link key attribute value is '+' or '++', it is evaluated as described above.

(20) - If fewer key values are present than the number of key attribute types of the given link type, then the number is effectively made up by adding further '-' key attribute values, except that if the origin object of the link has no preferred link key, or if its preferred link type is not the given actual link type, a missing key attribute value corresponding to a string key attribute gives the empty string.

(21) An actual key returned as or as part of the result of an operation has the form of a key with no '+', '++', or '-' key attribute values.

NOTES

(22) 1 Although a key must contain at least one key attribute value, the effect of an empty key can be obtained by using a key '-' consisting of a single key nil value.

(23) 2 All names are valid key string values.

23.1.3 Mapping of other values

23.1.3.1 Security labels

(1) A value of type Security_label in clauses 9 to 22 corresponds to a security label in a language binding. A security label is a text value with the following syntax.

(2) security_label = class name | conjunction | disjunction | '*';

(3) conjunction = unit, {space, 'AND', space, unit};

(4) disjunction = unit, {space, 'OR', space, unit};

(5) unit = class name | '(', security label,')';

(6) class name = name;

(7) space = (* space character *);

(8) A class name corresponds to the mandatory class designator of the security or integrity class which is the destination of a "known_mandatory_class" link from the mandatory directory with that class name as "name" key attribute value. The units in a conjunction or disjunction correspond to the security labels of the UNITS field in some order; all orders are equivalent.

(9) Class names are interpreted as confidentiality class names in confidentiality labels and as integrity class names in integrity labels. The security label value '*' is valid only as the high label of a range and represents MAXIMUM_LABEL (see 20.1.5). A null label is represented by an empty text value.

(10) Evaluation of a security label can give rise to the following error:

(11) CLASS_NAME_IS_INVALID (*name*)

23.1.3.2 Names

(1) A value of type Name is represented by a string having the syntax of a name according to the following rules.

(2) name = name first character, { name character };

(3) name first character = name character - ('/', '$', '#', '-', '_', '.', '~', '*', '(');

(4) name character = character - ('/', '$', '#', '-', '~', ')');

(5) The following synonyms for national characters (i.e outside the ISO 646 invariant subset) are permitted in names: /S for $, /T for ~, and /H for #.

(6) Names are used as SDS names and local names of types within SDSs, and as class names. The syntax is the same as for identifiers in DDL (see B.7).

23.1.3.3 Other compound types

(1) There are no constraints on the mapping of values of other VDM-SL compound types: sequences, sets, optional types, product types, record types, union types, and map types. Bindings may specify rules for special values, e.g. for a set containing all possible elements.

23.2 Object reference operations

23.2.1 OBJECT_REFERENCE_COPY

(1)
```
OBJECT_REFERENCE_COPY (
    reference          : Object_reference,
    point              : Evaluation_point
)
    new_reference      : Object_reference
```

(2) OBJECT_REFERENCE_COPY returns a new object reference *new_reference* designating the same object as *reference*. The evaluation status and evaluability of *new_reference* are specified by *point* (see 23.1.2.1).

(3) If *reference* is external and *point* is NOW, then *reference* is pre-evaluated.

Errors

(4) EVALUATION_STATUS_IS_INCONSISTENT_WITH_EVALUATION_POINT (*reference*, *point*)

23.2.2 OBJECT_REFERENCE_GET_EVALUATION_POINT

(1)
```
OBJECT_REFERENCE_GET_EVALUATION_POINT (
    reference  : Object_reference
)
    point      : Evaluation_point
```

(2) OBJECT_REFERENCE_GET_EVALUATION_POINT returns an evaluation point indicating the evaluation status and evaluability of the object reference *reference*, as defined in 23.1.2.1.

Errors

(3) None.

23.2.3 OBJECT_REFERENCE_GET_PATH

(1)
```
OBJECT_REFERENCE_GET_PATH (
    reference  : Object_reference
)
    pathname  : Pathname
```

(2) OBJECT_REFERENCE_GET_PATH returns the pathname of the external object reference *reference*.

Errors

(3) OBJECT_REFERENCE_IS_INTERNAL (*reference*)

23.2.4 OBJECT_REFERENCE_GET_STATUS

(1)
```
OBJECT_REFERENCE_GET_STATUS (
    reference  : Object_reference
)
    status     : Evaluation_status
```

(2) OBJECT_REFERENCE_GET_STATUS returns the evaluation status of the object reference *reference*.

Errors

(3) OBJECT_REFERENCE_IS_UNSET (*reference*)

23.2.5 OBJECT_REFERENCE_SET_ABSOLUTE

(1)
```
OBJECT_REFERENCE_SET_ABSOLUTE (
    pathname        : Pathname,
    point           : Evaluation_point
)
    new_reference   : Object_reference
```

(2) OBJECT_REFERENCE_SET_ABSOLUTE creates a new object reference *new_reference* from a pathname *pathname* and an evaluation point *point*. The evaluation status and evaluability of *new_reference* are specified by *point* (see 23.1.2.1).

Errors

(3) PATHNAME_SYNTAX_IS_WRONG (*pathname*)

23.2.6 OBJECT_REFERENCE_SET_RELATIVE

(1)
```
OBJECT_REFERENCE_SET_RELATIVE (
    reference       : Object_reference,
    pathname        : Relative_pathname,
    point           : Evaluation_point
)
    new_reference   : Object_reference
```

(2) OBJECT_REFERENCE_SET_RELATIVE creates a new object reference from an existing object reference *reference*, a relative pathname *pathname*, and an evaluation point *point*. The evaluation status and evaluability of *new_reference* are specified by *point* (see 23.1.2.1).

(3) If *reference* is external and *point* is NOW, then *reference* is pre-evaluated.

Errors

(4) EVALUATION_STATUS_IS_INCONSISTENT_WITH_EVALUATION_POINT (*reference*, *point*)

(5) If *point* is NOW:
REFERENCE_CANNOT_BE_ALLOCATED

(6) OBJECT_REFERENCE_IS_INVALID (*reference*)

(7) PATHNAME_SYNTAX_IS_WRONG (*pathname*)

23.2.7 OBJECT_REFERENCE_UNSET

(1)
```
OBJECT_REFERENCE_UNSET (
    reference   : Object_reference
)
```

(2) OBJECT_REFERENCE_UNSET deletes the object reference *reference*, releasing any associated resources.

Errors

(3) OBJECT_REFERENCE_IS_UNSET (*reference*)

23.2.8 OBJECT_REFERENCES_ARE_EQUAL

(1)
```
OBJECT_REFERENCES_ARE_EQUAL (
    first_reference     : Object_reference,
    second_reference    : Object_reference
)
    equal               : Reference_equality
```

(2) OBJECT_REFERENCES_ARE_EQUAL returns EQUAL_REFS if both object references *first_reference* and *second_reference* are internal and designate the same object; UNEQUAL_REFS if both object references are internal and designate different objects; and EXTERNAL_REFS otherwise (i.e. if one or both are external). REFERENCES_ARE_EQUAL does not evaluate either object reference.

Errors

(3) OBJECT_REFERENCE_IS_UNSET (*first_reference*)

(4) OBJECT_REFERENCE_IS_UNSET (*second_reference*)

23.3 Link reference operations

23.3.1 LINK_REFERENCE_COPY

(1)
```
LINK_REFERENCE_COPY (
    reference       : Link_reference
    point           : Evaluation_point
)
    new_reference   : Link_reference
```

(2) LINK_REFERENCE_COPY returns a new link reference *new_reference* which would identify the same link relative to a given object as *reference*. The evaluation status and evaluability of *new_reference* are specified by *point* (see 23.1.2.1).

(3) If *reference* is external and *point* is NOW, then *reference* is pre-evaluated.

Errors

(4) EVALUATION_STATUS_IS_INCONSISTENT_WITH_EVALUATION_POINT (*reference*, *point*)

(5) LINK_REFERENCE_IS_UNSET (*reference*)

23.3.2 LINK_REFERENCE_GET_EVALUATION_POINT

(1)
```
LINK_REFERENCE_GET_EVALUATION_POINT (
    reference   : Link_reference
)
    point       : Evaluation_point
```

(2) LINK_REFERENCE_GET_EVALUATION_POINT returns an evaluation point indicating the evaluation status and evaluability of the link reference *reference*, as defined in 23.1.2.1.

Errors

(3) LINK_REFERENCE_IS_UNSET (*reference*)

23.3.3 LINK_REFERENCE_GET_KEY

(1)
```
LINK_REFERENCE_GET_KEY (
    reference   : Link_reference
    )
    key         : Key
```

(2) LINK_REFERENCE_GET_KEY returns the key of the link reference *reference*.

(3) The key *key* is the key before any evaluation.

Errors

(4) LINK_REFERENCE_IS_UNSET (*reference*)

23.3.4 LINK_REFERENCE_GET_KEY_VALUE

(1)
```
LINK_REFERENCE_GET_KEY_VALUE (
    reference   : Link_reference,
    index       : Natural
    )
    key_value   : Natural | Text
```

(2) LINK_REFERENCE_GET_KEY_VALUE returns the key value *key_value* indexed by *index* of the link reference *reference*, if this key value exists.

(3) The first key value of the key of *reference* has index 1, the second 2, and so on.

Errors

(4) KEY_VALUE_DOES_NOT_EXIST (*reference*, *index*)

(5) LINK_REFERENCE_IS_UNSET (*reference*)

23.3.5 LINK_REFERENCE_GET_NAME

(1)
```
LINK_REFERENCE_GET_NAME (
    reference   : Link_reference
    )
    link_name   : Link_name
```

(2) LINK_REFERENCE_GET_NAME returns the link name of the link reference *reference*.

(3) If *reference* is external, then *reference* is pre-evaluated.

(4) If the link type of *reference* is visible then the link type name of *link_name* is the local name of the first type in working schema in the sequence of SDSs in the working schema, or if there is no local name, the link type name of *link_name* is the full type name of the first type in working schema in the sequence of SDSs in the working schema.

Errors

(5) LINK_NAME_IS_TOO_LONG_IN_CURRENT_WORKING_SCHEMA (link name of *reference*)

(6) LINK_REFERENCE_IS_UNSET (*reference*)

23.3.6 LINK_REFERENCE_GET_STATUS

(1)
```
LINK_REFERENCE_GET_STATUS (
    reference   : Link_reference
    )
    status      : Evaluation_status
```

(2) LINK_REFERENCE_GET_STATUS returns the evaluation status of the link reference *reference*.

Errors

(3) LINK_REFERENCE_IS_UNSET (*reference*)

23.3.7 LINK_REFERENCE_GET_TYPE

(1)
```
LINK_REFERENCE_GET_TYPE (
    reference           : Link_reference
)
    type_reference      : Link_type_reference
```

(2) LINK_REFERENCE_GET_TYPE returns the link type reference of the link reference *reference*.

Errors

(3) LINK_REFERENCE_IS_UNSET (*reference*)

23.3.8 LINK_REFERENCE_SET

(1)
```
LINK_REFERENCE_SET (
    link_name       : Link_name | Type_reference | (Key * Type_reference)
    point           : Evaluation_point
)
    new_reference   : Link_reference
```

(2) LINK_REFERENCE_SET creates a new link reference *new_reference* from a link name *link_name*. The evaluation status and evaluability of *new_reference* are specified by *point* (see 23.1.2.1).

(3) If *link_name* is provided as a link name (i.e. as a text value), then it must conform to the syntax rules defined in 23.1.2.4.

(4) If *link_name* is provided as a type reference *reference*, the link reference *new_reference* designates a cardinality one link with *reference* as a link type reference.

(5) If *link_name* is provided as a key *key* and a type reference *reference*, the link reference *new_reference* designates a cardinality many link with *key* as a key and *reference* as a link type reference.

Errors

(6) KEY_SYNTAX_IS_WRONG (*key*)

(7) If *link_name* is a type reference or a key and a type reference:
EVALUATION_STATUS_IS_INCONSISTENT_WITH_EVALUATION_POINT (*reference*, *point*)

(8) If *link_name* is a link name:
LINK_NAME_SYNTAX_IS_WRONG (*link_name*)

23.3.9 LINK_REFERENCE_UNSET

(1)
```
LINK_REFERENCE_UNSET (
    reference   : Link_reference
)
```

(2) LINK_REFERENCE_UNSET deletes the link reference *reference*, releasing any associated resources.

Errors

(3) LINK_REFERENCE_IS_UNSET (*reference*)

23.3.10 LINK_REFERENCES_ARE_EQUAL

(1)
```
LINK_REFERENCES_ARE_EQUAL (
    first_reference     : Link_reference,
    second_reference    : Link_reference
)
    equal               : Reference_equality
```

(2) LINK_REFERENCES_ARE_EQUAL returns EQUAL_REFS if both link references *first_reference* and *second_reference* are internal, they have textually equal keys, and their link types are equal as defined by TYPE_REFERENCES_ARE_EQUAL (see 23.4.8); UNEQUAL_REFS if both references are internal, but do not satisfy the equality rules for EQUAL_REFS; and EXTERNAL_REFS otherwise (i.e. if one or both are external). LINK_REFERENCES_ARE_EQUAL does not evaluate either link reference.

Errors

(3) LINK_REFERENCE_IS_UNSET (*first_reference*)

(4) LINK_REFERENCE_IS_UNSET (*second_reference*)

23.4 Type reference operations

23.4.1 TYPE_REFERENCE_COPY

(1)
```
TYPE_REFERENCE_COPY (
    reference       : Type_reference,
    point           : Evaluation_point
)
    new_reference : Type_reference
```

(2) TYPE_REFERENCE_COPY returns a new type reference *new_reference* identifying the same type as *reference*. The evaluation status and evaluability of *new_reference* are specified by *point* (see 23.1.2.1).

(3) If *reference* is external and *point* is NOW, then *reference* is pre-evaluated.

Errors

(4) EVALUATION_STATUS_IS_INCONSISTENT_WITH_EVALUATION_POINT (*reference, point*)

(5) TYPE_REFERENCE_IS_UNSET (*reference*)

23.4.2 TYPE_REFERENCE_GET_EVALUATION_POINT

(1)
```
TYPE_REFERENCE_GET_EVALUATION_POINT (
    reference   : Type_reference
)
    point       : Evaluation_point
```

(2) TYPE_REFERENCE_GET_EVALUATION_POINT returns an evaluation point indicating the evaluation status and evaluability of the type reference *reference*, as defined in 23.1.2.1.

Errors

(3) TYPE_REFERENCE_IS_UNSET (*reference*)

23.4.3 TYPE_REFERENCE_GET_IDENTIFIER

(1)
```
TYPE_REFERENCE_GET_IDENTIFIER (
    reference   : Type_reference
)
    identifier  : Type_name
```

(2) TYPE_REFERENCE_GET_IDENTIFIER returns the type identifier *identifier* of the type reference *reference*.

(3) If *reference* is external, then *reference* is pre-evaluated.

Errors

(4) TYPE_IS_NOT_VISIBLE (*reference*)

(5) TYPE_REFERENCE_IS_UNSET (*reference*)

23.4.4 TYPE_REFERENCE_GET_NAME

(1)
```
TYPE_REFERENCE_GET_NAME (
    sds         : [ Sds_designator ],
    reference   : Type_reference
)
    name        : Type_name
```

(2) TYPE_REFERENCE_GET_NAME returns the type name *name* of the type reference *reference*.

(3) If *sds* is not provided, *name* is the local name, full type name, or type identifier, according to the rules for returned names in 23.1.2.5.

(4) If *sds* is provided, *name* is the local name of the associated type in SDS in the SDS *sds*.

(5) If *reference* is external, then *reference* is pre-evaluated.

Errors

(6) If *sds* is supplied:
ACCESS_ERRORS (*sds*, ATOMIC, READ, READ_LINKS)
SDS_IS_UNKNOWN (*sds*)
TYPE_HAS_NO_LOCAL_NAME (*sds*, *reference*)
TYPE_IS_UNKNOWN_IN_SDS (*sds*, *reference*)

(7) TYPE_REFERENCE_IS_UNSET (*reference*)

23.4.5 TYPE_REFERENCE_GET_STATUS

(1)
```
TYPE_REFERENCE_GET_STATUS (
    reference   : Type_reference
)
    status      : Evaluation_status
```

(2) TYPE_REFERENCE_GET_STATUS returns the evaluation status of the type reference *reference*.

Errors

(3) TYPE_REFERENCE_IS_UNSET (*reference*)

23.4.6 TYPE_REFERENCE_SET

(1)
```
TYPE_REFERENCE_SET (
    name            : Type_name,
    point           : Evaluation_point
)
    new_reference   : Type_reference
```

(2) TYPE_REFERENCE_SET creates a new type reference *new_reference* from a type name *name* and an evaluation point *point*. The evaluation status and evaluability of *new_reference* are specified by *point* (see 23.1.2.1).

Errors

(3) TYPE_IDENTIFIER_IS_INVALID (*name*)

(4) If *point* is NOW:
TYPE_IS_NOT_VISIBLE (*name*)

(5) TYPE_NAME_IS_INVALID (*name*)

23.4.7 TYPE_REFERENCE_UNSET

(1)
```
TYPE_REFERENCE_UNSET (
    reference   : Type_reference
)
```

(2) TYPE_REFERENCE_UNSET deletes the type reference *reference* releasing any associated resources.

Errors

(3) TYPE_REFERENCE_IS_UNSET (*reference*)

23.4.8 TYPE_REFERENCES_ARE_EQUAL

(1)
```
TYPE_REFERENCES_ARE_EQUAL (
    first_reference     : Type_reference,
    second_reference    : Type_reference
)
    equal               : Reference_equality
```

(2) TYPE_REFERENCES_ARE_EQUAL returns EQUAL_REFS if both type references *first_reference* and *second_reference* are internal and designate the same type; UNEQUAL_REFS if both references are internal and designate different types; and EXTERNAL_REFS otherwise (i.e. if one or both are external). TYPE_REFERENCES_ARE_EQUAL does not evaluate either type reference.

Errors

(3) TYPE_REFERENCE_IS_UNSET (*first_reference*)

(4) TYPE_REFERENCE_IS_UNSET (*second_reference*)

24 Implementation limits

(1) Implementations may impose limits on the range, size, or number of certain quantities. Any attempt to exceed these limits gives rise to an error condition in the operation concerned. These error conditions are defined in the appropriate operation definitions.

(2) This section defines bounds on these implementation-defined limits. Each bound defines the most constrained value of the limit that an implementation may impose, and is therefore the limit that should be assumed for portability by a tool writer.

(3) The limits fall into two categories:

(4) - *installation-wide limits*, which must be the same for all workstations in a PCTE installation;

(5) - *workstation-dependent limits*, which may vary for different workstations in a PCTE installation.

24.1 Bounds on installation-wide limits

(1) MAX_ACCESS_CONTROL_LIST_LENGTH: The maximum number of entries which may appear in each of the atomic ACLs of an object or the default access control list of a process, as a natural; must be at least 64.

(2) MAX_ACCOUNT_DURATION, DELTA_ACCOUNT_DURATION: Upper and lower limits respectively for duration values in an accounting record, as floats. MAX_ACCOUNT_DURATION must be at least 86400 seconds; DELTA_ACCOUNT_DURATION must be at most 1 second.

(3) MAX_ACCOUNT_INFORMATION_LENGTH: The maximum number of octets in the INFORMATION field of an accounting record, as a natural; must be at least 128.

(4) MAX_AUDIT_INFORMATION_LENGTH: The maximum number of octets in the TEXT part of an audit record, as a natural; must be at least 128.

(5) MAX_DIGIT_FLOAT_ATTRIBUTE: The precision of a float attribute value, in decimal digits, as a natural; must be at least 6.

(6) MAX_FLOAT_ATTRIBUTE, MIN_FLOAT_ATTRIBUTE: Upper and lower limits respectively for float attribute values, as floats. MAX_FLOAT_ATTRIBUTE must be at least 10^{32}. MIN_FLOAT_ATTRIBUTE must be negative and its absolute value must be at least that of MAX_FLOAT_ATTRIBUTE.

(7) MAX_INTEGER_ATTRIBUTE, MIN_INTEGER_ATTRIBUTE: Upper and lower limits respectively for integer attribute values, as integers. MAX_INTEGER_ATTRIBUTE must be at least 2147483647; MIN_INTEGER_ATTRIBUTE must be negative, and its absolute value must be one greater than MAX_INTEGER_ATTRIBUTE.

(8) MAX_KEY_SIZE: The maximum number of octets in a string key attribute value, as a natural; must be at least 127.

(9) MAX_KEY_VALUE: The maximum natural value in a natural key attribute value; must be at least 32000.

(10) MAX_LINK_NAME_SIZE: The maximum number of octets in a link reference, as a natural; must be at least 191.

(11) MAX_MESSAGE_SIZE: The maximum number of octets in the data a message, as a natural; must be at least 1024.

(12) MAX_NAME_SIZE: The maximum number of octets in a name or enumeral type image, as a natural; must be at least 31.

(13) MAX_NATURAL_ATTRIBUTE: The upper limit for a non-key natural attribute value, as a natural. Must be at least 2147483647.

(14) MAX_PRIORITY_VALUE: The maximum priority value for a process, as a natural; must be at least 31.

(15) MAX_SECURITY_GROUPS: The maximum number of security groups which may be initialized in a PCTE installation, as a natural; must be at least 32000.

(16) MAX_STRING_ATTRIBUTE_SIZE: The maximum number of octets in a string non-key attribute value, as a natural; must be at least 32000.

(17) MAX_TIME_ATTRIBUTE: The latest time attribute value, as a time value; must be no earlier than 2044-12-31T24:00:00Z (24:00 on 31 December 2044 UTC).

(18) MIN_TIME_ATTRIBUTE: The earliest time attribute value, as a time value; must be no later than 1980-01-01T00:00:00Z (00:00 on 1 January 1980 UTC).

(19) SMALLEST_FLOAT_ATTRIBUTE: The lower limit on the absolute value of float attribute values, as a float; must be at greatest 10^{-32}.

24.2 Bounds on workstation-dependent limits

(1) MAX_ACTIVITIES: The maximum number of activities on the workstation, as a natural; must be at least 256.

(2) MAX_ACTIVITIES_PER_PROCESS: The maximum number of activities for a process executing on the workstation, as a natural; must be at least 8. MAX_ACTIVITIES_PER_PROCESS is also the maximum depth of nesting of activities on the workstation.

(3) MAX_FILE_SIZE: The maximum size in octets of a file on the workstation, as a natural; must be at least 100,000,000.

(4) MAX_MESSAGE_QUEUE_SPACE: The maximum total space of a message queue on the workstation, as a natural; must be at least 32000.

(5) MAX_MOUNTED_VOLUMES: The maximum number of volumes mounted on devices controlled by this workstation, as a natural; must be at least 16.

(6) MAX_OPEN_OBJECTS: The maximum number of concurrently open objects on the workstation, as a natural; must be at least 512.

(7) MAX_OPEN_OBJECTS_PER_PROCESS: The maximum number of concurrently open objects for a process executing on the workstation, as a natural; must be at least 16.

(8) MAX_PIPE_SIZE: The maximum size in octets in a pipe on the workstation, as a natural; must be at least 4096.

(9) MAX_PROCESSES: The maximum number of processes running, stopped, or suspended on the workstation, as a natural; must be at least 64.

(10) MAX_PROCESSES_PER_USER: The maximum number of processes existing simultaneously and created by a user on the workstation, as a natural; must be at least 16.

(11) MAX_SDS_IN_WORKING_SCHEMA: The maximum number of SDSs in a working schema on the workstation, as a natural; must be at least 32.

24.3 Limit operations

24.3.1 Datatypes for limit operations

(1) Limit_category = STANDARD | IMPLEMENTATION | REMAINING

(2) Limit_name = MAX_ACCESS_CONTROL_LIST_LENGTH | MAX_ACCOUNT_DURATION | DELTA_ACCOUNT_DURATION | MAX_ACCOUNT_INFORMATION_LENGTH | MAX_ACTIVITIES | MAX_ACTIVITIES_PER_PROCESS | MAX_AUDIT_INFORMATION_LENGTH | MAX_DIGIT_FLOAT_ATTRIBUTE | MAX_FILE_SIZE | MAX_FLOAT_ATTRIBUTE | MIN_FLOAT_ATTRIBUTE | MAX_INTEGER_ATTRIBUTE | MIN_INTEGER_ATTRIBUTE | MAX_KEY_SIZE | MAX_KEY_VALUE | MAX_LINK_REFERENCE_SIZE | MAX_MESSAGE_QUEUE_SPACE | MAX_MESSAGE_SIZE | MAX_MOUNTED_VOLUMES | MAX_NAME_SIZE | MAX_NATURAL_ATTRIBUTE | MAX_OPEN_OBJECTS | MAX_OPEN_OBJECTS_PER_PROCESS | MAX_PIPE_SIZE | MAX_PRIORITY_VALUE | MAX_PROCESSES | MAX_PROCESSES_PER_USER | MAX_SDS_IN_WORKING_SCHEMA | MAX_SECURITY_GROUPS | MAX_STRING_ATTRIBUTE_SIZE | MAX_TIME_ATTRIBUTE | MIN_TIME_ATTRIBUTE | SMALLEST_FLOAT_ATTRIBUTE

(3) Limit_value = Natural | Integer | Float | Time

(4) These datatypes are used in the operation LIMIT_GET_VALUE.

24.3.2 LIMIT_GET_VALUE

(1)
```
LIMIT_GET_VALUE (
    limit_category  : Limit_category,
    limit_name      : Limit_name
)
    limit_value     : [ Limit_value ]
```

(2) LIMIT_GET_VALUE returns for the limit named by *limit_name* the limit bound given in 24.1 or 24.2, the limit imposed by the implementation, or (where relevant) the current available quantity of the resource in the PCTE installation or local workstation, according as *limit_category* is STANDARD, IMPLEMENTATION, or REMAINING respectively. If the implementation does not impose a particular limit, then IMPLEMENTATION returns no value for that limit.

(3) REMAINING returns a value for the following values of *limit_name*, otherwise no value is returned: MAX_SECURITY_GROUPS, MAX_ACTIVITIES, MAX_MOUNTED_VOLUMES, MAX_OPEN_OBJECTS, MAX_PROCESSES.

Errors

(4) None.

Annex A
(normative)

VDM Specification Language for the Abstract Specification

A.1 Introduction

(1) The Abstract Specification uses a very limited subset of VDM-SL, the Specification Language of the Vienna Development Method as defined in ISO/IEC DIS 13817. This annex defines the subset, explains the semantics of the subset informally, and defines concrete syntax for use in the Abstract Specification.

(2) The definition of VDM-SL in ISO/IEC DIS 13817 is principally in terms of its abstract syntax, i.e. the bare structure of the language with all concrete syntactic details removed. It is permissible to use *any* concrete syntax which can be mapped to the abstract syntax, but ISO/IEC DIS 13817 does define two particular concrete syntaxes, called the mathematical and the ISO 646 syntaxes. As their names suggest, the former uses many mathematical and quasimathematical symbols, and is intended for typeset text; the latter uses only the ISO 646 character set and is intended for use with unsophisticated data preparation equipment and for information interchange. The two are virtually isomorphic down to the lexical level.

(3) The VDM-SL subset used in this ECMA Standard uses the ISO 646 syntax, which for the subset is perfectly readable and avoids the need for special symbols. Use is also made of a convention, defined below, for different styles for different kinds of identifiers.

A.2 The VDM-SL subset

(1) The subset consists of:

(2) - type definitions, to define the types of the various conceptual entities;

(3) - state definitions, to define the PCTE state;

(4) - a simple form of operation definitions, to define operation headings;

(5) - a simple form of function definition, to define auxiliary function headings;

(6) - operation and function calls;

(7) - identifiers;

(8) - literals, to express values.

(9) There are three small extensions. The first is of a purely syntactic nature; it has been found very useful for defining composite types as extensions of previously defined composite types (the '&&' notation). The second is used to relate the VDM-SL and the DDL definitions (the '**represented by**' notation). The third is to allow an operation to return more than result; it is simple in each case to map such an operation to an equivalent operation returning a single result of a composite type with the results of the original operation as its fields.

A.2.1 Type definitions

Syntax

(1) type definition = identifier, '=', type expression | identifier, '::', [identifier, '&&'], field list, ['**represented**', '**by**', identifier];

(2) type expression = bracketed type expression | type name | quote type expression | set type expression | map type expression | sequence type expression | union type expression | optional type expression | product type expression;

(3) bracketed type expression = '(', type expression, ')';

(4) type name = identifier;

(5) quote type expression = quote literal;

(6) set type expression = '**set**', '**of**', type expression;

(7) map type expression = '**map**', type expression, '**to**', type expression;

(8) sequence type expression = '**seq**', '**of**', type expression | '**seq1**', '**of**', type expression;

(9) union type expression = type expression, '|', type expression;

(10) optional type expression = '[', type expression, ']';

(11) product type expression = type expression, '*', type expression;

(12) field list = {field};

(13) field = [identifier, ':'], type expression;

Semantics

(14) A *type definition* declares a *type*, i.e. a set of values, and associates it with an identifier. The declared type is defined as follows, according to the different kinds of type definition.

(15) - I = T. The type denoted by the identifier I is defined by the type expression T.

(16) - I :: I1 : T1 I2 : T2 ... . The type denoted by I is a *composite type*: each value of the type is in effect an ordered set of values called *fields*, one from each of the field types T1, T2, ... in that order. The field identifiers I1, I2, ... are used in full VDM-SL to access the fields; in our subset they are descriptive only. The different notation is to emphasize that values from two different composite types are always distinct, even though they have the same field types and values: composite types use name equivalence, whereas other types use structural equivalence.

(17) The form I :: I' && I1 : T1 I2 : T2 ... is short for I :: I1' : T1' I2' : T2' ... I1 : T1 I2 : T2 ... where I1' : T1' I2' : T2' ... are the fields of the composite type I'.

(18) The '**represented by** I', if present, indicates that the type is represented by the predefined DDL object type definition with local name I. All the local names of object types in the predefined SDSs are different, so there is no ambiguity.

(19) The various forms of type expression define types as follows.

(20) - bracketed type expression (T). This denotes the same type as the type expression T; the brackets are used to override or emphasize the precedence of the type operators.

(21) - type name I. This denotes the type associated with the identifier I in a type definition. The predefined PCTE types (which take the place of VDM-SL basic types) are defined in 23.1.1.

(22) - quote type expression. This denotes a *quote type* containing a single *quote value* denoted by the same quote literal as the quote type expression. Quote values have no predefined

properties except equality and inequality. Types containing quote values are built up as union types, e.g.:

Colour = RED | GREEN | BLUE

Result = Natural | ERROR

(23) - set type expression: **set of** T. Values of this type are the finite subsets of values of type T.

(24) - sequence type expression: **seq of** T, **seq1 of** T. Values of these types are finite ordered sequences of values of type T; in the first case including, and in the second case excluding, the empty sequence.

(25) - map type expressions: **map** T1 **to** T2. A value of either of these types is a finite *map* from type T1 to type T2, i.e. an association with each of a finite subset of T1 (the *domain* of the map) of a value of T2. The element of T2 associated with an element *x* of the domain of a map *M* is denoted by *M*(*x*).

(26) - union type expression: T1 | T2 | ... . This denotes the union of the constituent types T1, T2, ... ; a value of a union type is a value of one of the constituent types (which must be disjoint).

(27) - optional type expression: [T]. This denotes the same type as T | **nil**, where **nil** is the only value of a special anonymous type (the *nil type*). **nil** is used conventionally to stand for absence of a value.

(28) - product type expression: T1 * T2 * ... . Values of a product type are ordered tuples of values from the constituent types T1, T2, ... The distinction from a composite type is that product types use structural equivalence, and the components are not named.

(29) NOTE - Optional types are sometimes used to indicate the preferred default value.

A.2.2 State definitions

Syntax

(1) state definition = '**state**', identifier, '**of**', field list, '**end**';

Semantics

(2) The *state definition*

```
state I of
    I1 : T1
    I2 : T2
      ...
end
```

defines the state to consist of variables I1, .. of types T1, ... respectively. In the VDM-SL subset the identifier I is descriptive only (in full VDM-SL it is used to identify the state from within a different module).

A.2.3 Operation headings

Syntax

(1) operation heading = identifier, '(', [typed parameter list], ')', [typed results];

(2) typed parameter list = identifier, ':', type expression, {',', identifier, ':', type expression};

(3) typed results = identifier, ':', type expression, {',', identifier, ':', type expression};

Semantics

(4) The *operation heading*

```
Op (
    I1: T1,
    I2: T2,
    ...,
)
    R1 : T1'
    R2 : T2'
    ...
```

declares Op as the name of an operation with formal parameters I1 of type T1, I2 of type T2, etc., and with results R1 of type T1', R2 of type T2', etc. If R1 : T1' etc. is omitted, the operation has no result. Note that the ( ) are needed even if the operation has no parameters.

A.2.4 Function headings

Syntax

(1) function heading = identifier, '(', [typed parameter list, ')', identifier, ':', type expression;

Semantics

(2) The *function heading*

```
Fn (
    I1 : T1,
    I2 : T2,
    ...
)
    R : Tr
```

declares Fn as the name of a function with formal parameters I1 of type T1, I2 of type T2, ..., and with result R of type Tr. Note that the ( ) are needed even if the function has no parameters.

A.2.5 Operation and function calls

Syntax

(1) operation call = identifier, '(', [expression, {',', expression}], ')';

(2) function application = identifier, '(', [expression, {',', expression}], ')';

Semantics

(3) The operation call or function application

```
Id (E1, E2, ... )
```

denotes a call to the operation or function with name Id, with actual parameters E1, E2, ...

A.2.6 Identifiers

Syntax

(1) identifier = plain letter , {plain letter | digit | '_'};

(2) plain letter = capital letter | small letter;

(3) capital letter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z';

(4) small letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z';

(5) digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

Semantics

(6) Identifiers have no intrinsic meaning; they are used as names of various entities.

A.2.7 Literals

Syntax

(1) symbolic literal = boolean literal | numeric literal | character literal | text literal | quote literal;

(2) boolean literal = '**true**' | '**false**';

(3) numeric literal = numeral, ['.', digit, {digit}], [exponent];

(4) exponent = 'E', ['+' | '-'], numeral;

(5) numeral = digit, {digit};

(6) character literal = ' ' ', character, ' ' ';

(7) text literal = ' " ', {' "" ' | character - ' " '}, ' " ';

(8) quote literal = capital letter, {'_' | capital letter};

Semantics

(9) Literals denote values of basic types.

(10) Boolean literals denote the corresponding truth values.

(11) Numeric literals denote rational numbers in the usual decimal notation, using '.' as the decimal point. An exponent E+*n* (or E*n*) or E-*n* denotes multiplication by 10^{n} or 10^{-n} respectively.

(12) A character literal denotes a graphic character. A text literal denotes a sequence of characters, i.e. a value of type seq of char; as usual, the double quote character " is denoted by two successive double quote characters : "".

(13) A quote literal denotes the only value of a quote type, and also the quote type itself.

A.3 Conventions for identifiers and keywords

(1) Identifiers are used in the VDM-SL subset to name entities of the kinds shown below. The conventions for these identifiers used in the Abstract Specification are shown. Note that two identifiers cannot be distinguished, strictly speaking, just by the use of italics or not, as the ISO 646 syntax only recognizes one set of letters (capitals and small letters).

(2) - Types: small letters with capital initial, e.g. Object, Type_in_sds.

(3) - Fields of composite types: capital letters, e.g. DIRECT_COMPONENTS.

(4) - States: there is only one, PCTE_Installation.

(5) - State variables: capital letters, e.g. OBJECT_BASE.

(6) - Operations: capital letters, e.g. OBJECT_GET_ATTRIBUTE

(7) - Operation parameters and results: small italic letters: *queue, next_message.*

(8) The same conventions are used in English text, except that names of types, fields, states, state variables, and values are converted to conventional English phrases (usually by replacing underscores by spaces and capital by small letters, but not always: PCTE installation, type in SDS).

(9) In the ISO 646 syntax keywords are essentially reserved; there is a way of using a keyword as an identifier, but this is not used in the Abstract Specification. The keywords in the subset are:

end false map of seq seq1 set
state to true value

Annex B
(normative)

The Data Definition Language (DDL)

(1) This annex defines the PCTE Data Definition Language (DDL). DDL is used to define SDSs and the type definitions within them, and so serves for the definition of the schema of a PCTE installation. 'This definition serves as a language standard for DDL.

B.1 SDSs and clauses

Syntax

(1) DDL definition = sds section, {sds section};

(2) sds section =
'**sds**', sds name, ':',
{clause, ';'},
'**end**', sds name, ';';

(3) clause =
type importation | object type declaration | object type extension |
attribute type declaration | link type declaration | link type extension |
enumeration type declaration;

(4) type importation =
'**import**', import type, global name, ['**as**', local name], [type mode declaration],
{ ',', global name, ['**as**', local name], [type mode declaration] };

(5) import type = '**object**', '**type**' | '**attribute**', '**type**' | '**link**', '**type**' | '**enumeral**', '**type**';

Meaning

(6) A *DDL definition* defines a number of SDSs containing types in SDS, and the corresponding types.

(7) All the *SDS sections* in the schema declaration with a particular SDS name define an SDS (schema definition set). This SDS has the common SDS name, and the set of types in SDS defined by the SDS sections as explained below.

(8) The SDS section in which a clause occurs, and the SDS to which it contributes, are called the *current* SDS section and SDS, respectively.

(9) The *type importation*

import import_type sds_name-local_name_1 **as** local_name_2 [type_mode_declaration]

defines a type in SDS in the current SDS. This type in SDS is a copy of the type in SDS denoted by local_name_1 in the SDS denoted by sds_name, with the same type identifier, except that the creation or importation date is set to the value of the system clock at the time of importation. It is denoted by local name local_name_2 in the current SDS; if local_name_2 is not given, the default is local_name_1. Except for an enumeral type, the type mode declaration defines the usage and export modes of the new type in SDS; they may not contain any access values not in the export mode of the imported type. The default for both modes is the export mode of the imported type. The maximum mode is set to the export mode of the imported type.

(10) A multiple type importation:

import import_type sds_name1-local_name1 [**as** new_local_name1] [type_mode_declaration1],
sds_name2-local_name2 [**as** new_local_name2] [type_mode_declaration2],
...,
sds_name*n*-local_name*n* [**as** new_local_name*n*] [type_mode_declaration*n*]

is equivalent to *n* single type importations:

import import_type sds_name1-local_name1 [**as** new_local_name1]
[type_mode_declaration1];
import import_type sds_name2-local_name2 [**as** new_local_name2]
[type_mode_declaration2];
...,
import import_type sds_name*n*-local_name*n* [**as** new_local_name*n*]
[type_mode_declaration*n*]

(11) Every type name used in an SDS must be declared in the same SDS, in a type importation, a type declaration, or type extension. Except for destination object types and reverse link types in link type declarations, all type names must be declared before use.

(12) If a type mode declaration is omitted, all the definition mode values are set for the imported type.

B.2 Object types

Syntax

(1) object type declaration =
local name, ':', [type mode declaration], ['**child**', '**type**', '**of**', object type list], ['**with**',
['**contents**', contents type indication, ';'],
['**attribute**',
attribute indication list, ';'],
['**link**',
link indication list, ';'],
['**component**',
component indication list, ';'],
'**end**', local name];

(2) object type extension =
'**extend**', '**object**', '**type**', local name,'**with**',
['**attribute**',
attribute indication list, ';'],
['**link**',
link indication list, ';'],
['**component**',
component indication list, ';'],
'**end**', local name;

(3) contents type indication = '**file**' | '**pipe**' | '**device**' | '**audit_file**' | '**accounting_log**';

(4) attribute indication list = attribute indication list item, { ';', attribute indication list item };

(5) attribute indication list item = attribute type name | attribute type declaration;

(6) link indication list = link indication list item, { ';', link indication list item };

(7) link indication list item = link type name | link type declaration;

(8) component indication list = component indication list item, { ';', component indication list item };

(9) component indication list item = link type name | link type declaration;

Constraints

(10) The '**child type of**' clause may be omitted only for the object type "object" (see 9.1.1).

(11) The local name after '**end**' in an object type declaration or object type extension, if present, must be the same as the first local name of that object type declaration or object type extension.

(12) In an object type declaration the local name must be distinct from the local names of all other object types, and of all attribute types and link types, defined in the same SDS as the object type declaration.

(13) In an object type extension the local name must be the name of an object type introduced earlier in the SDS by an object type declaration or a type importation.

(14) Each attribute type name in an attribute indication list must be the local name of an attribute type introduced earlier in the specification by an attribute type declaration or a type importation.

(15) Each link type name in a link indication list must be the local name of a non-composition link type introduced earlier in the specification by an object or link type declaration or a type importation.

(16) Each link type name in a component indication list must be the local name of a composition link type introduced earlier in the specification by an object or link type declaration or a type importation.

(17) The type mode declaration must define either **protected** or **create** for each of the usage mode and the export mode.

(18) All the attribute types and link types in the list must be different.

(19) If any parents of an object type have contents, then they must all have the same contents type (there may be other parents with no contents). The child type inherits the common contents type, or if none of its parents has contents, neither has the child.

Meaning

(20) An *object type declaration* defines an object type, and an object type in SDS in the current SDS with the local name within that SDS. The new object type has the following characteristics (see 8.3.1).

(21) - The contents type is as defined by the contents type indication. If a contents type indication is given, it defines the contents type of the object type as FILE, PIPE, DEVICE, AUDIT_FILE, or ACCOUNTING_LOG respectively (see 8.3.1). The contents type indication is used only for the predefined types "file", "pipe", "device", "audit_file", and "accounting_log"; user-defined types always inherit contents type (or lack of it) from their parents.

(22) - The parent types are the object types defined by the object type list after '**child type of**'; the object type is added to the child types of all its parent types. The object type has no child types initially.

(23) The new object type in SDS has the following characteristics (see 8.4.1).

(24) - The direct outgoing link types in SDS are all those defined by the link indication list and component indication list (see below).

(25) - The direct attribute types in SDS are all those defined by the attribute indication list (see below)

(26) - The direct component object types in SDS are all those defined as the destination object types of the composition links defined in the component indication list.

(27) - The usage and export modes are set by the type mode declaration, if present (see B.6); the default is usage mode and export mode both set to CREATE.

(28) - The maximum usage mode is set to CREATE.

(29) An *object type extension* extends the object type in SDS with the same local name in the current SDS section, by adding further outgoing link types, attribute types, and component object types, defined as for an object type declaration.

(30) An attribute indication list defines the following set of attribute types.

(31) - For each attribute indication list item which is an attribute type name, the attribute type with that local name in the current SDS.

(32) - For each attribute indication list item which is an attribute type declaration, all the attribute types defined by that attribute type declaration (see clause B.3).

(33) A link indication list or a component indication list defines the following set of link types.

(34) - For each link indication list item or component indication list item which is a link type name, the link type with that local name in the current SDS.

(35) - For each link indication list item or component indication list item which is a link type declaration, the link type defined by that link type declaration (see clause B.4).

B.3 Attribute types

Syntax

(1) attribute type declaration =
local name, {',' local name}, ':', [type mode declaration], ['**non_duplicated**'],
value type indication, [':=', initial value];

(2) value type indication= '**integer**' | '**natural**' | '**boolean**' | '**time**' | '**float**' | '**string**' |
'**enumeration**', enumeration type name | enumeration type indication;

(3) enumeration type indication = '**enumeration**', '(', basic enumeration,
{',', basic enumeration}, ')';

(4) basic enumeration = enumeration image | enumeration subrange;

(5) enumeration image = identifier | ' " ', {character}, ' " ' ;

(6) enumeration subrange = attribute type name, '**range**', enumeration image, '..',
enumeration image;

(7) initial value =
['+' | '-'], digit, {digit} (* integer *)
| digit, {digit} (* natural *)
| '**true**' | '**false**' (* boolean *)
| year, '-', month, '-', day, ['T', hour, ':', minute, ':', second], 'Z' (* time *)
| ['+' | '-'], digit, {digit}, ['.', digit, {digit}], ['E', ['+' | '-'], digit, {digit}] (* float *)
| ' " ', {character}, ' " ' (* string*)
| enumeration image; (* enumeration value type*)

(8) day = digit, digit;

(9) month = digit, digit;

(10) year = [digit, digit], digit, digit;

(11) hour = digit, digit;

(12) minute = digit, digit;

(13) second = digit, digit;

Constraints

(14) All the local names of an attribute type declaration must be distinct from the local names of all other attribute types, and of all object types and link types, defined in the current SDS.

(15) The initial value, if any, in an attribute type declaration must denote a value of the value type defined by the value type indication.

(16) In a basic enumeration which is an enumeration subrange, the two enumeration images must identify different enumeral types of the enumeration value type of the enumeration attribute type denoted by the attribute type name. In that enumeration value type the enumeral type identified by the second enumeration image must not precede that identified by the first.

(17) The type mode declaration must define either **protected** or one or both of **read** and **write** for each of the usage mode and the export mode.

Meaning

(18) An *attribute type declaration* defines an attribute type, and an attribute type in SDS in the current SDS with the local name within that SDS. The new attribute type has the following characteristics (see 8.3.2).

(19) - The value type is as given by the value type indication.

(20) - The initial value of the attribute type is defined by the initial value of the attribute type declaration, if present, as follows.

(21) . An initial value ['+' | '-'], digit, {digit} denotes an integer in the conventional decimal representation.

(22) . An initial value '**true**' or '**false**' denotes the logical value TRUE or FALSE respectively.

(23) . An initial value year, '-', month, '-', day, ['T' hour, ':', minute, ':', second] 'Z' denotes the time consisting of the given year, month, day, and the time of day given by hour, minute, second (if present) or otherwise 00:00:00, in Coordinated Universal Time (UCT). When omitted, the first two digits of the year are taken to be '19', thus '88-11-09Z' denotes the same time as '1988-11-09T00:00:00Z'.

(24) . An initial value ['+' | '-'], digit, {digit}, ['.', digit, {digit}], ['E', ['+' | '-'], digit, {digit}] denotes a rational number in the conventional decimal representation, where '.' represents the decimal point and E a decimal exponent. Leading and trailing zeros are permitted.

(25) . An initial value ' " ', {character}, ' " ' denotes the string value consisting of the sequence of characters, without the delimiting ' " ' characters, except that the character ' " ' within the string is denoted by ' "" '.

(26) . An initial value which is an enumeration image denotes the corresponding enumeral type.

(27) - The duplication is NON_DUPLICATED if the keyword '**non_duplicated**' appears, and otherwise DUPLICATED.

(28) The new attribute type in SDS has the following properties (see 8.4.2).

(29) - The usage and export modes are set by the type mode declaration, if present (see clause B.6); the default is usage mode and export mode both set to READ and WRITE.

(30) - The maximum usage mode is set to READ and WRITE.

(31) An enumeration type indication defines an enumeration value type consisting of the concatenation of the sequences of enumeral types defined by the constituent basic enumerations, as follows. An enumeration image which is delimited by ' " " ' is interpreted as for a string attribute initial value.

(32) - If the basic enumeration is an enumeration image, then a sequence containing the single enumeral type with that image in the current SDS.

(33) - If the basic enumeration is an enumeration subrange, then the sequence containing the enumeral types between and including those identified by the two enumeration images, in the ordering of the enumeration value type of the attribute type.

B.4 Link types

Syntax

(1) link type declaration =
local name, ':', [type mode declaration], ['**exclusive**'], ['**non_duplicated**'],
[stability name], category name, '**link**', [cardinality range], [key list],
['**to**', object type list], ['**reverse**', link type name], ['**with**',
'**attribute**',
attribute indication list, ';',
'**end**', local name];

(2) link type extension =
'**extend**', '**link**', '**type**', local name,['**to**', object type list], ['**with**',
'**attribute**',
attribute indication list, ';',
'**end**', local name];

(3) category name = ['**composition**'] | '**existence**' | '**reference**' | '**implicit**' | '**designation**';

(4) cardinality range = '[', [lower bound], '..', [upper bound], ']';

(5) lower bound = digit, { digit };

(6) upper bound = digit, { digit };

(7) stability name = '**atomic**', '**stable**' | '**composite**', '**stable**';

(8) key list = '(', attribute indication list, ')';

Constraints

(9) In a cardinality range:

(10) - The lower bound must be greater than or equal to zero.

(11) - The upper bound must be greater than zero.

(12) - The lower bound must be less than or equal to the upper bound.

(13) - Links of category name **implicit** or **existence** must have a lower bound equal to zero.

(14) If the upper bound of a link type is greater than one, and the link type has category name **implicit**, then the key list must consist of a single attribute type name. If the upper bound of a link type is greater than one, then the key list must not be omitted.

(15) The link type denoted by the link type name after '**reverse**', if present, must be a direct outgoing link type of each destination object type of the link type, and must have the link type as its reverse. See 8.3.3 for constraints on the properties of the reverse link type.

(16) The type mode declaration must define either **protected** or a combination of one or more of **create**, **navigate**, and **delete** for each of the usage mode and the export mode.

Meaning

(17) A *link type declaration* defines a link type, and a link type in SDS in the current SDS with the local name within that SDS. The new link type has the following properties (see 8.3.3).

(18) - Category: COMPOSITION, EXISTENCE, REFERENCE, IMPLICIT, or DESIGNATION, as given by the category name. The default is COMPOSITION.

(19) - Cardinality. As defined by the cardinality range. A missing lower bound is equivalent to a lower bound of 0. A missing upper bound is equivalent to an implementation-defined upper bound (which may depend on the link type) (see clause 24). A missing cardinality is equivalent to [0 ..] if there is a key list and [0..1] if not.

(20) - Exclusiveness: EXCLUSIVE if the keyword '**exclusive**' is present, otherwise SHARABLE.

(21) - Stability: ATOMIC_STABLE if the stability name '**atomic stable**' is present, COMPOSITE_STABLE if the stability name '**composite stable**' is present, and NON_STABLE if no stability name is present.

(22) - Duplication: NON_DUPLICATED if the keyword '**non_duplicated**' is present, otherwise DUPLICATED.

(23) - The reverse link type is the link type denoted by the link type name after '**reverse**'; if this is absent, there is no reverse link type if the link type is a designation link type, and otherwise the reverse link type is an implicit link type of cardinality many with no local name.

(24) The new link type in SDS has the following properties (see 8.4.3).

(25) - Destination object types. This is the set of object types denoted by the object type list.

(26) - Key attribute types. This is the sequence of attributes defined by that attribute indication list (see below).

(27) - Non-key attribute types. This is the set of attributes defined by the attribute indication list following the keyword '**with**', or the empty set if there is no such attribute indication list present.

(28) - The usage and export modes are set by the type mode declaration, if present (see clause B.6); the default is usage mode and export mode both set to CREATE, DELETE, and NAVIGATE.

(29) - The maximum usage mode is set to CREATE, DELETE, and NAVIGATE.

(30) When used in a key list, an attribute indication list denotes the list of attribute types as defined above, in the order of the appearance of their attribute type names or attribute type declarations in the attribute indication list.

(31) A *link type extension* extends the link type definition with the same local name in the current SDS section, by adding further outgoing destination object types and attribute types, defined as for a link type declaration.

B.5 Enumeration types

Syntax

(1)
```
enumeration type declaration =
    local name, ':', enumeration image, {',', enumeration image};
```

Meaning

(2) An enumeration type declaration defines one enumeral type, and one enumeral type in SDS in the current SDS, for each of the enumeration images. The local name may be used in an enumeration type name (see clause B.7) in an attribute type declaration to denote the enumeration value type consisting of the defined sequence of enumeral types.

(3) Each enumeral type in SDS has the corresponding enumeration image as its image (see 8.4.4).

B.6 Type mode declarations

Syntax

(1) type mode declaration = '(', '**usage**', type mode, ';', '**export**', type mode, ')'
| '(', ['**usage**', ',', '**export**'], type mode, ')';

(2) type mode = '**protected**' | allowed access, {',', allowed access};

(3) allowed access = '**read**' | '**write**' | '**navigate**' | '**create**' | '**delete**';

Meaning

(4) A type mode declaration occurs in a type declaration, and sets the usage and export modes of the type in SDS denoted by the type declaration to the values denoted by the type modes after **usage** and **export** respectively. The second form of type mode declaration sets both usage and export modes to the values denoted by the type mode.

(5) The type mode denotes the set of definition mode values corresponding to the allowed access values mentioned (i.e. READ_MODE, WRITE_MODE, NAVIGATE_MODE, CREATE_MODE, and DELETE_MODE, corresponding to **read**, **write**, **navigate**, **create**, and **delete** respectively), or the empty set if the type mode is **protected.**

B.7 Names

Syntax

(1) object type name = global name | local name;

(2) object type list = object type name, {',', object type name};

(3) attribute type name = global name | local name;

(4) attribute type list = attribute type name, {',', attribute type name};

(5) link type name = global name | local name;

(6) link type list = link type name, {',', link type name};

(7) enumeration type name = global name | local name;

(8) sds name = name;

(9) local name = name;

(10) global name = sds name, '-', local name;

(11) name = identifier | ' " ', character, { character }, ' " ';

(12) identifier = letter, {letter | digit | '_'};

(13) character = (* any value of the PCTE datatype Character (see 23.1.1.6) *);

(14) newline = (* implementation-defined *);

(15) capital letter = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z';

(16) small letter = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z';

(17) letter = capital letter | small letter;

(18) digit = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

Constraints

(19) An object, attribute, link, or enumeration type name without an SDS name must occur in an object, attribute, link, or enumeration type declaration, respectively, within the local specification.

(20) The local name of an object, attribute, link, or enumeration type name with an SDS name must occur in an object, attribute, link, or enumeration type declaration, respectively, within the specification with that SDS name.

(21) No spaces or control characters may occur between the component parts of a global name, nor within compound lexical items, i.e. keywords and compound symbols ('..', '--', ':=', """), enumeration images, initial values (except in strings, where they are significant), lower and upper bounds, and identifiers. One or more separators (spaces and/or newlines) must occur between adjacent keywords and identifiers.

(22) The sequence of characters of a name of the second form must respect the syntax of names in 23.1.3.2.

Meaning

(23) *Local names* denote object types, attribute types, and link types within an SDS, as established by the corresponding clauses.

(24) *Sds names* denote SDSs; an SDS name may be prefixed to a local name declared in that SDS to disambiguate it.

(25) NOTE – Differences of font and style are not significant; in particular there is no mandatory convention for distinguishing keywords from identifiers. Upper and lower case letters are however distinguished, and underscores in identifiers and keywords are significant, so that 'type1', 'Type1', and 'Type_1' are all distinct.

B.8 Comments

Syntax

(1) comment = '– –', { character }, newline;

Constraints

(2) A comment may contain any graphic characters. It may appear anywhere that a control character can, and nowhere else. It is terminated by the first newline after the opening '--'.

Meaning

(3) A comment has no effect on the meaning of the DDL definition.

B.9 Use of DDL identifiers as technical terms

(1) There are two styles for referring to DDL identifiers in running text:

(2) - Style 1. The description is phrased in representational terms, i.e. in terms of objects, attributes, and links. The appropriate style is as in:

(3) . A "type_in_sds" object is created.

(4) . The destination of the "adopted_user_group" link from the "system-process" object *process* is ...

(5) . ... the value of the "contents_confidentiality_label" attribute of the "volume_object" object must be ...

(6) The SDS name is not given; all DDL object type identifiers are distinct.

(7) - Style 2. The description is phrased in logical terms, i.e. in terms of logical entities. The appropriate style is as in:

(8) . A type in SDS is created.

(9) . The adopted user group of the process *process* is ...

(10) . ... the contents confidentiality label of the device must be ...

(11) These phrases are equivalent. Style 1 is always possible and is used when the mapping between logical entities and objects is not straightforward (e.g. types), at least to describe the mapping. Style 2 is preferred when there is no ambiguity; though the names used, especially for links, sometimes make this style grammatically impossible.

(12) The equivalent phrases 'a "type_in_sds" object' and 'a type in SDS' signify an object of type "type_in_sds" or of any descendant of "type_in_sds"; similarly for other object types.

Annex C
(normative)

Specification of Errors

C.1 Error conditions and error names

(1) For each error condition, the definition is obtained by textually substituting the actual parameters for the corresponding formal parameters throughout the text given in this annex. A list of two or more items as an actual parameter is enclosed in parentheses '( )' with the items separated by commas. A formal parameter in square brackets '[ ]' means that the corresponding actual parameter may be omitted.

(2) Each error name is in two parts: the name of the troublesome entity followed by an indication of what is wrong.

C.2 Scope of error conditions

(1) Some error condition definitions take as parameters an object designator *object* and an object scope *scope*. These are used to define a set of objects to any of which the error condition may apply. This set of objects is the *scope* of the error condition and is defined as follows.

(2) - *scope* = ATOMIC. The scope is the object *object*.

(3) - *scope* = COMPOSITE. The scope is the object *object* and all its subcomponents.

(4) - *scope* = COMPONENTS. The scope is an implementation-dependent subset of the object *object* and its components.

C.3 Groupings of errors

C.3.1 Access errors

(1) Certain error conditions apply to many operations which access an object or objects; they are grouped together for convenience under the name of *access errors*:

(2) ACCESS_ERRORS (*object*, *scope*, *access_mode*, [*permission*])

(3) If several objects are mentioned, the error conditions apply to each separately. The error conditions represented by ACCESS_ERRORS depend on the value of *access_mode*, as follows.

(4) - For *access_mode* = READ, MODIFY, CHANGE, or SYSTEM_ACCESS:

(5) OBJECT_IS_ARCHIVED (*object*)

(6) OBJECT_IS_INACCESSIBLE (*object*, *scope*)

(7) if object resides in a cluster then:
ACCESS_ERRORS (cluster of *object*, *scope*, *access_mode*, *permission*)

(8) - Additionally, for *access_mode* = READ, MODIFY, or CHANGE:

(9) CONFIDENTIALITY_WOULD_BE_VIOLATED (*object*, *scope*)

(10) DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*object*, *scope*, [*permission*])

(11) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (*object*, *scope*)

(12) - Additionally, for *access_mode* = MODIFY, CHANGE, or SYSTEM_ACCESS:

(13) VOLUME_IS_FULL (volume on which *object* resides)

(14) - Additionally, for *access_mode* = MODIFY or CHANGE:

(15) CONFIDENTIALITY_CONFINEMENT_WOULD_BE_VIOLATED (*object*, *scope*)

(16) INTEGRITY_WOULD_BE_VIOLATED (*object*, *scope*)

(17) PRIVILEGE_IS_NOT_GRANTED (PCTE_REPLICATION)

(18) REPLICATED_COPY_UPDATE_IS_FORBIDDEN (*object*, *scope*)

(19) VOLUME_IS_READ_ONLY (volume on which *object* resides)

(20) - Additionally, for *access_mode* = MODIFY:

(21) OBJECT_IS_STABLE (*object*)

C.3.2 Value limit errors

(1) Error conditions relating to the limits on attribute values are grouped together as *value limit errors*:

VALUE_LIMIT_ERRORS (*value*)

(2) This is equivalent to one or more cases of LIMIT_WOULD_BE_EXCEEDED, according to the value type of *value*:

(3) - value type is INTEGER:

LIMIT_WOULD_BE_EXCEEDED ((MAX_INTEGER_ATTRIBUTE, MIN_INTEGER_ATTRIBUTE))

(4) - value type is FLOAT:

LIMIT_WOULD_BE_EXCEEDED ((MAX_FLOAT_ATTRIBUTE, MIN_FLOAT_ATTRIBUTE, SMALLEST_FLOAT_ATTRIBUTE))

(5) - value type is STRING:

LIMIT_WOULD_BE_EXCEEDED (MAX_STRING_ATTRIBUTE_SIZE)

(6) - value type is TIME:

LIMIT_WOULD_BE_EXCEEDED ((MAX_TIME_ATTRIBUTE, MIN_TIME_ATTRIBUTE))

C.3.3 OWNER right errors

(1) OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*object*)

(2) This stands for the following, where *group* is the group for which the OWNER mode is being changed:

(3) ATOMIC_ACL_IS_INCOMPATIBLE_WITH_OWNER_CHANGE (*object*)

(4) OBJECT_HAS_GROUP_WHICH_IS_ALREADY_OWNER (*object*, *group*)

(5) COMPONENT_ADDITION_ERRORS (*dest*, *link*)

(6) This stands for:

(7) ACCESS_ERRORS (*dest*, COMPOSITE, CHANGE, OWNER)

(8) ACCESS_ERRORS (*dest*, ATOMIC, CHANGE, CONTROL_DISCRETIONARY)

(9) ACCESS_ERRORS (component of *dest*, ATOMIC, CHANGE, CONTROL_DISCRETIONARY)

(10) LINK_EXCLUSIVENESS_WOULD_BE_VIOLATED (*dest*, *link*)

(11) If origin of *link* has OWNER granted or denied for a group *group*:
OBJECT_HAS_GROUP_WHICH_IS_ALREADY_OWNER (*dest*, *group*)

(12) OBJECT_OWNER_CONSTRAINT_WOULD_BE_VIOLATED (*dest*)

(13) The following implementation-dependent error may be raised:
OBJECT_IS_INACCESSIBLE (outer object of *dest*, ATOMIC)

C.4 Other errors

(1) ACCESS_MODE_IS_INCOMPATIBLE (*scope*, *modes*) An attempt is being made to check for OWNER permission in *modes* when the scope *scope* is ATOMIC, or for CONTROL_DISCRETIONARY permission in *modes* when the scope *scope* is COMPOSITE.

(2) ACCESS_MODE_IS_NOT_ALLOWED (*scope*, *modes*) The access mode *modes* contains OWNER discretionary access mode and *scope* is ATOMIC or *modes* contains CONTROL_DISCRETIONARY and *scope* is COMPOSITE.

(3) ACCOUNTING_LOG_IS_NOT_ACTIVE (*accounting_log*) *accounting_log* is not currently an active accounting log.

(4) ACTIVITY_IS_OPERATING_ON_A_RESOURCE A resource is currently being operated on by the current activity.

(5) ACTIVITY_STATUS_IS_INVALID (*activity*, *status*) The activity status of *activity* does not have the value *status*.

(6) ACTIVITY_WAS_NOT_STARTED_BY_CALLING_PROCESS The current activity was not started by the calling process.

(7) ARCHIVE_EXISTS (*archive_identifier*) The natural value *archive_identifier* is already the archive identifier of a known archive.

(8) ARCHIVE_HAS_ARCHIVED_OBJECTS (*archive*) The archive *archive* already has archived objects.

(9) ARCHIVE_IS_INVALID_ON_DEVICE (*device*, *archive*) The contents of *device* either does not correspond in format to the implementation-defined format of an archive, or does not contain the archive identifier of *archive*.

(10) ARCHIVE_IS_UNKNOWN (*archive*) The archive *archive* is unknown, i.e. not the destination of a "known_archive" link from the archive directory.

(11) ATOMIC_ACL_IS_INCOMPATIBLE_WITH_OWNER_CHANGE (*object*) The groups for which OWNER is granted or denied for the origin object of a newly created composition link do not have CONTROL_DISCRETIONARY mode granted or denied respectively in the atomic ACL of the destination object *object* of that link.

(12) ATTRIBUTE_TYPE_IS_NOT_VISIBLE (*reference*) The attribute type identified by *reference* is not a visible type.

(13) ATTRIBUTE_TYPE_OF_LINK_TYPE_IS_NOT_APPLIED (*reference*) The attribute type of the attribute reference *reference* is not applied to the type of the link relative to which *reference* is being evaluated.

(14) ATTRIBUTE_TYPE_OF_OBJECT_TYPE_IS_NOT_APPLIED (*reference*) The attribute type of the attribute reference *reference* is not applied to the type of the object relative to which *reference* is being evaluated, nor to any of its visible ancestor types.

(15) AUDIT_FILE_IS_NOT_ACTIVE (*audit_file*) The object *audit_file* is not currently active as the audit file of any station.

(16) BREAKPOINT_IS_NOT_DEFINED (*breakpoint*) *breakpoint* is not a breakpoint set by PROCESS_ADD_BREAKPOINT

(17) CARDINALITY_IS_INVALID (*link_type*) The cardinality of *link_type* is not valid for the operation.

(18) CATEGORY_IS_BAD (*object*, *link*, *categories*) The category of the link *link* of the object *object* is not one of *categories*.

(19) CLASS_NAME_IS_INVALID (*name*) The string *name* is not a valid confidentiality or integrity class name (according to context).

(20) CLUSTER_EXISTS (*cluster_identifier*, *volume*) The specified cluster number *cluster_identifier* corresponds to an existing cluster in the volume *volume*.

(21) CLUSTER_HAS_OTHER_LINKS (*cluster*) There are links starting from the cluster *cluster* which are not the "cluster_in_volume" link to its associated volume.

(22) CLUSTER_IS_UNKNOWN (*cluster*) The "cluster" object *cluster* is not linked to a volume via link of type "known_cluster".

(23) CONFIDENTIALITY_CONFINEMENT_WOULD_BE_VIOLATED (*object*, *scope*) A confidentiality confinement violation would occur on one or more objects of the scope defined by *object* and *scope*. More precisely, for some object A of the scope of the error condition:

LABEL_DOMINATES (confidentiality label of A, confidentiality context of calling process) = **false**

(24) CONFIDENTIALITY_CRITERION_IS_NOT_SELECTED (*criterion*) The event to be removed in *criterion* is not in the list of confidentiality label dependent criteria.

(25) CONFIDENTIALITY_LABEL_IS_INVALID (*label*) The confidentiality label *label* does not conform to the defined syntax or contains a class that is not a known confidentiality class.

(26) CONFIDENTIALITY_WOULD_BE_VIOLATED (*object*, *scope*) A confidentiality violation would occur on one or more objects of the scope defined by *object* and *scope*. More precisely, for some object A of the scope of the error condition:

LABEL_DOMINATES (confidentiality context of calling process, confidentiality label of A) = **false**

(27) CONNECTION_IS_DENIED The requested connection to the network cannot be made.

(28) CONSUMER_GROUP_IS_IN_USE (*group*) The consumer group *group* is currently the associated consumer group of a process.

(29) CONSUMER_GROUP_IS_KNOWN (*group*) *group* is already a known consumer group of the accounting directory.

(30) CONSUMER_GROUP_IS_UNKNOWN (*object*) *object* is not a known consumer group of the accounting directory.

(31) CONTENTS_FORMAT_IS_INVALID (*file*) The contents of the file *file* was not saved by QUEUE_SAVE.

(32) CONTENTS_IS_NOT_EMPTY (*object*) *object* has a non-empty contents.

(33) CONTENTS_IS_NOT_FILE_CONTENTS (*contents*) The contents handle *contents* does not refer to a file contents.

(34) CONTENTS_IS_NOT_OPEN (*contents*) The contents handle *contents* does not refer to an open object.

(35) CONTENTS_OPERATION_IS_INVALID (*contents_handle*) *contents_handle* is the result of an opening where the opening mode or positioning does not allow the current operation to be performed.

(36) CONTROL_WOULD_NOT_BE_GRANTED (*new_object*) The CONTROL_DISCRETIONARY access right or CONTROL_MANDATORY access right would be no longer granted to any group or would be denied to the predefined user group ALL_USERS in the atomic ACL of *new_object* or one of its components.

(37) DATA_ARE_NOT_AVAILABLE (*contents*) It is not possible to read any octets from the opened object *contents.*

(38) DEFAULT_ACL_WOULD_BE_INCONSISTENT_WITH_DEFAULT_OBJECT_OWNER (*group*) *group* must have its CONTROL_DISCRETIONARY access mode value granted in the default atomic ACL of a given process if *group* is the default object owner of the same process.

(39) DEFAULT_ACL_WOULD_BE_INVALID (*process*, *group*, *modes*) No group would have atomic CONTROL_DISCRETIONARY discretionary access right or atomic CONTROL_MANDATORY discretionary access right to any objects (as defined in 19.1.3) created by *process* if *group* were set to *modes*.

(40) DEFINITION_MODE_VALUE_WOULD_BE_INVALID (*definition_mode*, *type*) The definition mode *definition_mode* contains a mode value which is invalid for the type kind of the type *type* as defined in 8.4.1, 8.4.2, and 8.4.3.

(41) DESTINATION_OBJECT_TYPE_IS_INVALID (*origin*, *link*, *destination_object*) In the current working schema the object type of *destination_object* is not among the destination object types of the link type of the link designator *link* of the object *origin*.

(42) DEVICE_CHARACTERISTICS_ARE_INVALID (*device_characteristics*) The string *device-characteristics* is not a valid device characteristics value.

(43) DEVICE_CONTROL_OPERATION_IS_INVALID (*device*, *operation*) The operation *operation* is not a valid operation or the device *device*.

(44) DEVICE_EXISTS (*device_identifier*) *device_identifier* is already the device identifier of a known device.

(45) DEVICE_IS_BUSY (*device_object*, *volume_identifier*) Another volume than the volume designated by the identifier *volume_identifier* is mounted on the device represented by *device_object*.

(46) DEVICE_IS_IN_USE (*device*) The device *device* is in use, i.e. a volume is mounted on it, or its contents is open, or it is in use by an archiving operation.

(47) DEVICE_IS_UNKNOWN (*device_object*) The physical device described by the object *device_object* does not exist (i.e. there is no "controlled_device" link from a workstation to *device_object*).

(48) DEVICE_LIMIT_WOULD_BE_EXCEEDED (*data, contents*) The writing of *data* to *contents* would exceed a device-dependent maximum size.

(49) DEVICE_SPACE_IS_FULL (*device*) The device *device* has insufficient available space.

(50) DISCRETIONARY_ACCESS_IS_NOT_GRANTED (*object, scope,* [*permission*]) The calling process does not have the atomic or composite right (according to *scope*) *permission* (or one of the rights if a list is given) to the object *object*. If *permission* is absent, the calling process does not have any atomic or composite right (according to *scope*) to *objects*. (If a list of access rights is given, they are all required.)

(51) DISCRETIONARY_ACCESS_IS_NOT_GRANTED_TO_PROCESS (*process, object, scope,* [*permission*]) The process *process* does not have the atomic or composite right (according to *scope*) *permission* (or one of the rights if a list is given) to the object *object*. If *permission* is absent, *process* does not have any atomic or composite rights (according to *scope*) to *object*. (If a list of access rights is given, they are all required.)

(52) ENUMERATION_ATTRIBUTE_WOULD_HAVE_NO_ENUMERAL_TYPES (*values*) An enumeration attribute type would be created with an empty sequence *values* of enumeral types.

(53) ENUMERAL_TYPE_IS_INVALID (*value*) *value* is not an enumeral type which is a member of the sequence of enumeral types defined for the enumeration attribute type.

(54) ENUMERAL_TYPE_IS_NOT_IN_ATTRIBUTE_VALUE_TYPE (*type1*, *type2*) The enumeral type *type1* is not a member of the value type of the enumeration attribute type *type2*.

(55) ENUMERAL_TYPE_IS_NOT_VISIBLE (*reference*) The enumeral type identified by *reference* is not a visible type.

(56) ENUMERAL_TYPES_ARE_MULTIPLE (*enumeral_types*) An enumeral type occurs more than once in the sequence *enumeral_types*.

(57) ENUMERATION_VALUE_IS_OUT_OF_RANGE (*initial_value*, *values*) The value given by *initial_value* is outside the range of positions defined by the sequence of enumerals *values*.

(58) EVALUATION_STATUS_IS_INCONSISTENT_WITH_EVALUATION_POINT (*reference*, *point*) The evaluation status of the object reference, link reference, or type reference *reference* is internal but the evaluation point *point* is not NOW.

(59) EVENT_TYPE_IS_NOT_SELECTED (*event_type*) There is no general selection criterion with the selectable event type *event_type* in the list of criteria.

(60) EXECUTION_CLASS_HAS_NO_USABLE_EXECUTION_SITES (*execution_class*) The execution class *execution_class* has no usable execution sites.

(61) EXECUTION_SITE_IS_INACCESSIBLE (*site*) The execution site *site* is not accessible.

(62) EXECUTION_SITE_IS_NOT_IN_EXECUTION_CLASS (*site*, *static_context*) The execution site *site* is not in the execution class of the static context *static_context*.

(63) EXECUTION_SITE_IS_UNKNOWN (*site*) The execution site *site* is unknown to the PCTE installation (i.e. it is not linked to the execution site directory with a "known_execution_site" link).

(64) EXTERNAL_LINK_IS_BAD (*version*) One or more of the outgoing external existence, reference, or designation links of *version* which is to be copied is to an inaccessible object.

(65) EXTERNAL_LINK_IS_NOT_DUPLICABLE (*object*) One or more of the outgoing external existence, reference, or designation links of *object* is reversed by a link which is not an implicit link of cardinality many.

(66) FOREIGN_DEVICE_IS_INVALID (*foreign_device*) *foreign_device* cannot be interpreted as specifying an appropriate device.

(67) FOREIGN_EXECUTION_IMAGE_HAS_NO_SITE (*image*) There is no "on_foreign_system" link from the foreign execution image *image*.

(68) FOREIGN_EXECUTION_IMAGE_IS_BEING_EXECUTED (*foreign_system*, *foreign_name*) The file designated by *foreign_name* on *foreign_system* is a foreign execution image which is being executed.

(69) FOREIGN_OBJECT_IS_INACCESSIBLE (*foreign_system*, *foreign_name*) The object *foreign_object* residing on the foreign system *foreign_system* is not accessible.

(70) FOREIGN_SYSTEM_IS_INACCESSIBLE (*foreign_system*) The foreign system identified by *foreign_system* is valid but is not accessible.

(71) FOREIGN_SYSTEM_IS_INVALID (*foreign_system*, *process*, *class*) The foreign system *foreign_system*, that is or would be the execution site of *process*, does not support the functionality of the class *class*.

(72) FOREIGN_SYSTEM_IS_UNKNOWN (*foreign_system*) The foreign system *foreign_system* does not identify an execution site known to the PCTE installation.

(73) GROUP_IDENTIFIER_IS_IN_USE (*natural*) The natural value *natural* is already in use as a security group identifier.

(74) GROUP_IDENTIFIER_IS_INVALID (*natural*) The natural value *natural* has never been returned by GROUP_INITIALIZE.

(75) IMAGE_IS_ALREADY_ASSOCIATED (*image*, *sds*, *enumeral_type*) The string *image* is already the associated image of an enumeral type in SDS in the SDS *sds*, and there is an enumeration attribute type in SDS in *sds* with value type containing both that enumeral type and *enumeral_type*.

(76) IMAGE_IS_DUPLICATED (*enumeral_types*, *sds*) Two or more of the enumeral types in SDS associated with the elements of *enumeral_types* in the SDS *sds* have the same image.

(77) INTEGRITY_CONFINEMENT_WOULD_BE_VIOLATED (*object*, *scope*) An integrity confinement violation would occur on one or more objects of the scope defined by *object* and *scope*. More precisely, for some object A of the scope of the error condition:

LABEL_DOMINATES (integrity label of A, integrity context of calling process) = **false**

(78) INTEGRITY_CRITERION_IS_NOT_SELECTED (*criterion*) The event to be removed in *criterion* is not in the list of integrity label dependent criteria.

(79) INTEGRITY_LABEL_IS_INVALID (*label*) The integrity label *label* does not conform to the defined syntax or contains a class that is not a known integrity class.

(80) INTEGRITY_WOULD_BE_VIOLATED (*object*, *scope*) An integrity violation would occur on one or more objects of the scope defined by *object* and *scope*. More precisely, for some object A of the scope of the error condition:

(81) LABEL_DOMINATES (integrity context of calling process, integrity label of A) = **false**

(82) INTERPRETER_IS_INTERPRETABLE (*static_context*) The selected interpreter for the static context *static_context* is itself an interpretable static context and so cannot be used as an interpreter.

(83) INTERPRETER_IS_NOT_AVAILABLE (*static_context*) *static_context* is an interpretable static context but no interpreter for it can be executed.

(84) KEY_ATTRIBUTE_TYPE_APPLY_IS_FORBIDDEN (*attribute_type*) The attribute type *attribute_type* is a key attribute type and cannot be applied.

(85) KEY_IS_BAD (*object*, *reference*) The link reference *reference* has a key which supplies either too many or too few key attribute values or supplies key attribute values of the wrong value type for the link type specified by reference and the origin *object*.

(86) KEY_SYNTAX_IS_WRONG (*key*) The text value *key* does not have the syntax of a key.

(87) KEY_TYPE_IS_BAD (*type*) The value type of the attribute type *type* is not natural or string.

(88) KEY_TYPES_ARE_MULTIPLE (*types*) An attribute type occurs more than once in *types*.

(89) KEY_UPDATE_IS_FORBIDDEN (*object*, *link*, *attribute*) The attribute *attribute* of the link *link* of the object *object* is a key attribute and cannot be modified.

(90) KEY_VALUE_DOES_NOT_EXIST (*link_reference*, *index*) The index *index* designates a key value in the key of the link reference *link_reference* which does not exist.

(91) LABEL_IS_OUTSIDE_RANGE (*object*, *device*) A mandatory label of the object *object* is or would be outside the corresponding mandatory security range of the multi-level secure device *device*. More precisely, one of the following is **false**:

CONFIDENTIALITY_LABEL_WITHIN_RANGE (*object*, *device*)

INTEGRITY_LABEL_WITHIN_RANGE (*object*, *device*)

(92) LABEL_RANGE_IS_BAD (*high_label*, *low_label*) *high_label* does not dominate *low_label* in a new security range for a multi-level secure device. More precisely, LABEL_DOMINATES (*high_label*, *low_label*) is **false**.

(93) LAN_ERROR_EXISTS An error has occurred on the local area network.

(94) LIMIT_WOULD_BE_EXCEEDED (*limit*) The implementation limit *limit* would be exceeded.

(95) LINK_DESTINATION_DOES_NOT_EXIST (*link*) The destination of the link *link* does not exist and so cannot be accessed.

(96) LINK_DESTINATION_IS_NOT_VISIBLE (*link*) The object type of the destination of the link *link* is not a visible destination object type of the link type of *link*.

(97) LINK_DOES_NOT_EXIST (*object*, *link_name*) The link specified by *link_name* and *object* as origin does not exist and therefore cannot be accessed.

(98) LINK_EXCLUSIVENESS_WOULD_BE_VIOLATED (*dest*, *link*) Creation of the link *link* to the object *dest* would violate exclusivity: either *link* is an exclusive composition link and there is already a composition link to *dest*, or *link* is a composition link and there is already an exclusive composition link to *dest*.

(99) LINK_EXISTS (*object*, *link*) The link *link* of the object *object* already exists and therefore cannot be created.

(100) LINK_NAME_IS_TOO_LONG_IN_CURRENT_WORKING_SCHEMA (*link_name*) The link name *link_name* cannot be represented in the current working schema because the limit MAX_LINK_NAME_SIZE would be exceeded.

(101) LINK_NAME_SYNTAX_IS_WRONG (*link_name*) The syntax of the link name *link_name* is wrong.

(102) LINK_REFERENCE_IS_UNSET (*reference*) The link reference *reference* has never been set or has been explicitly unset by LINK_REFERENCE_UNSET.

(103) LINK_TYPE_CATEGORY_IS_BAD (*link_type*, *categories*) The category of the link type *link_type* is not one of *categories*.

(104) LINK_TYPE_IS_NOT_APPLIED_TO_OBJECT_TYPE (*object_reference*, *link_reference*) The link type of *link_reference* is not a visible type of the object type in working schema of the object *object_reference*.

(105) LINK_TYPE_IS_NOT_VISIBLE (*reference*) The link type identified by *reference* is not a visible type.

(106) LINK_TYPE_IS_UNKNOWN (*name*) *name* is not the name of a link type in the current working schema.

(107) LINK_TYPE_PROPERTIES_AND_KEY_TYPES_ARE_INCONSISTENT (*link_type_properties*, *key_attribute_types*) The properties specified by *link_type_properties* are inconsistent with the key attribute types specified by *key_attribute_types*.

(108) LINK_TYPE_PROPERTIES_ARE_INCONSISTENT (*link_type_properties*) The properties specified by *link_type_properties* are inconsistent among themselves (see 8.3.3).

(109) LOCK_COULD_NOT_BE_ESTABLISHED (*resource*, *scope*) A lock could not be immediately established or promoted on the resource *resource* with scope *scope*.

(110) LOCK_INTERNAL_MODE_CANNOT_BE_CHANGED (*object*, *lock_mode*) There is a conflict between the requested internal lock mode *lock_mode* of object *object* and other concurrent acquisitions of resources in the concerned domain of *object*.

(111) LOCK_IS_NOT_EXPLICIT (*object*) The object *object* is not explicitly locked by the current activity.

(112) LOCK_MODE_IS_NOT_ALLOWED (*lock_mode*) The lock mode *lock_mode* is WTR or DTR which is not allowed in the current activity.

(113) LOCK_MODE_IS_TOO_STRONG (*lock_mode*, *resource*) The required internal mode *lock_mode* is stronger than the external one established on *resource*.

(114) LOWER_BOUND_WOULD_BE_VIOLATED (*object*, *link*) The object *object* is not deleted by the operation, and the deletion of *link* from its origin *object* would leave the number of links of *object* with the link type of *link* less than the lower bound of the link type of *link*.

(115) MANDATORY_CLASS_IS_ALREADY_DOMINATED (*object*) There already exists a confidentiality or integrity class which directly dominates the confidentiality or integrity class designated by *object* . Only one such class is permitted.

(116) MANDATORY_CLASS_IS_KNOWN(*object*) *object* is already a known mandatory class.

(117) MANDATORY_CLASS_IS_UNKNOWN (*class*) *class* does not identify a mandatory class known to the PCTE installation, i.e. there is no link from the mandatory directory to *class*.

(118) MANDATORY_CLASS_NAME_IS_IN_USE (*class_name*) There already exists a mandatory class with the name *class_name*.

(119) MASTER_IS_INACCESSIBLE (*object*, *scope*) The master of the object *object*, or *object* itself if normal, is inaccessible.

(120) MAXIMUM_USAGE_MODE_WOULD_BE_EXCEEDED (*type*, *definition_mode*) The usage mode or export mode *definition_mode* exceeds the maximum usage mode associated with the type *type*.

(121) MEMORY_ADDRESS_IS_OUT_OF_PROCESS (*address*, *process*) The address *address* is not a valid address in the memory space of the process *process*.

(122) MEMORY_REGION_IS_NOT_IN_PROFILING_SPACE (*start*, *end*) The addresses *start* and *end* do not define a region in the profiling space of the calling process.

(123) MESSAGE_IS_NOT_A_NOTIFICATION_MESSAGE (*message*) The message type of the message *message* is not a notification message type.

(124) MESSAGE_POSITION_IS_NOT_VALID (*context*, *queue*) The natural *position* does not denote a position in the message queue *queue*.

(125) MESSAGE_QUEUE_HAS_BEEN_DELETED (*queue*) The message queue *queue* was deleted while the calling process was waiting to receive a message from it.

(126) MESSAGE_QUEUE_HAS_BEEN_WOKEN (*queue*) A message of message type WAKE was received while the calling process was waiting to receive a message from the message queue *queue*.

(127) MESSAGE_QUEUE_HAS_NO_HANDLER (*queue*) No valid handler routine has been specified for the message queue *queue*.

(128) MESSAGE_QUEUE_IS_BUSY (*queue*) There are messages on the message queue *queue*.

(129) MESSAGE_QUEUE_IS_NOT_RESERVED (*queue*) The caller has not reserved the message queue *queue*.

(130) MESSAGE_QUEUE_IS_RESERVED (*queue*) The message queue *queue* is reserved by another process.

(131) MESSAGE_QUEUE_TOTAL_SPACE_WOULD_BE_TOO_SMALL (*queue*, *total_space*) The space currently in use by message queue *queue* is greater than the requested space limit of *total_space* or *total_space* is less than four times MAX_MESSAGE_SIZE.

(132) MESSAGE_QUEUE_WOULD_BE_TOO_BIG (*queue*, *file*) The size of the message queue *queue* would exceed the queue's current value of the total space.

(133) MESSAGE_TYPES_NOT_FOUND_IN_QUEUE (*queue*, *types*, *context*) There is no message of a message type in *types* in the message queue *queue* after the position given by *context*.

(134) NON_BLOCKING_IO_IS_INVALID (*object*, *non_blocking_io*) An attempt is being made to open an object *object* of type "file" with *non_blocking_io* **false**, or to open an object *object* of type "pipe" or "device" with *non_blocking_io* **true** when it does not support non-blocking input-output.

(135) NOTIFIER_KEY_DOES_NOT_EXIST (*natural*) No notifier with the key *natural* exists.

(136) NOTIFIER_KEY_EXISTS (*natural*) A notifier with the key *natural* exists.

(137) NUMBER_OF_PARAMETERS_IS_WRONG (*operation*) The number of parameters of the operation *operation* does not match the operation signature.

(138) OBJECT_ARCHIVING_IS_INVALID (*objects*) One or more of the objects of *objects* cannot be archived, i.e. it is one of the following:

- a master or copy object;
- a volume;
- a locked object;
- a message queue that is reserved or contains one or more messages;
- a pipe or device;
- an active audit file or accounting log;
- a static context that is being executed or interpreted;
- an active process or activity;
- a mandatory class;
- an archive.

(139) OBJECT_CANNOT_BE_CLUSTERED (*object*) An attempt is being made to create an object *object* or a copy of an object *object* in a cluster, or to move an object *object* into a cluster, but the type of *object* is "file", "pipe", "message_queue", "device", "accounting_log", "audit_file", "volume", "cluster", "archive", "archive_directory", "process", "activity", "common_root", "sds", "workstation", "execution_class", "execution_site", "execution_site_directory", "replica_set_directory", "replica_set", "security_group", "program_group", "mandatory_directory", "mandatory_class", or "security_group_directory", or a descendant of one of those types.

(140) OBJECT_CANNOT_BE_STABILIZED (*object*) The object *object* cannot be stabilized, i.e. it is one of the following:

- an active process or activity;
- an active audit file or accounting log;
- a mounted volume;
- a message queue;
- a pipe.

(141) OBJECT_CRITERION_IS_NOT_SELECTED (*criterion*) The event to be removed in *criterion* is not in the list of object-dependent criteria.

(142) OBJECT_HAS_COPIES (*object*) The master object *object* has copies and therefore cannot be removed.

(143) OBJECT_HAS_EXTERNAL_LINKS_PREVENTING_DELETION (*object*) The object *object* or a component of *object* that should be deleted by the operation has incoming external reference links or outgoing external existence links.

(144) OBJECT_HAS_GROUP_WHICH_IS_ALREADY_OWNER (*object*, *group*) An attempt is being made to change the OWNER discretionary access mode value to DENIED for *group* when *group* is not in the effective discretionary groups for the calling process, and *group* has OWNER right granted in the composite ACL of *object*. See 19.1.2.

(145) OBJECT_HAS_INTERNAL_LINKS_PREVENTING_DELETION (*object*) The object *object* or a component of *object* cannot be deleted as there are internal reference links from another component which is the destination of an external composition or existence link.

(146) OBJECT_HAS_LINKS_PREVENTING_DELETION (*object*) Other reference links to the object *object* or composition or existence links from it exist.

(147) OBJECT_IS_A_PROCESS (*object*). The object *object* is a process.

(148) OBJECT_IS_A_REPLICA_SET (*object*) *object* is a replica set and so must be replicated in itself. A copy of *object* must exist on each of its copy volumes.

(149) OBJECT_IS_ALREADY_IN_RESOURCE_GROUP (*object*, *group*) The object *object* is already a member of the resource group *group*.

(150) OBJECT_IS_ARCHIVED (*object*) The object *object* resides in an archive.

(151) OBJECT_IS_FINE_GRAIN (*object*) *object* is fine-grain and an attempt is being made to perform one of the operations which are not permitted on fine-grain objects.

(152) OBJECT_IS_IN_USE_FOR_DELETE (*object*) The object *object* , or any of its non-shared components, is being operated on or is one of:

- the "process" object of an executing process;
- a static context being executed or interpreted;
- the "activity" object of a non-terminated activity;
- a message queue which is non-empty or reserved.

(153) OBJECT_IS_IN_USE_FOR_MOVE (*object*) The object *object* or any of its components is being operated on or is one of:

- the "process" object of an executing process;
- a static context being executed or interpreted;
- the "activity" object of a non-terminated activity;
- a message queue which is non-empty or reserved.
- *object* is locked (including an object with open contents);
- *object* is an active accounting log and accounting is switched on;
- *object* is an active audit file.

(154) OBJECT_IS_INACCESSIBLE (*object*, *scope*) One or more of the objects defined by *object* and *scope* is not accessible.

(155) OBJECT_IS_INACCESSIBLY_ARCHIVED (*object*, *scope*) One or more of the objects defined by *object* and *scope* is archived and the archive is inaccessible.

(156) OBJECT_IS_LOCKED (*object*, *scope*) One or more objects defined by *object* and *scope* is locked.

(157) OBJECT_IS_NOT_ACCOUNTABLE_RESOURCE (*object*) The object *object* is not an accountable resource.

(158) OBJECT_IS_NOT_ARCHIVED (*object*) The object *object* does not reside in an archive.

(159) OBJECT_IS_NOT_CONVERTIBLE (*object*) The object *object* cannot have its type converted because its replicated state is not NORMAL, and it is an object of a type such that it cannot be replicated (see 17.1.2).

(160) OBJECT_IS_NOT_IN_RESOURCE_GROUP (*object*, *group*) The object *object* is not in the resource group *group*.

(161) OBJECT_IS_NOT_LOCKED (*object*) There is no lock on the object resource *object*.

(162) OBJECT_IS_NOT_MASTER_REPLICATED_OBJECT (*object*) The object *object* does not have replication status MASTER.

(163) OBJECT_IS_NOT_MOVABLE (*object*, *scope*) The type of one or more of the objects of the scope defined by *object* and *scope* is one.of the predefined types "volume", "device", and "workstation", or a descendant of one of those types.

(164) OBJECT_IS_NOT_ON_ADMINISTRATION_VOLUME (*object*, *station*) The object *object* does not reside on the administration volume of the workstation *station*.

(165) OBJECT_IS_NOT_ON_MASTER_VOLUME_OF_REPLICA_SET (*replica_set*, *object*) The object *object* does not reside on the master volume of the replica set *replica_set*.

(166) OBJECT_IS_NOT_REPLICABLE (*object*) The type of the object *object* is one of the predefined types "process", "activity", "pipe", "message_queue", "volume", "audit_file", "accounting_log", "device", and "execution_site", or a descendant of one of them

(167) OBJECT_IS_NOT_REPLICATED_ON_VOLUME (*object*, *volume*) The object *object* is not a replicated object with a master or copy replica on the volume *volume*.

(168) OBJECT_IS_OF_WRONG_TYPE (*reference*) The object type of the object *reference* is not the object type required by the operation, nor a descendant of that type.

(169) OBJECT_IS_OPERATED_ON (*object*, *scope*) One or more objects of the scope defined by *object* and *scope* which is to be unlocked is currently being operated on.

(170) OBJECT_IS_PREDEFINED_REPLICATED (*object*) The object *object* is a predefined replicated object.

(171) OBJECT_IS_REPLICATED (*object*) The object *object* is either a master or copy of a replicated object.

(172) OBJECT_IS_STABLE (*object*) The object *object* is the destination of an atomically stabilizing link, or is a component of the destination of a compositely stabilizing link.

(173) OBJECT_LABEL_CANNOT_BE_CHANGED_IN_TRANSACTION (*object*) The object *object* is a message queue or a pipe and an attempt is being made to change the confidentiality or integrity label of *object* on behalf of a transaction.

(174) OBJECT_OWNER_CONSTRAINT_WOULD_BE_VIOLATED (*object*) An attempt to change the atomic or composite ACL of *object* would result in an inconsistency with the OWNER rights on *object* or an outer object of *object*.

(175) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL (*object*) An attempt has been made to set up an atomic ACL for newly created *object* with CONTROL_DISCRETIONARY mode not granted for a discretionary group which is the default object owner and which therefore has OWNER mode granted in the composite ACL.

(176) OBJECT_REFERENCE_IS_INTERNAL (*reference*) *reference* is an internal object reference.

(177) OBJECT_REFERENCE_IS_INVALID (*reference*) The reference *reference* designates an object which has been moved, archived, or deleted.

(178) OBJECT_REFERENCE_IS_UNSET (*reference*) The object reference *reference* has never been set or has been explicitly unset by REFERENCE_UNSET.

(179) OBJECT_TYPE_IS_ALREADY_IN_DESTINATION_SET (*link_type*, *object_type*, *sds*) The object type *object_type* is already in the destination object types of *link_type* in the SDS *sds*.

(180) OBJECT_TYPE_IS_INVALID (*type*) The object type *type* is "volume", "device", or a descendant of "volume" or "device".

(181) OBJECT_TYPE_IS_NOT_IN_DESTINATION_SET (*link_type*, *object_type*, *sds*) The object type *object_type* is not in the destination object types of *link_type* in the SDS *sds*.

(182) OBJECT_TYPE_IS_NOT_VISIBLE (*reference*) The object type identified by *reference* is not a visible object type.

(183) OBJECT_TYPE_IS_UNKNOWN (*type*) *type* is not an object type in the current working schema.

(184) OBJECT_TYPE_WOULD_HAVE_NO_PARENT_TYPE (*parents*) An object type would be created with an empty set *parents* of parent types.

(185) OPEN_KEY_IS_INVALID (*key*). The key *key* is not a valid open key because there already exists an "open_object" link with key *key* from the calling process, or because it is greater than MAX_OPEN_OBJECTS_PER_PROCESS.

(186) OPENING_MODE_IS_INVALID (*object, opening_mode*) *opening_mode* is not compatible with the type of *object*

(187) OPERATION_HAS_TIMED_OUT The duration of the operation has exceeded the time out of the calling process at the time the operation was called.

(188) OPERATION_IS_INTERRUPTED The current process has been interrupted while executing the current operation.

(189) OPERATION_IS_NOT_ALLOWED_ON_TYPE (*reference*) The operation has used a type which is not visible and the operation is one of the operations where its usage is not allowed, even when PCTE_CONFIGURATION is an effective group of the process.

(190) OPERATION_METHOD_CANNOT_BE_FOUND (*operation*) The method related to the operation *operation* cannot be found in the method repository.

(191) OPERATION_METHOD_CANNOT_BE_ACTIVATED (*operation*) The method related to the operation *operation* cannot be activated.

(192) PARENT_BASIC_TYPES_ARE_MULTIPLE (*parent_types*) *parent_types* contains types which are or are descended from at least two different types from the following list: "system-file", "system-pipe", "system-device", "system-volume", "system-message_queue", "system-process", "system-activity", "security-audit_file", "accounting-accounting_log"; or which are or are descended from at least two different types from the following list: "security-user", "security-user_group", "security-program_group".

(193) PATHNAME_SYNTAX_IS_WRONG (pathname) The syntax of the text value *pathname* is not that of a pathname.

(194) PIPE_HAS_NO_WRITERS (*contents*) An attempt has been made to read the blocking pipe *contents* and no contents handle was open in APPEND_ONLY mode when the operation was called.

(195) POSITION_HANDLE_IS_INVALID (*position_handle, contents_handle*) *position_handle* does not designate a valid position in the sequence of octets designated by *contents_handle*.

(196) POSITION_IS_INVALID (*position*) The position *position* is less than FIRST.

(197) POSITIONING_IS_INVALID (*contents, positioning*) The required positioning *positioning* is invalid for the specified file or device.

(198) PREFERENCE_DOES_NOT_EXIST (*object, reference*) Evaluation of the link reference *reference* relative to the origin object *object* requires some preferred key attributes or some preferred link type to be specified, but the required preference information is not provided by *object*.

(199) PREFERRED_LINK_KEY_IS_BAD (*object, link, string*) The string *string* is unsuitable as the preferred key type for the link *link* of the object *object*; it supplies too many or too few key attribute types, supplies key value attributes of the wrong types, or is syntactically invalid for a preferred key.

(200) PREFERRED_LINK_TYPE_IS_UNSET (*object*) The preferred link type of the object *object* is already unset.

(201) PRIVILEGE_IS_NOT_GRANTED (*group*) The predefined security group *group* is not effective for the calling process.

(202) PROCESS_CONFIDENTIALITY_IS_NOT_DOMINATED (*confidentiality_label, process*) The confidentiality label of the process *process* is not dominated by *confidentiality_label*.

(203) PROCESS_FILE_SIZE_LIMIT_WOULD_BE_EXCEEDED (*data, contents*) The writing of *data* to the file *contents* would cause *contents* to exceed the process file size limit for the calling process.

(204) PROCESS_HAS_NO_UNTERMINATED_CHILD The calling process has no child process.

(205) PROCESS_INTEGRITY_DOES_NOT_DOMINATE (*integrity_label, process*) The integrity label of the process *process* does not dominate *integrity_label*.

(206) PROCESS_IS_IN_TRANSACTION The calling process is currently running in a transaction or the calling process is currently running in an activity nested in a transaction.

(207) PROCESS_IS_INITIAL_PROCESS (*process*) The process *process* is the initial process of a workstation, and so cannot be terminated.

(208) PROCESS_IS_NOT_ANCESTOR (*process*) The process *process* is neither the calling process nor an ancestor of the calling process.

(209) PROCESS_IS_NOT_CHILD (*process*) The process *process* is not a child of the calling process.

(210) PROCESS_IS_NOT_TERMINABLE_CHILD (*process*) The process *process* is either in a READY state or is not a child of the current process.

(211) PROCESS_IS_NOT_THE_CALLER (*process*) The process *process* is not the calling process.

(212) PROCESS_IS_THE_CALLER (*process*) The process *process* is the calling process.

(213) PROCESS_IS_UNKNOWN (*process*) The process *process* has process status UNKNOWN.

(214) PROCESS_LABELS_WOULD_BE_INCOMPATIBLE(*user*) The determination of the new mandatory labels of the calling process has failed because the new label values are incompatible with the clearance of user *user* or the security ranges of the workstation or volume on which the calling process is located (see 13.4.11).

(215) PROCESS_LACKS_REQUIRED_STATUS (*process*, *status*) The process *process* has not got the required status *status* (or any of the list of statuses *status*), other than UNKNOWN.

(216) PROCESS_TERMINATION_IS_ALREADY_ACKNOWLEDGED (*process*) The acknowledged termination of *process* is already **true**.

(217) PROFILING_IS_NOT_SWITCHED_ON Profiling is not yet switched on for the calling process.

(218) PROGRAM_GROUP_IS_NOT_EMPTY (*group*) There are static contexts in the program group *group* so that it cannot be removed.

(219) RANGE_IS_OUTSIDE_RANGE (*device1*, *device2*) The mandatory security range of the device *device1* is or would be not entirely enclosed within the mandatory security range of the multi-level secure device *device2*. More precisely, one of the following does not hold:

CONFIDENTIALITY_RANGE_WITHIN_RANGE (*device1*, *device2*)

INTEGRITY_RANGE_WITHIN_RANGE (*device1*, *device2*)

(220) REFERENCE_CANNOT_BE_ALLOCATED There is not enough space to create an internal reference.

(221) REFERENCE_NAME_IS_INVALID (*reference_name*) *reference_name* is not a valid referenced object name.

(222) REFERENCED_OBJECT_IS_NOT_MUTABLE (*referenced_object*) The referenced object *referenced_object* is non-mutable (e.g. $common_root, $static_context).

(223) REFERENCED_OBJECT_IS_UNSET(*reference*) The object reference *reference* contains a referenced object name which is neither the key of a "referenced_object" link from the calling process, nor an alias.

(224) RELATIONSHIP_TYPE_PROPERTIES_ARE_INCONSISTENT (*forward_properties*, *backward_properties*) The forward link type properties *forward_properties* and the reverse link type properties *backward_properties* are inconsistent with the properties of a link type and its reverse link type (see 8.3.3).

(225) REPLICA_SET_COPY_IS_NOT_EMPTY (*replica_set*, *volume*) The volume *volume* contains copies of objects in the replica set *replica_set* other than a copy of *replica_set* itself.

(226) REPLICA_SET_HAS_COPY_VOLUMES (*replica_set*) The replica set *replica_set* has at least one associated copy volume.

(227) REPLICA_SET_IS_NOT_EMPTY (*replica_set*) The replica set *replica_set* contains masters of objects other than the master of *replica_set* itself.

(228) REPLICA_SET_IS_NOT_KNOWN (*replica_set*) The replica set *replica_set* is not known within the replica set directory.

(229) REPLICATED_COPY_IS_IN_USE (*object*) The copy object *object* has a usage designation link.

(230) REPLICATED_COPY_UPDATE_IS_FORBIDDEN (*object*, *scope*) An object of the scope defined by *object* and *scope* is replicated and the calling process is attempting to update a copy.

(231) RESOURCE_GROUP_IS_KNOWN (*object*) The object *object* is already known by the PCTE installation as a valid resource group.

(232) RESOURCE_GROUP_IS_UNKNOWN (*group*) *group* does not identify a known resource group of the accounting directory.

(233) REVERSE_KEY_IS_BAD (*origin*, *link*, *destination*, *reverse_key*) The link *link* of the object *origin* is reversed by a link of cardinality many for which *reverse_key* supplies either too many or too few key attribute values or supplies key attribute values of the wrong value type.

(234) REVERSE_KEY_IS_NOT_SUPPLIED (*origin*, *link*, *destination*) The link *link* of the object *origin* has a reverse link of cardinality many for which no key can be inferred, i.e. it is not an implicit link, *destination* does not have a preferred key specified for its type, and no specific key value is provided as a parameter to the operation.

(235) REVERSE_KEY_IS_SUPPLIED (*reverse_key*) A reverse key *reverse_key* is supplied for the reverse of a created link though the reverse link is of category IMPLICIT or cardinality one..

(236) REVERSE_LINK_EXISTS (*origin*, *link*, *destination*, *reverse_key*) The link *link* of the object *origin* is reversed by a link of cardinality many for which *reverse_key* would result in the creation of a link which already exists in the links of *destination*.

(237) SDS_IS_IN_A_WORKING_SCHEMA (*sds_name*) The SDS *sds_name* is currently included in a working schema (i.e there is an "in_working_schema_of" link from *sds_name*)

(238) SDS_IS_KNOWN (*sds*) The SDS *sds* is already known to the PCTE installation.

(239) SDS_IS_NOT_EMPTY_NOR_VERSION (*sds*) The "sds" object *sds* is neither empty (i.e. has no types in SDS) nor a version of a known SDS. An object is a version of another object if related by a series of "predecessor" or a series of "successor" links.

(240) SDS_IS_NOT_IN_WORKING_SCHEMA (*sds*) The SDS *sds* is not in the current working schema.

(241) SDS_IS_PREDEFINED (*sds*) The SDS *sds* is predefined and cannot be changed.

(242) SDS_IS_UNDER_MODIFICATION (*sds_name*) The SDS *sds_name* contains typing information currently being modified in an uncommitted non-enclosing transaction.

(243) SDS_IS_UNKNOWN (*sds*) The SDS designator *sds* does not identify a known SDS in the SDS directory.

(244) SDS_NAME_IS_DUPLICATE (*sds_name*) There is already an SDS of the name *sds_name*.

(245) SDS_NAME_IS_INVALID (*name*) *name* is not valid for an SDS name.

(246) SDS_WOULD_APPEAR_TWICE_IN_WORKING_SCHEMA (*sds_sequence*) *sds_sequence* has two SDS names of the same "sds" object. This would result in an invalid working schema.

(247) SECURITY_GROUP_ALREADY_HAS_THIS_SUBGROUP (*group*, *subgroup*) *subgroup* is already a subgroup of *group*.

(248) SECURITY_GROUP_IS_ALREADY_ENABLED (*group*, *class*) *group* is already enabled for confidentiality downgrade/integrity upgrade from *class*.

(249) SECURITY_GROUP_IS_IN_USE (*group*) The group *group* is effective for a process.

(250) SECURITY_GROUP_IS_KNOWN (*group*) *group* is already defined as a security group.

(251) SECURITY_GROUP_IS_NOT_A_SUBGROUP (*subgroup*, *group*) *subgroup* is not a subgroup of *group*.

(252) SECURITY_GROUP_IS_NOT_ADOPTABLE (*user_group*, *process*) *user_group* is not an adoptable user group for the process *process*.

(253) SECURITY_GROUP_IS_NOT_ENABLED (*group*, *class*) *group* is not enabled for confidentiality downgrade/integrity upgrade from *class*.

(254) SECURITY_GROUP_IS_PREDEFINED (*group*) *group* is one of the predefined security groups (see 19.1.1).

(255) SECURITY_GROUP_IS_REQUIRED_BY_OTHER_GROUPS (*group*) *group* has subgroups or is a subgroup or (for a user group or program group) has group members or (for a user) is a member of user groups.

(256) SECURITY_GROUP_IS_UNKNOWN (*group*) *group* does not identify a security group known to the PCTE installation i.e. there is no link from the security group directory.

(257) SECURITY_GROUP_WOULD_BE_IN_INVALID_GRAPH (*subgroup*, *group*) *subgroup* may not become a subgroup of *group* since this would result in a graph of security groups which is not a directed acyclic graph.

(258) SECURITY_POLICY_WOULD_BE_VIOLATED A violation of the built-in security policy of the PCTE implementation (see 20.1.8) has been attempted.

(259) STATIC_CONTEXT_CONTENTS_CANNOT_BE_EXECUTED (*static_context*, *station*) The contents of the static context *static_context* cannot be executed on the workstation *station*.

(260) STATIC_CONTEXT_IS_ALREADY_MEMBER (*program*, *group*) The static context *program* is already a member of the program group *group*.

(261) STATIC_CONTEXT_IS_BEING_WRITTEN (*static_context*) The object contents of *static_context* is open for writing.

(262) STATIC_CONTEXT_IS_IN_USE (*object*) *object* is a static context that is being executed or interpreted.

(263) STATIC_CONTEXT_IS_NOT_MEMBER (*program*, *program_group*) *program* is not a member of the program group *program_group*.

(264) STATIC_CONTEXT_REQUIRES_TOO_MUCH_MEMORY (*static_context*) Execution of *static_context* requires more memory than is available.

(265) STATUS_IS_BAD (*status*) The value of *status* is neither CONNECTED nor CLIENT.

(266) TIME_CANNOT_BE_CHANGED The system time cannot be changed.

(267) TRANSACTION_CANNOT_BE_COMMITTED The activity to be committed is a transaction and the system cannot commit the updates made on behalf of this transaction. Note that this error implies an ABORT_ACTIVITY. This may be due, for example, to network failure, system crash, disk crash, disk physically switched to a read only protection etc. The circumstances under which this occurs are implementation-defined.

(268) TYPE_CANNOT_BE_APPLIED_TO_LINK_TYPE (*link_type*, *attribute_type*). The attribute type *attribute_type* cannot be applied to the link type *link_type* because it is the "object_on_volume" link type or a usage designation or service designation link type.

(269) TYPE_HAS_DEPENDENCIES (*sds*, *type*) The type *type* cannot be deleted because it has type dependencies and its deletion would violate SDS well-formedness rules, i.e. within the SDS *sds*:

- for an attribute type, it is applied to an object or link type;

- for a link type, it is applied to an object type, or it has a non-empty set of destination link types, or non-key attribute types are applied to it; or the same is true of its reverse link type, if any;
- for an object type, it has a child type, or there is an attribute or link type applied to it;
- for an enumeral type, it is be associated with an enumeration attribute type.

(270) TYPE_HAS_NO_LOCAL_NAME (*sds*, *type_reference*) The type in sds identified by *type_reference* does not have a local name in the SDS *sds*.

(271) TYPE_IDENTIFIER_IS_INVALID (*identifier*) *identifier* is not a valid type identifier.

(272) TYPE_IDENTIFIER_USAGE_IS_INVALID (*reference*) The operation does not allow a type identifier such as *reference* to be used.

(273) TYPE_IS_ALREADY_APPLIED (*sds*, *type*, *definition_type*) In the SDS *sds*, the type (either an attribute type or a link type) *type* is already applied to *definition_type* (either a link type or an object type).

(274) TYPE_IS_ALREADY_CONSTRAINED (*sds*, *parameter_type*) The parameter type *parameter_type* is already constrained to an attribute type, an object type, or an interface type.

(275) TYPE_IS_ALREADY_KNOWN_IN_SDS (*type*, *to_sds*) The type *type* is already defined in the SDS *sds*.

(276) TYPE_IS_NOT_APPLIED (*sds*, *type1*, *type2*) The type *type1* is not applied to the type *type2* in the SDS *sds*.

(277) TYPE_IS_NOT_DESCENDANT (*type*, *other_type*) The object type *other_type* is neither *type* nor a descendant object type of *type*.

(278) TYPE_IS_NOT_VISIBLE (*type*) The type reference or type name *type* is not visible.

(279) TYPE_IS_OF_WRONG_KIND (*reference*) The reference *reference* identifies a type which is not of the kind expected by the operation (object type, link type, attribute type, or enumeral type).

(280) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*) The type *type* is not defined in the SDS *sds*.

(281) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds_name*, *type_name*) The name *type_name* duplicates a local name of a type already defined in the SDS *sds_name*.

(282) TYPE_NAME_IS_INVALID (*type_name*) *type_name* does not conform to the syntax of a type name.

(283) TYPE_OF_OBJECT_IS_INVALID (*object*, *scope*) The type of one or more of the objects of the scope defined by *object* and *scope* is "volume", "device", or a descendant of "volume" or "device".

(284) TYPE_OF_PARAMETER_IS_WRONG (*operation, parameter*) The type of the value of the parameter *parameter* does not match that defined by the signature of the operation *operation*.

(285) TYPE_REFERENCE_IS_INVALID (*reference*) The type in working schema referenced by the evaluated type reference *reference* no longer exists as a result of a call to PROCESS_SET_WORKING_SCHEMA.

(286) TYPE_REFERENCE_IS_UNSET (*reference*) The type reference *reference* has never been set or has been explicitly unset by TYPE_REFERENCE_UNSET.

(287) UNLOCKING_IN_TRANSACTION_IS_FORBIDDEN To perform the operation would unset or reset a lock within a transaction.

(288) UPPER_BOUND_WOULD_BE_VIOLATED (*object*, *link*) The creation of *link* from its origin *object* would violate the upper bound of the link type of *link*.

(289) USAGE_MODE_ON_ATTRIBUTE_TYPE_WOULD_BE_VIOLATED (*object*, [*link*], *attribute*, *mode_value*) The usage mode of the type in working schema of the attribute *attribute* of the object *object* or of the link *link* of the object *object* does not include *mode_value*.

(290) USAGE_MODE_ON_LINK_TYPE_WOULD_BE_VIOLATED (*object*, *link*, *mode_value*) The usage mode of the type in working schema of the link *link* of the object *object* does not include *mode_value*.

(291) USAGE_MODE_ON_OBJECT_TYPE_WOULD_BE_VIOLATED (*initial_type*, *type*) The usage mode of *initial_type*, *type*, or any type in working schema which is a descendant of *initial_type* and an ancestor of *type* does not include CREATE_MODE.

(292) USER_CRITERION_IS_NOT_SELECTED (*criterion*) The event to be removed in *criterion* is not in the list of user-dependent criteria.

(293) USER_GROUP_IS_IN_USE (*user*, *group*) A process executing on behalf of *user* currently has the group *group* as its adopted group.

(294) USER_GROUP_LACKS_ALL_USERS_AS_SUPERGROUP (*group*) The predefined user group ALL_USERS is not a supergroup of *group*.

(295) USER_GROUP_WOULD_NOT_HAVE_ALL_USERS_AS_SUPERGROUP (*group*) Deleting the "user_subgroup_of" link from *group* would mean that the predefined group ALL_USERS would no longer be a supergroup of *group*; and either *group* has users as members, or there is a group G which has members and which has *group* as one of its supergroups and G would no longer have ALL_USERS as a supergroup.

(296) USER_IS_ALREADY_CLEARED_TO_CLASS (*group*, *class*) The user group *group* is already cleared to a class which dominates or is dominated by *class*.

(297) USER_IS_ALREADY_MEMBER (*user*, *group*) The user *user* is already a member of *group*.

(298) USER_IS_IN_USE (*user*) The user *user* is the destination of a "user_identity" link from a process.

(299) USER_IS_NOT_CLEARED (*process*, *mandatory_label*) The user group on whose behalf *process* is executing does not have an overall clearance to the level given by the label *mandatory_label*. More precisely:

LABEL_DOMINATES (user confidentiality clearance of *process*, *mandatory_label*)

(300) USER_IS_NOT_CLEARED_TO_CLASS (*group*, *class*) The user group *group* is not cleared to mandatory class *class* nor to a mandatory class dominating *class*.

(301) USER_IS_NOT_MEMBER (*user*, *user_group*) *user* is not a member of the user group *user_group*.

(302) USER_IS_UNKNOWN (*object*) The user designated by *object* in a user-dependent criterion is not a known user, i.e. it is not a known security group which is of the child type "user" nor of one of its descendent types.

(303) VALUE_TYPE_IS_INVALID (*value*, *object*, [*link*], *attribute*) The type of *value* is not of the value type of the associated type of the attribute *attribute* of the object *object* or of the link *link* of the object *object*.

(304) VERSION_GRAPH_IS_INVALID (*version*, *predecessor*) Adding *predecessor* as a predecessor of *version* would render its version graph invalid; i.e. not a directed acyclic graph.

(305) VERSION_IS_REQUIRED (*version*, *scope*) One of the objects specified by *version* and *scope* cannot be removed as it has no successors and there are no incoming composition or existence links other than the predecessor link from one of the objects to at least one of its predecessors.

(306) VOLUME_CANNOT_BE_MOUNTED_ON_DEVICE (*volume*, *device*) The device *device* is inappropriate for the mounting of the volume *volume*.

(307) VOLUME_EXISTS (*volume_identifier*) The specified volume number *volume_identifier* corresponds to an existing volume.

(308) VOLUME_HAS_OBJECT_OUTSIDE_RANGE (*volume*, *high_label*, *low_label*) The volume *volume* has one or more objects whose mandatory label lies outside the range *low_label* to *high_label*.

(309) VOLUME_HAS_OBJECTS_IN_USE (*volume_identifier*) The volume *volume_identifier* cannot be unmounted. This could occur if:

- some resources (objects or links) residing on that volume are currently locked;
- some "sds", "type_in_sds", or "type" objects residing on the volume are currently included in some working schemas;
- a message queue object residing on the volume is associated with a reserved message queue;
- some static contexts residing on the volume are being currently executed or are currently used as an interpreter;
- some process objects residing on the volume are associated with processes currently running or some activity objects residing on the volume are associated with non- terminated activities.

(310) VOLUME_HAS_OTHER_LINKS (*volume*) There are links starting from the volume *volume* which are not the "object_on_volume" link to itself, the "mounted_on" link to the device on which the volume is mounted, or the reverse of the link from the volume directory.

(311) VOLUME_HAS_OTHER_OBJECTS (*volume*) The volume *volume* has other objects residing on it apart from the volume object representing the volume itself.

(312) VOLUME_IDENTIFIER_IS_INVALID (*volume_identifier*) *volume_identifier* is not a valid volume number.

(313) VOLUME_IS_ADMINISTRATION_VOLUME (*volume*) The volume *volume* is an administration volume.

(314) VOLUME_IS_ALREADY_COPY_VOLUME_OF_REPLICA_SET (*replica_set*, *volume*) The volume *volume* is already a copy volume of the replica set *replica_set*.

(315) VOLUME_IS_ALREADY_MOUNTED (*volume*) The volume *volume* is already mounted.

(316) VOLUME_IS_FULL (*volume*) There is insufficient space for one of the objects or for one of the links to be created on the volume *volume*.

(317) VOLUME_IS_INACCESSIBLE (*volume*) The volume *volume* is not accessible (see 18.1.5).

(318) VOLUME_IS_MASTER_VOLUME_OF_REPLICA_SET (*replica_set*, *volume*) The volume *volume* is already the master volume of the replica set replica_set.

(319) VOLUME_IS_NOT_COPY_VOLUME_OF_REPLICA_SET (*replica_set*, *volume*) The volume *volume* is not a copy volume of the replica set *replica_set*.

(320) VOLUME_IS_NOT_MASTER_OR_COPY_VOLUME_OF_REPLICA_SET (*replica_set*, *volume*) The volume *volume* is neither the master nor a copy volume of the replica set *replica_set*.

(321) VOLUME_IS_READ_ONLY (*volume*) The volume *volume* is mounted as a read-only volume.

(322) VOLUME_IS_UNKNOWN (*volume*) The "volume" object *volume* is not linked to the volume directory.

(323) WORKSTATION_EXISTS (*identifier*) A workstation already exists in the specified administration volume with the execution site identifier *identifier*.

(324) WORKSTATION_HAS_NO_CHOICE_OF_VOLUME_FOR_REPLICA_SET (*station*, *replica_set*) The workstation *station* has no chosen volume for accessing replica set *replica_set*.

(325) WORKSTATION_IDENTIFIER_IS_INVALID (*identifier*) The natural *identifier* is not a valid execution site identifier.

(326) WORKSTATION_IS_BUSY (*station*) The workstation *station* is busy.

(327) WORKSTATION_IS_CONNECTED (*station*) The connection status of the workstation *station* is not LOCAL or AVAILABLE.

(328) WORKSTATION_IS_NOT_CONNECTED (*station*) The status of the workstation *station* is LOCAL or AVAILABLE.

(329) WORKSTATION_IS_UNKNOWN (*station*) The workstation *station* is unknown to the PCTE installation (i.e. it is not linked to the execution site directory with a "known_execution_site" link).

Annex D
(normative)

Auditable Events

D.1 Selectable events

D.1.1 Selectable event type = WRITE

ACCOUNTING_LOG_COPY_AND_RESET
ACCOUNTING_OFF
ACCOUNTING_ON
ACCOUNTING_RECORD_WRITE
AUDIT_FILE_COPY_AND_RESET
CONSUMER_GROUP_DELETE
CONSUMER_GROUP_INITIALIZE
CONTENTS_COPY_FROM_FOREIGN_SYSTEM
CONTENTS_OPEN
CONTENTS_SET_POSITION
CONTENTS_SET_PROPERTIES
DEVICE_CREATE
DEVICE_REMOVE
LINK_CREATE
LINK_DELETE
LINK_DELETE_ATTRIBUTE
LINK_REPLACE
LINK_RESET_ATTRIBUTE
LINK_SET_ATTRIBUTE
LINK_SET_SEVERAL_ATTRIBUTES
MESSAGE_SEND_NO_WAIT
MESSAGE_SEND_WAIT
NOTIFY_DELETE
NOTIFY_SWITCH_EVENTS
OBJECT_CONVERT
OBJECT_CREATE
OBJECT_DELETE
OBJECT_DELETE_ATTRIBUTE
OBJECT_RESET_ATTRIBUTE
OBJECT_SET_ATTRIBUTE
OBJECT_SET_PREFERENCE
OBJECT_SET_SEVERAL_ATTRIBUTES
OBJECT_SET_TIME_ATTRIBUTES
PROCESS_SET_ADOPTABLE_FOR_CHILD
PROCESS_SET_ALARM
PROCESS_SET_CONSUMER_IDENTITY
PROCESS_SET_DEFAULT_ACL_ENTRY
PROCESS_SET_DEFAULT_OWNER
PROCESS_SET_OPERATION_TIME_OUT
PROCESS_UNSET_CONSUMER_IDENTITY

QUEUE_SET_TOTAL_SPACE
RESOURCE_GROUP_ADD_OBJECT
RESOURCE_GROUP_DELETE
RESOURCE_GROUP_INITIALIZE
RESOURCE_GROUP_REMOVE_OBJECT
SDS_ADD_DESTINATION
SDS_APPLY_ATTRIBUTE_TYPE
SDS_APPLY_LINK_TYPE
SDS_CREATE_BOOLEAN_ATTRIBUTE_TYPE
SDS_CREATE_ENUMERATION_ATTRIBUTE_TYPE
SDS_CREATE_ENUMERAL_TYPE
SDS_CREATE_FLOAT_ATTRIBUTE_TYPE
SDS_CREATE_INTEGER_ATTRIBUTE_TYPE
SDS_CREATE_LINK_TYPE
SDS_CREATE_NATURAL_ATTRIBUTE_TYPE
SDS_CREATE_OBJECT_TYPE
SDS_CREATE_RELATIONSHIP_TYPE
SDS_CREATE_STRING_ATTRIBUTE_TYPE
SDS_CREATE_TIME_ATTRIBUTE_TYPE
SDS_DELETE_TYPE
SDS_INITIALIZE
SDS_REMOVE
SDS_REMOVE_DESTINATION
SDS_SET_ENUMERAL_TYPE_IMAGE
SDS_SET_TYPE_MODES
SDS_SET_TYPE_NAME
SDS_UNAPPLY_ATTRIBUTE_TYPE
SDS_UNAPPLY_LINK_TYPE
VERSION_ADD_PREDECESSOR
VERSION_REMOVE
VERSION_REMOVE_PREDECESSOR
VOLUME_CREATE
VOLUME_DELETE

D.1.2 Selectable event type = READ

ACCOUNTING_LOG_READ
CONTENTS_COPY_TO_FOREIGN_SYSTEM
CONTENTS_GET_POSITION
CONTENTS_OPEN
GROUP_GET_IDENTIFIER
LINK_GET_ATTRIBUTE
LINK_GET_DESTINATION_ARCHIVE
LINK_GET_DESTINATION_VOLUME
LINK_GET_KEY
LINK_GET_SEVERAL_ATTRIBUTES
MESSAGE_DELETE
MESSAGE_PEEK
MESSAGE_RECEIVE_NO_WAIT
MESSAGE_RECEIVE_WAIT

OBJECT_CHECK_PERMISSION
OBJECT_CHECK_TYPE
OBJECT_GET_ACL
OBJECT_GET_ATTRIBUTE
OBJECT_GET_PREFERENCE
OBJECT_GET_SEVERAL_ATTRIBUTES
OBJECT_GET_TYPE
OBJECT_LIST_LINKS
OBJECT_LIST_VOLUMES
PROCESS_ADD_BREAKPOINT
PROCESS_CONTINUE
PROCESS_GET_WORKING_SCHEMA
PROCESS_PEEK
PROCESS_POKE
PROCESS_PROFILING_OFF
PROCESS_PROFILING_ON
PROCESS_REMOVE_BREAKPOINT
QUEUE_EMPTY
QUEUE_RESERVE
QUEUE_UNRESERVE
SDS_GET_ATTRIBUTE_TYPE_PROPERTIES
SDS_GET_ENUMERAL_TYPE_IMAGE
SDS_GET_ENUMERAL_TYPE_POSITION
SDS_GET_LINK_TYPE_PROPERTIES
SDS_GET_NAME
SDS_GET_OBJECT_TYPE_PROPERTIES
SDS_GET_TYPE_KIND
SDS_GET_TYPE_MODES
SDS_GET_TYPE_NAME
SDS_SCAN_ATTRIBUTE_TYPE
SDS_SCAN_ENUMERAL_TYPE
SDS_SCAN_LINK_TYPE
SDS_SCAN_OBJECT_TYPE
SDS_SCAN_TYPES
VERSION_IS_CHANGED
VERSION_TEST_ANCESTRY
VERSION_TEST_DESCENT
VOLUME_LIST_OBJECTS

D.1.3 Selectable event type = COPY

ACCOUNTING_LOG_COPY_AND_RESET
OBJECT_COPY
QUEUE_RESTORE
QUEUE_SAVE
SDS_IMPORT_ATTRIBUTE_TYPE
SDS_IMPORT_ENUMERAL_TYPE
SDS_IMPORT_LINK_TYPE
SDS_IMPORT_OBJECT_TYPE

VERSION_REVISE
VERSION_SNAPSHOT

D.1.4 Selectable event type = ACCESS_CONTENTS

CONTENTS_READ
CONTENTS_TRUNCATE
CONTENTS_WRITE

D.1.5 Selectable event type = EXPLOIT

PROCESS_CREATE
PROCESS_CREATE_AND_START
PROCESS_SET_WORKING_SCHEMA
PROCESS_START
VOLUME_MOUNT
VOLUME_UNMOUNT

D.1.6 Selectable event type = CHANGE_ACCESS_CONTROL

OBJECT_SET_ACL_ENTRY

D.1.7 Selectable event type = CHANGE_LABEL

DEVICE_SET_CONFIDENTIALITY_RANGE
DEVICE_SET_INTEGRITY_RANGE
EXECUTION_SITE_SET_CONFIDENTIALITY_RANGE
EXECUTION_SITE_SET_INTEGRITY_RANGE
OBJECT_SET_CONFIDENTIALITY_LABEL
OBJECT_SET_INTEGRITY_LABEL
PROCESS_SET_CONFIDENTIALITY_LABEL
PROCESS_SET_FLOATING_CONFIDENTIALITY_LEVEL
PROCESS_SET_FLOATING_INTEGRITY_LEVEL
PROCESS_SET_INTEGRITY_LABEL
VOLUME_SET_CONFIDENTIALITY_RANGE
VOLUME_SET_INTEGRITY_RANGE

D.1.8 Selectable event type = VIOLATION_CONFIDENTIALITY_WRITE and VIOLATION_INTEGRITY_WRITE

All operations with mandatory security errors plus:

DEVICE_SET_CONTROL
PROCESS_INTERRUPT_OPERATION

D.1.9 Selectable event type = VIOLATION_CONFIDENTIALITY_READ and VIOLATION_INTEGRITY_READ

All operations with mandatory security errors plus:

CONTENTS_GET_HANDLE_FROM_KEY
CONTENTS_GET_KEY_FROM_HANDLE
CONTENTS_OPEN
CONTENTS_SEEK
DEVICE_GET_CONTROL
LINK_GET_REVERSE
MESSAGE_RECEIVE_NO_WAIT
MESSAGE_RECEIVE_WAIT
MESSAGE_SEND_NO_WAIT
MESSAGE_SEND_WAIT
NOTIFY_CREATE
OBJECT_IS_COMPONENT
PROCESS_SET_REFERENCED_OBJECT
PROCESS_UNSET_REFERENCED_OBJECT
QUEUE_HANDLER_DISABLE
QUEUE_HANDLER_ENABLE
VOLUME_GET_STATUS

D.1.10 Selectable event type = USE_PREDEFINED_GROUP

All uses of type identifiers for access to non-visible types (PCTE_CONFIGURATION)

All modifications to master objects (PCTE_REPLICATION)

All operations defined in 10.2

ARCHIVE_RESTORE
ARCHIVE_SAVE
DEVICE_CREATE
DEVICE_REMOVE
OBJECT_SET_TIME_ATTRIBUTES
PROCESS_SET_FILE_SIZE_LIMIT
PROCESS_SET_PRIORITY
REPLICA_SET_CREATE
REPLICA_SET_REMOVE
REPLICA_SET_ADD_COPY_VOLUME
REPLICA_SET_REMOVE_COPY_VOLUME
REPLICATED_OBJECT_CREATE
REPLICATED_OBJECT_DELETE_REPLICA
REPLICATED_OBJECT_DUPLICATE
REPLICATED_OBJECT_REMOVE
TIME_SET
VERSION_ADD_PREDECESSOR
VERSION_REMOVE
VERSION_REMOVE_PREDECESSOR
VOLUME_CREATE
VOLUME_DELETE

WORKSTATION_REDUCE_CONNECTION
WORKSTATION_CONNECT
WORKSTATION_CREATE
WORKSTATION_DELETE
WORKSTATION_DISCONNECT
WORKSTATION_SELECT_REPLICA_SET_VOLUME
WORKSTATION_UNSELECT_REPLICA_SET_VOLUME

D.1.11 Selectable event type = SET_USER_IDENTITY

PROCESS_ADOPT_USER_GROUP
PROCESS_SET_USER

D.1.12 Selectable event type = COVERT_CHANNEL

ACTIVITY_ABORT
ACTIVITY_END
ACTIVITY_START
CONTENTS_GET_POSITION
CONTENTS_READ
CONTENTS_SET_POSITION
CONTENTS_TRUNCATE
CONTENTS_WRITE
LINK_CREATE
LOCK_RESET_INTERNAL_MODE
LOCK_SET_INTERNAL_MODE
LOCK_SET_OBJECT
LOCK_UNSET_OBJECT
PROCESS_RESUME
PROCESS_SET_REFERENCED_OBJECT
PROCESS_SUSPEND
PROCESS_TERMINATE

D.2 Mandatory Events

D.2.1 Mandatory event type = CHANGE_IDENTIFICATION

OBJECT_MOVE

D.2.2 Mandatory event type = SELECT_AUDIT_EVENT

AUDIT_ADD_CRITERION
AUDIT_FILE_COPY_AND_RESET
AUDIT_REMOVE_CRITERION
AUDIT_SELECTION_CLEAR
AUDIT_SWITCH_OFF_SELECTION
AUDIT_SWITCH_ON_SELECTION

D.2.3 Mandatory event type = SECURITY_ADMINISTRATION

CONFIDENTIALITY_CLASS_INITIALIZE
GROUP_DELETE
GROUP_DISABLE_FOR_CONFIDENTIALITY_DOWNGRADE
GROUP_DISABLE_FOR_INTEGRITY_UPGRADE
GROUP_ENABLE_FOR_CONFIDENTIALITY_DOWNGRADE
GROUP_ENABLE_FOR_INTEGRITY_UPGRADE
GROUP_INITIALIZE
GROUP_RESTORE
INTEGRITY_CLASS_INITIALIZE
PROGRAM_GROUP_ADD_MEMBER
PROGRAM_GROUP_ADD_SUBGROUP
PROGRAM_GROUP_REMOVE_MEMBER
PROGRAM_GROUP_REMOVE_SUBGROUP
USER_EXTEND_CONFIDENTIALITY_CLEARANCE
USER_EXTEND_INTEGRITY_CLEARANCE
USER_GROUP_ADD_MEMBER
USER_GROUP_ADD_SUBGROUP
USER_GROUP_REMOVE_MEMBER
USER_GROUP_REMOVE_SUBGROUP
USER_REDUCE_CONFIDENTIALITY_CLEARANCE
USER_REDUCE_INTEGRITY_CLEARANCE

D.3 List of operations not audited

AUDIT_FILE_READ
AUDIT_GET_CRITERIA
AUDIT_RECORD_WRITE
AUDITING_STATUS
CONTENTS_CLOSE
NOTIFICATION_MESSAGE_GET_KEY
PROCESS_GET_DEFAULT_ACL
PROCESS_GET_DEFAULT_OWNER
PROCESS_SET_TERMINATION_STATUS
PROCESS_WAIT_FOR_ANY_CHILD
PROCESS_WAIT_FOR_CHILD
REFERENCE_COPY
REFERENCE_GET_EVALUATION_POINT
REFERENCE_GET_PATH
REFERENCE_GET_STATUS
REFERENCE_SET_ABSOLUTE
REFERENCE_SET_RELATIVE
REFERENCE_UNSET
REFERENCES_ARE_EQUAL
TIME_GET
TYPE_IDENTIFIER_CONVERT_TO_NAME
TYPE_NAME_CONVERT_TO_IDENTIFIER
WORKSTATION_STATUS

Annex E
(informative)

The Predefined Schema Definition Sets

E.1 The system SDS

```
(1)   sds system:
(2)   volume_identifier (read) non_duplicated natural;
(3)   locked_link_name: (read) string;
(3)   lock_identifier  : (read) string;
(4)   exact_identifier: (read) non_duplicated string;
(5)   number: natural;
(6)   name: string;
(7)   system_key: (read) natural;
(8)   replica_set_identifier: 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;
          replicated_as_part_of: (navigate) implicit link to replica_set reverse includes_object;
          replica_on: implicit link to administration_volume reverse replica;
      end object;
```

(10)
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;

(11)
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
reverse mounted_volume **with**
attribute
read_only: (**read**) **boolean**;
end mounted_on;
end volume;

(12)
administration_volume: (**protected**) **child type of** volume **with**
link
administration_volume_of: **non_duplicated designation link** (number) **to** workstation;
replica: (**navigate**) **reference link** (exact_identifier) **to** object **reverse** replica_on;
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;

(13)
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;

(14)
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;

(15)
positioning: (**read**) **enumeration** (SEQUENTIAL, DIRECT, SEEK):= SEQUENTIAL;

(16)
file: **child type of** object **with**
contents file;
attribute
contents_size: (**read**) **natural**;
positioning;
end file;

(17)
pipe: **child type of** object **with**
contents pipe;
end pipe;

(18)
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;

(19)
device_supporting_volume: **child type of** device **with**
link
mounted_volume: (**navigate**) **non_duplicated designation link to** volume;
end device_supporting_volume;

(20)
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;

(21)
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;

(22)
execution_site_identifier: **natural**;

(23)
execution_class: **child type of** object **with**
link
usable_execution_site: **reference link** (execution_site_identifier) **to** execution_site;
end execution_class;

(24)
inheritable: **boolean** := **true**;

(25)
referenced_object: (**navigate**) **designation link** (reference_name: **string**) **to** object **with**
attribute
inheritable;
end referenced_object;

(26)
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;

(27)
is_listener: (**navigate**) **non_duplicated designation link** (number) **to** message_queue **with**
attribute
message_types: (**read**) **string**;
end is_listener;

(28)
lock_mode: READ_UNPROTECTED, READ_SEMIPROTECTED, WRITE_UNPROTECTED,
WRITE_SEMIPROTECTED, DELETE_UNPROTECTED, DELETE_SEMIPROTECTED,
READ_PROTECTED, DELETE_PROTECTED, WRITE_PROTECTED,
WRITE_TRANSACTIONED, DELETE_TRANSACTIONED;

(29) lock_external_mode: (**read**) **enumeration** (lock_mode):= READ_UNPROTECTED;

(30) lock_internal_mode: (**read**) **enumeration** (lock_external_mode **range** READ_UNPROTECTED .. WRITE_PROTECTED):= READ_UNPROTECTED;

(31)
```
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;
    lock_external_mode;
    lock_internal_mode;
end process_waiting_for;
```

(32)
```
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;
```

(33)
```
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;
```

(34)
```
activity_class: (read) enumeration (UNPROTECTED, PROTECTED, TRANSACTION) :=
    UNPROTECTED;
```

(35)
```
activity_status: (read) non_duplicated enumeration (UNKNOWN, ACTIVE, COMMITTING,
    ABORTING, COMMITTED, ABORTED):= UNKNOWN;
```

(36)
```
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;
    lock: (navigate) non_duplicated designation link (number) 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;
```

(37)
```
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;
```

(38) 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;
includes_object: (**navigate**) **reference link** (exact_identifier) **to** object **reverse** replicated_as_part_of;
end replica_set;

(39) 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;

(40) execution_site: **child type of** object **with**
link
running_process: (**navigate**) **non_duplicated designation link** (number) **to** process;
end execution_site**;**

(41) 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;
replica_set_chosen_volume: (**navigate**) **designation link** (replica_set_identifier) **to** administration_volume;
end workstation;

(42) 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;

(43) 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;

(44) **end** system;

(45) The fine-grain objects module requires the following extensions to the system SDS:

(46) **sds** system:

(47)
```
extend object with
attribute
    cluster_identifier: (read) non_duplicated natural;
end object;
```

(48)
```
extend object type volumewith
link
    known_cluster: (navigate) non_duplicated existence link (cluster_identifier) to cluster
        reverse cluster_in_volume;
end volume;
```

(49)
```
cluster: child type of object with
attribute
    cluster_characteristics: (read) string;
link
    object_in_cluster: (navigate) non_duplicated designation link (exact_identifier) to object;
    cluster_in_volume: (navigate) implicit link to volume reverse known_cluster;
end cluster;
```

(50) **end** system;

(51) The object-orientation module requires the following extensions to the system SDS:

(52) **sds** system:

(53) exec_class_name: **string**;

(54) operation_id: (**read**) **string**;

(55) exploits: (**navigate**) **designation link** (name) **to** sds;

(56)
```
tool: child type of object with
link
    external_component_of: (navigate) reference link (number) to tool reverse
        external_component;
    executable: (navigate) reference link (exec_class_name) to static_context reverse
        implementing_tool;
    exploits;
    has_map: (navigate) reference link (number) to method_selection reverse map_used_by;
component
    external_component: (navigate) composition link (number) to tool reverse
        external_component_of;
    internal_component: (navigate) composition link (number) to module reverse
        internal_component_of;
end tool;
```

(57)
```
module: child type of object with
link
    internal_component_of: (navigate) reference link (number) to tool reverse
        internal_component;
    exploits;
    linkable: (navigate) reference link (exec_class_name) to linkable_library reverse linkable_to;
end module;
```

(58)
```
linkable_library: child type of file with
link
    linkable_to: implicit link (system_key) to module reverse linkable;
end linkable_library;
```

```
(59)  method_selection: child type of file with
      link
          realized_by: (navigate) reference link (number; operation_id; type_identifier) to
              method_actions reverse realizes;
          map_used_by: (navigate) implicit link (system_key) to tool reverse has_map;
      end method_selection;

(60)  method_actions: child type of file with
      link
          implemented_by: (navigate) designation link (number) to tool, module;
          realizes: (navigate) implicit link (system_key) to method_selection reverse realized_by;
      end method_actions;

(61)  dispatching_context: child type of file;

(62)  extend object type process with
      link
          has_dispatching_context: (navigate) designation link to dispatching_context;
      end process;

(63)  extend object type static_context with
      link
          implementing_tool: (navigate) implicit link to tool reverse executable;
      end static_context;

(64)  end system;
```

E.2 The metasds SDS

```
(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: 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) 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;

(9) 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;

(10) usage_mode: (**read**) **natural**;

(11) export_mode: (**read**) **natural**;

(12) maximum_usage_mode: (**read**) **natural**;

(13) 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;

(14) 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;

(15) duplication: (**read**) **enumeration** (DUPLICATED, NON_DUPLICATED):= DUPLICATED;

(16) key_attribute_of: (**navigate**) **implicit link** (system_key) **to** link_type **reverse** key_attribute;

(17) attribute_type: (**protected**) **child type of** type **with**
attribute
duplication;
end attribute_type;

(18) string_attribute_type: (**protected**) **child type of** attribute_type **with**
attribute
string_initial_value: (**read**) **string**;
link
key_attribute_of;
end string_attribute_type;

(19) integer_attribute_type: (**protected**) **child type of** attribute_type **with**
attribute
integer_initial_value: (**read**) **integer**;
end integer_attribute_type;

(20) natural_attribute_type: (**protected**) **child type of** attribute_type **with**
attribute
natural_initial_value: (**read**) **natural**;
link
key_attribute_of;
end natural_attribute_type;

(21) float_attribute_type: (**protected**) **child type of** attribute_type **with**
attribute
float_initial_value: (**read**) **float**;
end float_attribute_type;

(22) boolean_attribute_type: (**protected**) **child type of** attribute_type **with**
attribute
boolean_initial_value: (**read**) **boolean**;
end boolean_attribute_type;

(23) time_attribute_type: (**protected**) **child type of** attribute_type **with**
attribute
time_initial_value: (**read**) **time**;
end time_attribute_type;

(24) 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;

(25) 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;

(26) 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_INTEGER_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;

```
(27)    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;

(28)    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;

(29)    enumeral_type_in_sds: (protected) child type of type_in_sds with
        attribute
            image: (read) string;
        end enumeral_type_in_sds;

(30)    extend object type process with
        link
            sds_in_working_schema: (navigate) designation link (number) to sds;
        end process;

(31)    end metasds;
```

(32) The object-orientation module requires the following extensions to the metasds SDS:

```
(33)    sds metasds:

(34)    import object type method_selection;

(35)    extend object type object_type with
        link
            obj_used_in_map: (navigate) implicit link (system_key) to method_selection reverse
                uses_object;
        end object_type;

(36)    interface_type: child type of type with
        link
            parent_interface: (navigate) reference link (number) to interface_type reverse
                child_interface;
            child_interface: (navigate) implicit link (system_key) to interface_type reverse
                parent_interface;
            has_operation: (navigate) reference link (uuid: string) to operation_type reverse
                used_in_interface;
        end interface_type;
```

(37)
```
operation_type: child type of type with
attribute
    operation_kind: (read) enumeration (NORMAL_CALL, ONEWAY_CALL) := NORMAL_CALL;
link
    used_in_interface: (navigate) implicit link (system_key) to interface_type reverse
        has_operation;
    has_parameter: (navigate) reference link (position: natural; name) to parameter_type
        reverse parameter_of with
    attribute
        parameter_mode: (read) enumeration (IN, OUT, INOUT) := IN;
    end has_parameter;
    has_return_value: (navigate) reference link to parameter_type reverse return_value_of;
    op_used_in_map: (navigate) implicit link (system_key) to method_selection reverse
        uses_operation;
end operation_type;
```

(38)
```
parameter_type: child type of type with
link
    parameter_of: (navigate) implicit link (system_key) to operation_type reverse
        has_parameter;
    return_value_of: (navigate) implicit link (system_key) to operation_type reverse
        has_return_value;
end parameter_type;
```

(39)
```
data_parameter_type: child type of parameter_type with
link
    constrained_to_attribute_type: (navigate) reference link to attribute_type;
end data_parameter_type;
```

(40)
```
interface_parameter_type: child type of parameter_type with
link
    constrained_to_interface_type: (navigate) reference link to interface_type;
end interface_parameter_type;
```

(41)
```
object_parameter_type: child type of parameter_type with
link
    constrained_to_object_type: (navigate) reference link to object_type;
end object_parameter_type;
```

(42)
```
extend object type object_type_in_sds with
link
    supports_interface: (navigate) reference link (name) to interface_type_in_sds reverse
        applies_to;
end object_type_in_sds;
```

(43)
```
interface_type_in_sds: child type of type_in_sds with
link
    applies_to: (navigate) implicit link (type_identifier) to object_type_in_sds reverse
        supports_interface;
    in_operation_set: (navigate) reference link (number; name) to operation_type_in_sds
        reverse is_operation_of;
end interface_type_in_sds;
```

(44)
```
operation_type_in_sds: child type of type_in_sds with
link
    is_operation_of: (navigate) implicit link (system_key) to interface_type reverse
        in_operation_set;
end operation_type_in_sds;
```

```
(45)    extend object type method_selection with
        link
            uses_operation: (navigate) reference link (number) to operation_type reverse
                op_used_in_map;
            uses_object: (navigate) reference link (number) to object_type reverse obj_used_in_map;
        end method_selection;

(46)    end metasds;
```

E.3 The discretionary security SDS

```
(1)     sds discretionary_security:

(2)     import object type system-object, system-static_context, system-process, system-workstation,
            system-common_root;

(3)     import attribute type system-name, system-number;

(4)     security_group_directory: child type of object with
        link
            known_security_group: (navigate) existence link (group_identifier: natural) to
                security_group;
            security_groups_of: implicit link to common_root reverse security_groups;
        end security_group_directory;

(5)     security_group: child type of object;

(6)     user: child type of security_group with
        link
            user_identity_of: (navigate) non_duplicated designation link (number) to process;
            user_member_of: (navigate) reference link (number) to user_group reverse has_users;
        end user;

(7)     user_group: child type of security_group with
        link
            has_users: (navigate) reference link (number) to user reverse user_member_of;
            user_subgroup_of: (navigate) reference link (number) to user_group reverse
                has_user_subgroups;
            has_user_subgroups: (navigate) reference link (number) to user_group reverse
                user_subgroup_of;
            adopted_user_group_of: (navigate) non_duplicated designation link (number) to process;
        end user_group;

(8)     program_group: child type of security_group with
        link
            has_programs: (navigate) reference link (number) to static_context reverse
                program_member_of;
            program_subgroup_of: (navigate) reference link (number) to program_group reverse
                has_program_subgroups;
            has_program_subgroups: (navigate) reference link (number) to program_group reverse
                program_subgroup_of;
        end program_group;

(9)     extend object type static_context with
        link
            program_member_of: (navigate) implicit link (system_key) to program_group reverse
                has_programs;
        end static_context;
```

```
(10)    extend object type process with
        attribute
            default_atomic_acl: (protected) string;
            default_object_owner: (protected) natural;
        link
            user_identity: (navigate) designation link to user;
            adopted_user_group: (navigate) designation link to user_group;
            adoptable_user_group: (navigate) designation link (number) to user_group with
            attribute
                adoptable_for_child: (read) boolean:= true;
            end adoptable_user_group;
        end process;

(11)    extend object type object with
        attribute
            atomic_acl: (protected) non_duplicated string;
            composite_acl: (protected) non_duplicated string;
        end object;

(12)    extend object type common_root with
        link
            security_groups: (navigate) existence link to security_group_directory reverse
                security_groups_of;
        end common_root;

(13)    audit_file: child type of object with
        contents audit_file;
        link
            audit_of: reference link (number) to workstation reverse audit;
        end audit_file;

(14)    extend object type workstation with
        link
            audit: (navigate) existence link to audit_file reverse audit_of;
        end workstation;

(15)    end discretionary_security;
```

E.4 The mandatory security SDS

```
(1)     sds mandatory_security;

(2)     import object type system-object, system-volume, system-device, system-common_root,
            system-execution_site, system-process;

(3)     import attribute type system-name, system-number;

(4)     import object type discretionary_security-security_group, discretionary_security-user;

(5)     import attribute type discretionary_security-group_identifier;

(6)     extend object type security_group with
        link
            may_downgrade: (navigate) reference link (name) to confidentiality_class reverse
                downgradable_by;
            may_upgrade: (navigate) reference link (name) to integrity_class reverse upgradable_by;
        end security_group;
```

(7)
```
extend object type user with
link
    cleared_for: (navigate) reference link (name) to mandatory_class reverse having_clearance;
end user;
```

(8)
```
mandatory_directory: child type of object with
link
    known_mandatory_class: (navigate) existence link (name) to mandatory_class;
    mandatory_classes_of: implicit link to common_root reverse mandatory_classes;
end mandatory_directory;
```

(9)
```
mandatory_class: child type of object with
link
    having_clearance: (navigate) reference link (group_identifier) to user reverse cleared_for;
end mandatory_class;
```

(10)
```
confidentiality_class: child type of mandatory_class with
link
    dominates_in_confidentiality: (navigate) reference link to confidentiality_class reverse
        confidentiality_dominator;
    confidentiality_dominator: (navigate) reference link to confidentiality_class reverse
        dominates_in_confidentiality;
    downgradable_by: (navigate) reference link (group_identifier) to security_group reverse
        may_downgrade;
end confidentiality_class;
```

(11)
```
integrity_class: child type of mandatory_class with
link
    dominates_in_integrity: (navigate) reference link to integrity_class reverse
        integrity_dominator;
    integrity_dominator: (navigate) reference link to integrity_class reverse
        dominates_in_integrity;
    upgradable_by: (navigate) reference link (group_identifier) to security_group reverse
        may_upgrade;
end integrity_class;
```

(12)
```
extend object type object with
attribute
    confidentiality_label: (read) string;
    integrity_label: (read) string;
end object;
```

(13)
```
extend object type common_root with
link
    mandatory_classes: (navigate) existence link to mandatory_directory reverse
        mandatory_classes_of;
end common_root;
```

(14)
```
extend object type volume with
attribute
    confidentiality_high_label: (read) non_duplicated string;
    confidentiality_low_label: (read) non_duplicated string;
    integrity_high_label: (read) non_duplicated string;
    integrity_low_label: (read) non_duplicated string;
end volume;
```

```
(15)  extend object type device with
      attribute
          confidentiality_high_label;
          confidentiality_low_label;
          integrity_high_label;
          integrity_low_label;
          contents_confidentiality_label: (read) non_duplicated string;
          contents_integrity_label: (read) non_duplicated string;
      end device;

(16)  extend object type execution_site with
      attribute
          confidentiality_high_label;
          confidentiality_low_label;
          integrity_high_label;
          integrity_low_label;
      end execution_site;

(17)  floating_level: NO_FLOAT, FLOAT_IN, FLOAT_OUT, FLOAT_IN_OUT;

(18)  extend object type process with
      attribute
          floating_confidentiality_level: (read) non_duplicated enumeration (floating_level) :=
              NO_FLOAT;
          floating_integrity_level: (read) non_duplicated enumeration (floating_level) := NO_FLOAT;
      end process;

(19)  end mandatory_security;
```

E.5 The accounting SDS

```
(1)   sds accounting:

(2)   import object type system-object, system-process, system-common_root, system-workstation;

(3)   import attribute type system-number;

(4)   accounting_directory: child type of object with
      link
          known_consumer_group: (navigate) existence link (consumer_identifier: natural) to
              consumer_group;
          known_resource_group: (navigate) existence link (resource_identifier: natural) to
              resource_group;
          accounts_of: implicit link to common_root reverse accounts;
      end accounting_directory;

(5)   consumer_group: child type of object with
      link
          consumer_process: (navigate) non_duplicated designation link (number) to process;
      end consumer_group;

(6)   resource_group: child type of object with
      link
          resource_group_of: (navigate) reference link (number) to object reverse in_resource_group;
      end resource_group;

(7)   extend object type process with
      link
          consumer_identity: (navigate) designation link to consumer_group;
      end process;
```

```
(8)   extend object type object with
      link
          in_resource_group: (navigate) reference link to resource_group reverse resource_group_of;
      end object;

(9)   extend object type common_root with
      link
          accounts: (navigate) existence link to accounting_directory reverse accounts_of;
      end common_root;

(10)  extend object type workstation with
      link
          has_log: (navigate) reference link to accounting_log reverse is_log_for;
      end workstation;

(11)  accounting_log: child type of object with
      contents accounting_log;
      link
          is_log_for: (navigate) reference link (number) to workstation reverse has_log;
      end accounting_log;

(12)  end accounting;
```

Annex F
(normative)

The fine-grain objects module

F.1 Extensions to object management (see clause 9)

F.1.1 Additional object management concepts (see 9.1)

(1) **sds** system:

(2)
```
extend object with
attribute
    cluster_identifier: (read) non_duplicated natural;
end object;
```

(3) **end** system;

(4) The cluster identifier identifies the cluster in which the object resides. If the cluster identifier is 0, the object does not reside in a cluster. If the cluster identifier is not 0, it is the key of a "known_cluster" link from the volume on which the object resides to the cluster in which the object resides. See 11.1.

(5) An object which resides in a cluster is called a *fine-grain* object. An object which does not reside in a cluster is called a *coarse-grain* object. The same object can be created as coarse-grain object and become fine-grain after it is moved into a cluster and conversely.

(6) Objects which have the following types (or one of their descendant types) cannot reside in a cluster. They are always coarse-grain objects:

(7) - "file", "pipe", "message_queue", "device", "accounting_log", "audit_file";

(8) - "volume", "cluster", "archive", "archive_directory";

(9) - "process", "activity";

(10) - "common_root";

(11) - "sds";

(12) - "workstation", "execution_class", "execution_site", "execution_site_directory";

(13) - "replica_set_directory", "replica_set";

(14) - "security_group", "program_group", "mandatory_directory", "mandatory_class", "security_group_directory";

(15) - "accounting_directory", "consumer_group", "resource_group".

(16) The last access time of a fine-grain object is equal to the default initial value of time attributes (see 8.3.2).

(17) The last modification time of a fine-grain object is equal to the last modification time of the cluster in which it resides.

(18) The last change time of a fine-grain object is equal to the last change time of the cluster in which it resides.

(19) The last modification time of a fine-grain object is set only:

(20) - when the object is created in a cluster (operations OBJECT_CREATE, OBJECT_COPY, VERSION_REVISE, VERSION_SNAPSHOT),

(21) - when the object is moved into a cluster (operation OBJECT_MOVE),

(22) - when the last modification time of the cluster in which it resides is modified.

(23) The last modification time of a cluster is the system time of the last release of a write or delete lock for an object residing in the cluster.

(24) The replicated state of a fine-grain object is equal to the replicated state of the cluster in which it resides.

(25) NOTE - Neither a fine-grain object nor a cluster has contents, so the last access time is meaningless for both. That is why it is always equal to the default initial value of time attributes.

F.1.2 Link operations affected by support of fine-grain objects (see 9.2)

F.1.2.1 LINK_CREATE

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

New semantics

(2) If *dest* is a fine-grain object and *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 all objects which reside in the cluster of *dest*; similarly if *origin* is a fine-grain object and *reverse_link* (the reverse link of *new_link*) is a composition link, then any security group that has OWNER granted or denied to *dest* has OWNER granted or denied respectively to all objects which reside in the cluster of *origin*. This requires the process to have OWNER rights on *dest* or *origin* respectively. See 19.1.2 for details.

F.1.2.2 LINK_DELETE

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

New semantics

(2) For each deleted fine-grain object *object*, the "object_in_cluster" link from the cluster in which *object* was residing to *object* is also deleted.

F.1.2.3 LINK_REPLACE

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

New semantics

(2) The semantics of this operation refers to LINK_DELETE. It is therefore affected in the same way.

F.1.3 Object operations affected by support of fine-grain objects (see 9.3)

F.1.3.1 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
```

New semantics

(2) If *on_same_volume_as* is supplied, then *new_object* and all its components reside in the same volume as *on_same_volume_as*.

(3) If *on_same_volume_as* is not supplied, then *new_object* resides in the same volume as *object*, and each component of *new_object* resides in the same volume as its corresponding component in *object*.

(4) Additionally:

(5) - If *on_same_volume_as* is supplied and if the cluster identifier of *on_same_volume_as* is not 0, then *new_object* and all its components reside in the same cluster as *on_same_volume_as*.

(6) - If *on_same_volume_as* is supplied is itself a cluster, then *new_object* and all its components reside in the cluster *on_same_volume_as*.

(7) - If *on_same_volume_as* is not supplied and if the cluster identifier of *object* is not 0, then *new_object* resides in the same cluster as *object*. Similarly, if the cluster identifier of a component of *object* is not 0, the corresponding component of *new_object* is created in the same cluster as that component of *object*, and so on for subcomponents.

(8) If *new_object* or any of its components is created in a cluster, then an "object_in_cluster" link is created from that cluster to the new object or component. Each created link is keyed by the exact identifier of the created object.

(9) For each object X created in a cluster C:

(10) - The atomic ACL of X is set to the atomic ACL of C.

(11) - The security labels of X are set to the security labels of C.

(12) - The last modification time and last change time of X are set to the last modification time and last change time of C, respectively.

New errors

(13) OBJECT_CANNOT_BE_CLUSTERED (*object* or its components)

(14) If *object* is a fine-grain object:
ACCESS_ERRORS (cluster of *object*, ATOMIC, MODIFY, APPEND_LINKS)

F.1.3.2 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
```

New semantics

(2) If *on_same_volume_as* is supplied, then *new_object* resides in the same volume as *on_same_volume_as*.

(3) If *on_same_volume_as* is not supplied, then *new_object* resides in the same volume as *new_origin*.

(4) Additionally:

(5) - If *on_same_volume_as* is supplied and if the cluster identifier of *on_same_volume_as* is not 0, then *new_object* resides in the same cluster as *on_same_volume_as*.

(6) - If *on_same_volume_as* is itself a cluster, then *new_object* resides in the cluster *on_same_volume_as*.

(7) - If *on_same_volume_as* is not supplied and if the "cluster_identifier" of *new_origin* is not 0, then *new_object* resides in the same cluster as *new_origin*.

(8) If *new_object* is created in a cluster, then:

(9) - An "object_in_cluster" link is created from this cluster to the new object. The created link is keyed by the exact identifier of the created object.

(10) - The atomic ACL of *new_object* is set to the atomic ACL of the cluster.

(11) - The security labels of *new_object* are set to the security labels of the cluster.

(12) - The last modification time and last change time of *new_object* are set to the last modification time and last change time of the cluster, respectively.

New errors

(13) OBJECT_CANNOT_BE_CLUSTERED (object to be created)

(14) If *object* is a fine-grain object:
ACCESS_ERRORS (cluster of *object*, ATOMIC, MODIFY, APPEND_LINKS)

F.1.3.3 OBJECT_DELETE

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

New semantics

(2) For each deleted fine-grain object, if any, the "object_in_cluster" link from the cluster in which the deleted object resided to the deleted object is also deleted.

New errors

(3) If any deleted object is a fine-grain object:
- ACCESS_ERRORS (cluster of deleted object, ATOMIC, MODIFY, WRITE_LINKS)

F.1.3.4 OBJECT_MOVE

(1)
```
OBJECT_MOVE (
    object                : Object_designator,
    on_same_volume_as     : Object_designator,
    scope                 : Object_scope
)
```

New semantics

(2) *object* and all its components are moved to the same volume as *on_same_volume_as*.

(3) Additionally, if the cluster identifier of *on_same_volume_as* is not 0, or *on_same_volume_as* is itself a cluster, then:

(4) - *object* and all its components are moved into the cluster of *on_same_volume_as*, or the cluster *on_same_volume_as*.

(5) - An "object_in_cluster" link is created from that cluster to the moved object and to each of its components and subcomponents. Each created link is keyed by the exact identifier of the moved object.

(6) - The atomic ACLs of *object* and all its components are set to the atomic ACL of the cluster.

(7) - The security labels of *object* are set to the security labels of the cluster.

(8) - The last modification time and last change time of *object* and all its components are set to the last modification time and last change time of the cluster, respectively.

(9) For *object* (if moved) and each moved component, which was previously residing in a cluster, the "object_in_cluster" link from the cluster to the object is deleted.

New errors

(10) OBJECT_CANNOT_BE_CLUSTERED (*object*)

(11) If *object* is a fine-grain object:
ACCESS_ERRORS (cluster of *object*, ATOMIC, MODIFY, WRITE_LINKS)

(12) If *on_same_volume_as* resides in or is a cluster *cluster*:
ACCESS_ERRORS (*cluster*, ATOMIC, MODIFY, APPEND_LINKS)

F.1.3.5 OBJECT_SET_TIME_ATTRIBUTES

(1)
```
OBJECT_SET_TIME_ATTRIBUTES(
    object              : Object_designator,
    last_access         : [ Time ],
    last_modification   : [ Time ],
    scope               : Object_scope
)
```

New semantics

(2) If *object* is a cluster, the new time attributes are also set on all objects residing in the cluster.

New errors

(3) If *object* is a fine-grain object:
OBJECT_IS_FINE_GRAIN (*object*)

F.1.4 Version operations affected by support of fine-grain objects (see 9.4)

F.1.4.1 VERSION_IS_CHANGED

(1)
```
VERSION_IS_CHANGED (
    version         : Object_designator,
    predecessor     : Natural
)
    changed         : Boolean
```

New errors

(2) If *object* is a fine-grain object:
OBJECT_IS_FINE_GRAIN (*object*)

F.1.4.2 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
```

New semantics

(2) The semantics of VERSION_REVISE refers to the semantics of OBJECT_COPY. It is therefore indirectly changed in the same way.

F.1.4.3 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
```

New semantics

(2) The semantics of VERSION_SNAPSHOT refers to the semantics of OBJECT_COPY. It is therefore indirectly changed in the same way.

F.2 Volumes, clusters, devices, and archives (see clause 11)

F.2.1 Cluster concepts (see 11.1)

(1)
```
Cluster_identifier = Natural
```

(2)
```
sds system:
```

(3)
```
extend object type volume with
link
    known_cluster: (navigate) non_duplicated existence link (cluster_identifier) to cluster
        reverse cluster_in_volume;
end volume;
```

(4)
```
cluster: child type of object with
attribute
    cluster_characteristics: (read) string;
link
    object_in_cluster: (navigate) non_duplicated designation link (exact_identifier) to object;
    cluster_in_volume: (navigate) implicit link to volume reverse known_cluster;
end cluster;
```

(5)
```
end system;
```

(6) A cluster is an object which groups a set of objects sharing some common properties or behaviour in respect with concurrency control, time attributes, auditing, security, notification and accounting. See 9.1.

(7) The destinations of the "known_cluster" links from a volume are the clusters residing on that volume.

(8) The destinations of the "object_in_cluster" links from a cluster are called the objects *residing in* that cluster. The value of the "exact_identifier" attribute is the exact identifier of the object (see 9.1.1).

(9) All objects which reside in a cluster must also reside on the same volume as the volume of the cluster itself.

(10) The "cluster_characteristics" attribute is an implementation-defined string specifying implementation-dependent characteristics of the cluster.

F.2.2 Archive operations affected by support of fine-grain objects (see 11.1.4)

F.2.2.1 ARCHIVE_RESTORE

(1)
```
ARCHIVE_RESTORE (
    device                : Device_designator,
    archive               : Archive_designator,
    scope                 : Archive_selection,
    on_same_volume_as     : Object_designator
)
    restoring_status      : Archive_status
```

New semantics

(2) Additionally, if the cluster identifier of *on_same_volume_as* is not 0, or if *on_same_volume_as* is itself a cluster, then for each restored object:

(3) - The object is allocated in the cluster of *on_same_volume_as*, or the cluster *on_same_volume_as*.

(4) - An "object_in_cluster" link is created from that cluster to the restored object. Each created link is keyed by the exact identifier of the restored object.

(5) - The atomic ACLs of *object* and all its components are set to the atomic ACL of the cluster.

(6) - The last access time, last modification time, and last change time of *object* and all its components are set to the last access time, last modification time, and last change time of the cluster, respectively.

New errors

(7) OBJECT_CANNOT_BE_CLUSTERED (any object being restored)

(8) If *on_same_volume_as* resides in or is a cluster *cluster*:
ACCESS_ERRORS (*cluster*, ATOMIC, MODIFY, APPEND_LINKS)

F.2.2.2 ARCHIVE_SAVE

(1)
```
ARCHIVE_SAVE (
    device          : Device_designator,
    archive         : Archive_designator,
    objects         : Object_designators
)
    archiving_status : Archive_status
```

New semantics

(2) For each object to be archived which resides in a cluster, the "object_in_cluster" link from the cluster on which the object resides to the object is deleted.

New errors

(3) For any object X of *objects* which resides in a cluster
ACCESS_ERRORS (cluster of X, ATOMIC, MODIFY, APPEND_LINKS)

F.2.3 New operations on clusters (see 11.2)

F.2.3.1 CLUSTER_CREATE

(1)
```
CLUSTER_CREATE (
    on_same_volume_as   : Object_designator,
    cluster_identifier  : Natural,
    access_mask         : Atomic_access_rights,
    cluster_characteristics : String,
)
    new_cluster         : Cluster_designator
```

(2) CLUSTER_CREATE creates a new cluster *new_cluster* in the volume *volume* in which the object *on_same_volume_as* resides.

(3) A new "known_cluster" link with key *cluster_identifier* is created from *volume* to *new_cluster*.

(4) *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 confidentiality and integrity labels of *cluster* are respectively set to the confidentiality and integrity labels of the mandatory context of the calling process.

(6) The "cluster_characteristics" attribute of *new_cluster* is set to *cluster_characteristics*.

(7) Write locks of the default mode are obtained on *on_same_volume_as*, on *new_cluster*, and on the new "known_cluster" link.

Errors

(8) ACCESS_ERRORS (*volume*, ATOMIC, MODIFY, APPEND_LINKS)

(9) LIMIT_WOULD_BE_EXCEEDED (MAX_KEY_VALUE)

(10) OBJECT_OWNER_VALUE_WOULD_BE_INCONSISTENT_WITH_ATOMIC_ACL

(11) REFERENCE_CANNOT_BE_ALLOCATED

(12) CLUSTER_EXISTS (*cluster_identifier*, *volume*)

F.2.3.2 CLUSTER_DELETE

(1)
```
CLUSTER_DELETE (
    cluster : Cluster_designator
)
```

(2) CLUSTER_DELETE deletes the "known_cluster" link to *cluster* from the volume *volume* on which *cluster* is residing. and then deletes *cluster*.

(3) Write locks (of the default kind) are obtained on *cluster* and the deleted cluster and the deleted link.

Errors

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

(5) ACCESS_ERRORS (*cluster*, ATOMIC, CHANGE, WRITE_IMPLICIT)

(6) If the conditions hold for deletion of the "cluster" object *cluster*:
ACCESS_ERRORS (*volume*, ATOMIC, MODIFY, DELETE)

(7) CLUSTER_HAS_OTHER_LINKS (*cluster*)

(8) CLUSTER_IS_UNKNOWN (*cluster*)

F.2.3.3 CLUSTER_LIST_OBJECTS

(1)
```
CLUSTER_LIST_OBJECTS (
    cluster     : Cluster_designator,
    types       : Object_type_nominators
)
    objects     : Object_designators
```

(2) CLUSTER_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 in *cluster*, whose type in working schema is an element of *types*.

(4) A read lock of the default mode is obtained on *cluster*.

Errors

(5) ACCESS_ERRORS (*cluster*, ATOMIC, READ, READ_LINKS)

(6) REFERENCE_CANNOT_BE_ALLOCATED

F.3 Notification (see clause 15)

F.3.1 Notification concepts (see 15.1)

(1) Notifiers cannot be associated with fine-grain objects.

F.3.2 Notification operations affected by support of fine-grain objects (see 15.2)

F.3.2.1 NOTIFY_CREATE

(1)
```
NOTIFY_CREATE (
    notifier_key    : Natural,
    queue           : Message_queue_designator,
    object          : Object_designator
)
```

New errors

(2) OBJECT_IS_FINE_GRAIN (*object*)

F.4 Concurrency and integrity control (see clause 16)

(3) - The requested external lock mode is compatible with the external lock mode of other locks obtained by concurrent activities.

(4) - The requested internal lock mode is compatible with the external lock mode of the child activities.

(5) - The requested external lock mode is compatible with the internal lock mode of the parent activity (if any).

(6) - If a read lock is already acquired by at least one different process running in the same activity and the current process is performing an operation which requests a write lock, the lock acquisition (such a request is a promotion from read to write for a coarse-grain object) is delayed until the lock can be promoted to write and until all other processes which have made a lock request, are terminated.

(7) - If a write lock is already acquired by one (and only one) different process running in the same activity and the current process is performing an operation which requests a read or a write lock acquisition on a fine-grain object (such a request is necessarily satisfied in case of a lock on a coarse-grain object) , the lock acquisition is delayed until the process which made the write lock request on the cluster, is terminated.

NOTES

(8) 1. It is intended that an implementation supports caching of fine-grain objects by loading in main memory all the objects of a cluster. For performance reasons, it is intended that the loading of all the objects of a cluster is done in the private user space of processes which need to access the objects.

(9) 2. The additional locking rules prevent two different processes running in the same activity from accessing the same fine-grain objects and from performing concurrent non-synchronized updates on their caches. With these additional locking rules, the loading of the cache is intended to happen as follows:

(10) - when an activity acquires a read lock on a cluster, all objects of the cluster are placed in a read-only cache stored in the space of the process which is performing the operation causing the lock acquisition.

(11) - When an activity acquires a write lock on a cluster, all objects of the cluster are placed in a read-write cache stored in the space of the process which is performing the operation causing the lock acquisition.

(12) - Several processes can have read-only caches on the same cluster.

(13) - Only one process can have a read-write cache on a cluster, at a given time.

(14) 3. It is intended that a process unloads a cache when the activity causing the cache to be loaded is ended or aborted. If the activity commits, the cache has to be downloaded to the object base. If the activity is aborted, the cache must be simply discarded, without updates in the object base.

(15) 4. Whenever this ECMA Standard or this amendment says 'a read/write lock of the default mode is obtained on an object *object*', if *object* is a fine-grain object this is implicitly equivalent to 'a read/write lock of the default mode is obtained on the cluster in which *object* resides' as a consequence of the first rule above.

F.5 Replication (see clause 17)

F.5.1 Replication concepts (see 17.1)

(1) When a cluster is duplicated, all the fine-grain objects residing in the cluster are replicated.

(2) A fine-grain object cannot be replicated in isolation (i.e. the only way to duplicate it is by duplicating its cluster).

F.5.2 Replication operations affected by support of fine-grain objects (see 17.2)

F.5.2.1 REPLICATED_OBJECT_CREATE

(1)
```
REPLICATED_OBJECT_CREATE (
    replica_set     : Replica_set_designator,
    object          : Object_designator
)
```

New semantics

(2) If *object* is a cluster, its replicated state is set to MASTER.

New errors

(3) OBJECT_IS_FINE_GRAIN (*object*)

F.5.2.2 REPLICATED_OBJECT_DUPLICATE

(1)
```
REPLICATED_OBJECT_DUPLICATE (
    object          : Object_designator,
    volume          : Administration_volume_designator,
    copy_volume     : Administration_volume_designator
)
```

New semantics

(2) If *object* is a cluster, then all the objects which reside in the cluster are replicated

F.5.2.3 REPLICATED_OBJECT_REMOVE

(1)
```
REPLICATED_OBJECT_REMOVE (
    object  : Object_designator
)
```

New semantics

(2) If *object* is a cluster, its replicated state is set to NORMAL.

New errors

(3) OBJECT_IS_FINE_GRAIN (*object*)

F.6 Discretionary security (see clause 19)

F.6.1 Concepts of discretionary security (see 19.1)

(1) All fine-grain objects residing in a cluster have the same ACLs as the cluster.

F.6.2 Discretionary access control operations affected by support of fine-grain objects (see 19.2)

F.6.2.1 OBJECT_SET_ACL_ENTRY

(1)
```
OBJECT_SET_ACL_ENTRY (
    object      : Object_designator,
    group       : Group_identifier,
    modes       : Atomic_access_rights,
    scope       : Object_scope
)
```

New semantics

(2) If *object* is a cluster, then the same ACL entry is added to the ACL of all objects residing in the cluster.

New errors

(3) OBJECT_IS_FINE_GRAIN (*object*)

F.7 Mandatory security (see clause 20)

F.7.1 Mandatory security concepts (see 20.1)

(1) All fine-grain objects residing in a cluster have the same confidentiality and integrity labels as the cluster.

F.7.2 Mandatory security operations affected by support of fine-grain objects (see 20.2)

F.7.2.1 OBJECT_SET_CONFIDENTIALITY_LABEL

(1)
```
OBJECT_SET_CONFIDENTIALITY_LABEL (
    object  : Object_designator,
    label   : Security_label
)
```

New semantics

(2) If *object* is a cluster, then the same confidentiality label is set on all objects residing in the cluster.

New errors

(3) OBJECT_IS_FINE_GRAIN (*object*)

F.7.2.2 OBJECT_SET_INTEGRITY_LABEL

(1)
```
OBJECT_SET_INTEGRITY_LABEL (
    object  : Object_designator,
    label   : Security_label
)
```

New semantics

(2) If *object* is a cluster, then the same confidentiality label is set on all objects residing in the cluster.

New errors

(3) OBJECT_IS_FINE_GRAIN (*object*)

F.8 Auditing (see clause 21)

(1) Operations on fine-grain objects do not produce auditable events.

Annex G
(normative)

The object-orientation module

(1) This annex defines the object-orientation module in terms of some additions to the predefined SDSs metasds and system, additional error conditions, some new operations, and additional semantics to one operation.

G.1 Extensions to the foundation (see clause 8)

G.1.1 Additional classes of types (see 8.3)

G.1.1.1 New type classes

(1) Type = Object_type | Attribute_type | Link_type | Enumeral_type | Interface_type | Operation_type | Parameter_type

(2) Type_nominator = Object_type_nominator | Attribute_type_nominator | Link_type_nominator | Enumeral_type_nominator | Interface_type_nominator | Operation_type_nominator | Parameter_type_nominator

(3) Interface_type_nominator :: Token

(4) Interface_type_nominators = **set of** Interface_type_nominator

(5) Interface_scope = NO_OPERATION | ALL_OPERATION

(6) Operation_type_nominator :: Token

(7) Operation_type_nominators = **set of** Operation_type_nominator

(8) Parameter_type_nominator :: Token

(9) Data_parameter_type_nominator = Parameter_type_nominator

(10) Operation_parameter_type_nominator = Parameter_type_nominator

(11) Interface_parameter_type_nominator = Parameter_type_nominator

(12) Parameter_type_nominators = **seq of** Parameter_type_nominator

(13) The datatypes Type and Type_nominator are extended to include the new datatypes Interface_type, Operation_type, and Parameter_type, and their corresponding type nominator datatypes, respectively. Data parameter type nominators, operation parameter type nominators, and interface parameter type nominators denote data parameter types, operation parameter types, and interface parameter types respectively (see G.3.1).

G.1.1.2 Interface types

(1)
```
Interface_type ::
    TYPE_NOMINATOR      : Interface_type_nominator
    OPERATION_TYPES     : Operation_type_nominators
    PARENT_INTERFACES   : Interface_type_nominators
    CHILD_INTERFACES    : Interface_type_nominators
represented by interface_type
```

(2) The parent interfaces define the inheritance rules governing the ability of an object type to support a given interface. The operations supported by an object type supporting an interface are

the operations of that interface and of all its ancestor interfaces, where the *ancestor interfaces* of an interface are the parent interfaces of that interface, their parent interfaces, and so on, excluding the interface itself.

(3) The child interfaces are the interfaces which have that interface as parent interface. The child interfaces of an interface, their child interfaces, and so on, excluding the interface itself, are called the *descendant interfaces* of that interface.

(4) The parent-interface/child-interface relation between interfaces forms a directed acyclic graph.

(5) The 'operation types' operation types are the operations that can be invoked on an object of an object type supporting that interface.

G.1.1.3 Operation types

(1)
```
Operation_type ::
    TYPE_NOMINATOR      : Operation_type_nominator
    USED_IN_INTERFACE   : Interface_type_nominators
    PARAMETERS          : seq of (Parameter_type_nominator * Parameter_mode)
    KIND                : Operation_kind
    RETURN_VALUE        : Parameter_type_nominator
represented by operation_type
```

(2)
```
Parameter_mode = IN | OUT | INOUT
```

(3)
```
Operation_kind = NORMAL_CALL | ONEWAY_CALL
```

(4) The 'used in interface' interface types are the interface types for which this operation type is among the operation types.

(5) The sequence 'parameters' is the sequence of parameter types and modes of parameters that are passed during an invocation. The parameter mode specifies whether the parameter value is passed from the caller to the operation only (IN), from the operation to the caller only (OUT), or both ways (INOUT).

(6) The kind is NORMAL_CALL if an operation of this type is expected to return a value after the execution of the method associated with the operation completes, and ONEWAY_CALL otherwise.

(7) The return value is the parameter type of an extra out parameter that is returned by an invocation.

G.1.1.4 Parameter types

(1)
```
Parameter_type ::
    TYPE_NOMINATOR              : Parameter_type_nominator
    PARAMETER_TYPE_IDENTIFIER   : Attribute_type_nominator | Interface_type_nominator |
        Object_type_nominator
    represented by parameter_type
```

(2) The parameter type identifier constrains the datatype of parameters of the parameter type to be the value type of the attribute type, an object type supporting the interface type, or the object type, respectively.

G.1.2 Types in SDS (see 8.4)

G.1.2.1 New type in SDS classes

(1) Type_in_sds = Object_type_in_sds | Attribute_type_in_sds | Link_type_in_sds | Enumeral_type_in_sds | Interface_type_in_sds | Operation_type_in_sds | Parameter_type_in_sds

(2) Type_nominator_in_sds = Object_type_nominator_in_sds | Attribute_type_nominator_in_sds | Link_type_nominator_in_sds | Enumeral_type_nominator_in_sds | Interface_type_nominator_in_sds | Operation_type_nominator_in_sds | Parameter_type_nominator_in_sds

(3) Interface_type_nominator_in_sds :: Token

(4) Interface_type_nominators_in_sds = **set of** Interface_type_nominator_in_sds

(5) Operation_type_nominator_in_sds :: Token

(6) Operation_type_nominators_in_sds = **set of** Operation_type_nominator_in_sds

G.1.2.2 Interface types in SDS

(1) Interface_type_in_sds :: Type_in_sds_common_part &&
    APPLIED_OBJECT_TYPES : Object_type_nominators_in_sds
    APPLIED_OPERATIONS : Operation_type_nominators_in_sds
represented by interface_type_in_sds

(2) The *applied object types* are the object types that support the interface, i.e. of which instances can be used as the controlling objects of an invocations.

(3) The *applied operations* are the operations of the associated interface type that are visible.

G.1.2.3 Operation types in SDS

(1) Operation_type_in_sds :: Type_in_sds_common_part

G.1.3 Method invocation

(1) Parameter_item :: Attribute_value | Object_designator

(2) Parameter_items = **seq of** Parameter_item

(3) Method_request ::
    TARGET_OBJECT : Object_designator
    OPERATION_ID : Operation_type_nominator
    PARAMETERS : Parameter_items
    CONTEXT : Object_designator

(4) Method_requests = **seq of** Method_request

(5) Context_adoption = ADOPT_WORKING_SCHEMA | ADOPT_ACTIVITY | ADOPT_USER | ADOPT_OPEN_OBJECTS | ADOPT_REFERENCE_OBJECTS | ADOPT_ALL

(6) Context_adoptions = **set of** Context_adoption

(7) Method_request_id :: Token

(8) Method_request_ids = **seq of** Method_request_id

(9) Methods are invoked synchronously, but the synchronization may be deferred so that the requesting process does not wait immediately. Each invocation is described by a *method request*. A method request has the following properties:

(10) - a *target object* which is the controlling object of the request;

(11) - an *operation id(entifier)* that specifies the operation being invoked;

(12) - a sequence of *parameters* that specifies the sequence of parameter items required by the operation's definition;

(13) - a *context* that specifies where contextual information is held; the contextual information is implementation-defined and is used to determine the methods for the request or passed as required by the operation's definition.

(14) When an operation is performed, the corresponding request is assigned a set of context adoptions that specify which parts of the invoking process's current context may be adopted by the method for the request:

(15) - ADOPT_WORKING_SCHEMA: the method may adopt the current working schema of the invoking process.

(16) - ADOPT_ACTIVITY: the method may adopt the current activity of the invoking process.

(17) - ADOPT_USER: the method may adopt the security context of the invoking process.

(18) - ADOPT_OPEN_OBJECTS: the method has access to the same set of open objects as the invoking process.

(19) - ADOPT_REFERENCE_OBJECTS: the method has access to the same set of reference object as the invoking process.

(20) - ADOPT_ALL: the method has the same context as the invoking process.

(21) When an operation is performed, the corresponding request is assigned a method request id(entifier), which may be used to determine the completion status of the request.

(22) NOTE - Adopting a context is similar to starting a child process which adopts certain properties of the calling process.

G.2 Object-oriented invocation management

G.2.1 Invocation concepts

G.2.1.1 Datatypes for modules

(1) **sds** system:

(2) exec_class_name: **string**;

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

(4) exploits: (**navigate**) **designation link** (name) **to** sds;

(5)
```
tool: child type of object with
link
    external_component_of: (navigate) reference link (number) to tool reverse
        external_component;
    executable: (navigate) reference link (exec_class_name) to static_context reverse
        implementing_tool;
    exploits;
    has_map: (navigate) reference link (number) to method_selection reverse map_used_by;
component
    external_component: (navigate) composition link (number) to tool reverse
        external_component_of;
    internal_component: (navigate) composition link (number) to module reverse
        internal_component_of;
end tool;
```

(6)
```
module: child type of object with
link
    internal_component_of: (navigate) reference link (number) to tool reverse
        internal_component;
    exploits;
    linkable: (navigate) reference link (exec_class_name) to linkable_library reverse linkable_to;
end module;
```

(7)
```
linkable_library: child type of file with
link
    linkable_to: implicit link (system_key) to module reverse linkable;
end linkable_library;
```

(8)
```
end system;
```

(9) This part of the data model describes how tools and the methods they implement are represented and stored in the object base. This model is used to activate a specific method selected using the method mapping that connects the interfaces with the methods.

(10) An *exec[ution] class name* is a string uniquely identifying an execution class.

(11) An *operation id[entifier]* is used in the key of a "realized_by" link (see 9.1.2) to denote an operation.

(12) A *tool* is an executable program making use of the PCTE facilities. It is a composite object each of whose components is a tool or a module supporting part of the functionality of the tool.

(13) The destination of an "executable" link from a tool is an executable static context implementing the tool, keyed by the execution class name of the execution class of the workstations where the static context may be executed.

(14) The destinations of the "has_map" links from a tool constitute a set of method selections for use by the tool in resolving an invocation or a request to execute a method (see below).

(15) A *module* is a component of a tool that can be loaded and executed by the operating system.

(16) The destination of a "linkable" link from a module is a linkable library implementing the module, keyed by the execution class name of the execution class of the workstations where the linkable library may be loaded.

(17) The destinations of the "exploits" links from a tool or a module constitute a set of SDSs whose definitions were bound into the code of the tool module(s) at some time in order to interface with the object-oriented invocation management in a static rather than a dynamic way.

(18) A *linkable library* is a file containing information that can be linked by the operating system to produce a module.

G.2.1.2 Datatypes for method mapping

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

(2)
```
method_selection: child type of file with
link
    realized_by: (navigate) reference link (number; operation_id; type_identifier) to
        method_actions reverse realizes;
    map_used_by: (navigate) implicit link (system_key) to tool reverse has_map;
end method_selection;
```

(3)
```
method_actions: child type of file with
link
    implemented_by: (navigate) designation link (number) to tool, module;
    realizes: (navigate) implicit link (system_key) to method_selection reverse realized_by;
end method_actions;
```

(4)
```
dispatching_context: child type of file;
```

(5)
```
extend object type process with
link
    has_dispatching_context: (navigate) designation link to dispatching_context;
end process;
```

(6)
```
extend object type static_context with
link
    implementing_tool: (navigate) implicit link to tool reverse executable;
end static_context;
```

(7)
```
end system;
```

(8) This part of the data model describes how an operation is mapped into a specific method: this can depend on many factors, e.g. platform type, user context preferences, or user role.

(9) A "method_selection" object represents a ternary relationship that connects operations (the destinations of the "uses_operation" links), object types (the destinations of the "uses_object" links), and the method actions that realize the operation for that object type (the destinations of the "realized_by" links). (For "uses_operation" and "uses_object" see G.3.1.)

(10) The destination of the "realized_by" link is a "method_actions" object, which describes a set of methods to be activated in response to an operation request. The keys *operation_id* and *type_identifier* represent respectively the operation and the object type to which the method is connected, and an additional key *number* is used to select multiple realizations according to the method selection and the dispatching context. How the links from a method selection to a "method_actions" object are chosen is implementation-defined.

(11) The destinations of the "implemented_by" links from a "method_actions" object are tools and modules whose methods are to be activated, in an implementation-defined order.

(12) A *dispatching context* holds the information needed to resolve an operation mapping.

NOTES

(13) 1. The model is intended to provide a common basis to implement a generic mapping and is expected that each implementor may extend this model to support specific needs of its method of the object-orientation services. The data model should remain general enough to allow different styles of mapping.

(14) A dispatching context may resolve, among other things, the platform and the host where the invocation should be executed or the kind of tool class requested by the user (e.g. preferences over an editor).

(15) 2. Table 13 is an example of a possible method selection. The table is contained inside the "method_selection" object contents and is used to select the method according to the attributes specified by the user during the invocation or inside the invocation context.

(16) 3. The two first fields and the last correspond to the keys of the "realized_by" link.

Table 13 - Example of a method selection

Operation id	Type identifier	Attribute_1	Attribute_n	Number
123	555	"user"	"Platform_1"	1
345	666	-	"Platform_2"	2
789	777	"system"	"Platform_3"	3

G.2.2 Invocation operations

G.2.2.1 PROCESS_ADOPT_CONTEXT

(1)
```
PROCESS_ADOPT_CONTEXT (
    context_adoptions : Context_adoptions
)
```

(2) PROCESS_ADOPT_CONTEXT changes invoked parts of the current process's context to match those of the process whose request is being serviced. No part of the requesting process's context may be adopted unless permitted by the "context_adoptions" part of the request. When the method action which performed the context adoption returns, the changed parts of the current process's context return to their prior values.

(3) - If ADOPT_WORKING_SCHEMA is specified among the context adoptions, then the working schema of the current process is changed.

(4) - If ADOPT_ACTIVITY is specified among the context adoptions, then the activity of the process is changed.

(5) - If ADOPT_USER is specified among the context adoptions, then the current user of the current process is changed.

(6) - If ADOPT_OPEN_OBJECTS is specified, then the current opened objects of the current process are changed.

(7) - If ADOPT_REFERENCE_OBJECTS is specified, then the current object references of the current process are changed.

(8) - If ADOPT_ALL is specified, then the current context of the current process is set to that of the invoking process.

Errors

(9) For each SDS *sds* which is adopted by the current process from the new *context adoption*:
ACCESS_ERRORS (*sds*, ATOMIC, SYSTEM_ACCESS)

(10) For each open object *object* which is adopted by the current process from the new *context adoption*:
ACCESS_ERRORS (*object*, ATOMIC, SYSTEM_ACCESS)

(11) For activity *activity* which is adopted by the current process from the new *context adoption*:
ACCESS_ERRORS (*activity*, ATOMIC, SYSTEM_ACCESS)

(12) For user *user* which is adopted by the current process from the new *context adoption*:
ACCESS_ERRORS (*user*, ATOMIC, SYSTEM_ACCESS)

(13) For group *group* which is adopted by the current process from the new *context adoption*:
ACCESS_ERRORS (*group*, ATOMIC, SYSTEM_ACCESS)

G.2.2.2 REQUEST_INVOKE

(1)
```
REQUEST_INVOKE (
        request          : Method_request,
        context_adoptions : Context_adoptions,
)
        request_id       : Method_request_id
```

(2) REQUEST_INVOKE invokes the methods for the method request *request* and returns execution control when they have completed. It returns a unique method request identifier *request_id* for use in determining completion status.

(3) A "method_selection" object is selected which is the destination of a "has_map" link from the current tool. How the current tool and the key of the "has_map" link are determined is implementation-defined.

(4) The object type, operation identifier, and context of *request* are used to determine the key of a "realized_by" link from the method selection to a "method_actions" object.

(5) The "methods_actions" object determines one or more method actions to be performed and controls the order and resolution of the actions, and the manner in which the parameter lists for the actions are formed. The destination of the corresponding "implemented_by" link from the "methods_actions" object is either a module or a tool.

(6) If the method action is to take place in a module within the invoking process and the module is already loaded, then execution control is transferred directly to the method action.

(7) If the method action is to take place in a module within the invoking process which is not already loaded, the module is loaded and execution control is transferred directly to the method action.

(8) If the method action is to take place in a tool requiring another PCTE process (the *target tool*), then the request and context adoption information is delivered to the target tool.

(9) If the target tool is already executing, then the execution control is transferred directly to the method action in that process.

(10) If the tool is not already executing, then a new process is created immediately or at a later time. If the new process is to be started, it is started within the invoking process's space by way of a PROCESS_CREATE_AND_START. The new process inherits the invoking process's context, including working schema, activity, and user. When that process is started, the information is made available using the ACCEPT_REQUESTS operation. Then execution control is transferred directly to the method action in that process.

(11) If the tool is not already executing, a new process is created for it by way of some intermediate agent, immediately or at a later time. The intermediate agent transfers the request to the process. When that process is started, the information is made available using the ACCEPT_REQUESTS operation. Then execution control is transferred directly to the method action in that process.

NOTES

(12) 1. It is expected that the language bindings for this operation will yield the same language base code as is obtained for the corresponding operation defined in the Request Broker, with the context adoption information being passed within the request broker 'context'.

(13) 2. The actual object type is used for determination of the "method_actions" object from the method selection even if not visible in the invoking process's working schema.

(14) 3. The manner in which object type, operation type, and context information are combined to determine the 'realization key' may vary and may employ information stored in the "method_selection" object or its contents.

(15) 4. The method action may take place in a non-PCTE process, but the semantics is not specified by this Standard.

(16) 5. The formation of parameter lists and the manner of passing control to a method action in a process may vary and is analogous to the processing performed by the object-adapter skeleton introduced in the Common Object Request Broker. This process may employ information stored in the "method_actions" object or its contents.

(17) 6. The way in which the request and the context information is passed to the target object is implementation-defined.

Errors

(18) NUMBER_OF_PARAMETERS_IS_WRONG (*operation_id*)

(19) TYPE_OF_PARAMETER_IS_WRONG (*operation_id*, parameter item from parameters of *request*)

(20) OPERATION_METHOD_CANNOT_BE_FOUND (*operation_id*)

(21) OPERATION_METHOD_CANNOT_BE_ACTIVATED (*operation_id*)

G.2.2.3 REQUEST_SEND

(1)
```
PCTE_REQUEST_SEND (
        request            : Method_request,
        context_adoptions  : Context_adoptions,
)
        request_id         : Method_request_id
```

(2) REQUEST_SEND causes the methods for the request *request* to be executed as for REQUEST_INVOKE except that it may return execution control before they have begun.

Errors

(3) NUMBER_OF_PARAMETERS_IS_WRONG (*operation_id*)

(4) TYPE_OF_PARAMETER_IS_WRONG (*operation_id, parameter_item*)

(5) OPERATION_METHOD_CANNOT_BE_FOUND (*operation_id*)

(6) OPERATION_METHOD_CANNOT_BE_ACTIVATED (*operation_id*)

G.2.2.4 REQUEST_SEND_MULTIPLE

(1)
```
REQUEST_SEND_MULTIPLE (
        requests           : Method_requests,
        context_adoptions  : Context_adoptions,
)
        request_ids        : Method_request_ids
```

(2) REQUEST_SEND_MULTIPLE causes the methods for each request of *requests* to be executed as for REQUEST_SEND, employing the same context adoptions for all requests, and returning a unique request identifier in the corresponding position of *request_ids*.

Errors

(3) NUMBER_OF_PARAMETERS_IS_WRONG (*operation_id*)

(4) TYPE_OF_PARAMETER_IS_WRONG (*operation_id*, *parameter_item*)

(5) OPERATION_METHOD_CANNOT_BE_FOUND (*operation_id*)

(6) OPERATION_METHOD_CANNOT_BE_ACTIVATED (*operation_id*)

G.3 Object-oriented schema management

G.3.1 Datatypes for interface definition

(1)
```
sds metasds:
```

(2)
```
import object type method_selection;
```

(3)
```
extend object type object_type with
link
    obj_used_in_map: (navigate) implicit link (system_key) to method_selection reverse
        uses_object;
end object_type;
```

(4)
```
interface_type: child type of type with
link
    parent_interface: (navigate) reference link (number) to interface_type reverse
        child_interface;
    child_interface: (navigate) implicit link (system_key) to interface_type reverse
        parent_interface;
    has_operation: (navigate) reference link (uuid: string) to operation_type reverse
        used_in_interface;
end interface_type;
```

(5)
```
operation_type: child type of type with
attribute
    operation_kind: (read) enumeration (NORMAL_CALL, ONEWAY_CALL) := NORMAL_CALL;
link
    used_in_interface: (navigate) implicit link (system_key) to interface_type reverse
        has_operation;
    has_parameter: (navigate) reference link (position: natural; name) to parameter_type
        reverse parameter_of with
    attribute
        parameter_mode: (read) enumeration (IN, OUT, INOUT) := IN;
    end has_parameter;
    has_return_value: (navigate) reference link to parameter_type reverse return_value_of;
    op_used_in_map: (navigate) implicit link (system_key) to method_selection reverse
        uses_operation;
end operation_type;
```

(6)
```
parameter_type: child type of type with
link
    parameter_of: (navigate) implicit link (system_key) to operation_type reverse
        has_parameter;
    return_value_of: (navigate) implicit link (system_key) to operation_type reverse
        has_return_value;
end parameter_type;
```

(7)
```
data_parameter_type: child type of parameter_type with
link
    constrained_to_attribute_type: (navigate) reference link to attribute_type;
end data_parameter_type;
```

(8)
```
interface_parameter_type: child type of parameter_type with
link
    constrained_to_interface_type: (navigate) reference link to interface_type;
end interface_parameter_type;
```

(9)
```
object_parameter_type: child type of parameter_type with
link
    constrained_to_object_type: (navigate) reference link to object_type;
end object_parameter_type;
```

(10)
```
extend object type object_type_in_sds with
link
    supports_interface: (navigate) reference link (name) to interface_type_in_sds reverse
        applies_to;
end object_type_in_sds;
```

(11)
```
interface_type_in_sds: child type of type_in_sds with
link
    applies_to: (navigate) implicit link (type_identifier) to object_type_in_sds reverse
        supports_interface;
    in_operation_set: (navigate) reference link (number; name) to operation_type_in_sds
        reverse is_operation_of;
end interface_type_in_sds;
```

(12)
```
operation_type_in_sds: child type of type_in_sds with
link
    is_operation_of: (navigate) implicit link (system_key) to interface_type reverse
        in_operation_set;
end operation_type_in_sds;
```

(13)
```
extend object type method_selection with
link
    uses_operation: (navigate) reference link (number) to operation_type reverse
        op_used_in_map;
    uses_object: (navigate) reference link (number) to object_type reverse obj_used_in_map;
end method_selection;
```

(14)
```
end metasds;
```

(15) This part of the data model is used to define the characteristics of an interface (inheritance, operations, signature, etc.) used at run-time to determine if an invocation is syntactically acceptable (e.g. if the correct number and type of parameters have been passed). Figure 2.1 of annex D gives an overview of the data model, with the new object types, link types and attribute types.

(16) The interfaces are represented by new types "interface_type" and "interface_type_in_sds". An interface type has the following properties:

(17) - The parent interfaces are the interfaces from which an interface can inherit operations.

(18) - The destinations of the "has_operation" links are the operations supported by the interface type. The key is an implementation-defined unique identifier.

(19) An interface type in SDS has the following properties:

(20) - The destinations of the "applies_to" links are the visible object types supporting this interface;

(21) - The destinations of the "in_operation_set" links are the operations of the interface. The two keys of the link type are the local operation name and an increasing integer identifier, used to disambiguate operations in case of overloading.

(22) Operations are represented by new types "operation_type" and "operation_type_in_sds". An operation type has the following properties:

(23) - The operation kind is used to define if the operation must return values or not; see 8.4).

(24) - The destinations of the "has_parameter" links are the parameter types that constitutes the operation signature (excluding the return value). For the parameter mode see 8.4.

(25) - The destinations of the "has_return_value" link is the return value of the operation.

(26) Parameter types are represented by new type "parameter_type", which is specialized to "data_parameter_type", "interface_parameter_type", and "object_parameter_type". The value of a parameter of a data parameter type must be a value of the value type of the destination of the "constrained_to_attribute_type" link. The value of a parameter of an interface parameter type must be an object of an object type that supports the destination of the "constrained_to_interface_type" link. The value of a parameter of an object parameter type must be an object of the object type that is the destination of the "constrained_to_object_type" link.

(27) The type "object_type_in_sds" is extended by the "supports_interface" link type; the destinations of these links are the interfaces that are supported by the origin object type.

G.3.2 New SDS operations

G.3.2.1 SDS_APPLY_INTERFACE_TYPE

(1)
```
SDS_APPLY_INTERFACE_TYPE (
        sds             : Sds_designator,
        interface_type  : Interface_type_nominator_in_sds
        type            : Object_type_nominator_in_sds
)
```

(2) SDS_APPLY_INTERFACE_TYPE extends the object type *type* by the application of the interface type *interface_type* in the SDS *sds*.

(3) An "supports_interface" link and its reverse "applies_to" link are created between the type in SDS *type_in_sds* associated with *type* in *sds* and the interface type in SDS *interface_type_in_sds* associated with *interface_type* in *sds*.

(4) Neither the application of this link nor the notion of its existence is inherited by the child type of *type*.

(5) Write locks of the default mode are obtained on the created links.

Errors

(6) ACCESS_ERRORS (*type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(7) ACCESS_ERRORS (*interface_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(8) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(9) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(10) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(11) SDS_IS_UNKNOWN (*sds*)

(12) TYPE_IS_ALREADY_APPLIED (*sds*,*interface_type*,*type*)

(13) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *interface_type*)

(14) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type)*

G.3.2.2 SDS_APPLY_OPERATION_TYPE

(1)
```
SDS_APPLY_OPERATION_TYPE(
        sds             : Sds_designator,
        operation_type  : Operation_type_nominator_in_sds,
        type            : Interface_type_nominator_in_sds
)
```

(2) SDS_APPLY_OPERATION_TYPE extends the interface type *type* by the application of the operation type *operation_type* in the SDS *sds.*

(3) An "in_operation_set" link and its reverse "is_operation_of" link are created between the type in SDS *type_in_sds* associated with *type* in *sds* and the operation type in SDS *operation_type_in_sds* associated with *operation_type* in *sds*.

(4) In addition an "has_operation " link is created between *type* and *operation_type*, together with its reverse "used_in_interface" link, unless this link has already been applied to one of the ancestors of *type.*

(5) Write locks of the default mode are obtained on the created links.

Errors

(6) ACCESS_ERRORS (*type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(7) ACCESS_ERRORS (*operation_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(8) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(9) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(10) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(11) SDS_IS_UNKNOWN (*sds*)

(12) TYPE_IS_ALREADY_APPLIED (*sds*,*attribute_type*,*type*)

(13) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *attribute_type*)

(14) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type)*

G.3.2.3 SDS_CREATE_DATA_PARAMETER_TYPE

(1)
```
SDS_CREATE_DATA_PARAMETER_TYPE(
    sds             : Sds_designator,
    local_name      : [Name],
    data_type       : Attribute_type_nominator
)
    new_parameter   : Data_parameter_type_nominator
```

(2) SDS_CREATE_DATA_PARAMETER_TYPE creates a new parameter that is bound to support the data type *data_type.*

(3) The operation creates a "constrained_to_data_type" link from *new_parameter* to *data_type*.

Errors

(4) ACCESS_ERRORS (*local_name,* ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*data_type,* ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(7) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(8) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(9) SDS_IS_UNKNOWN (*sds*)

(10) TYPE_IS_ALREADY_CONSTRAINED (*sds*, *data_type*)

(11) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *data_type*)

G.3.2.4 SDS_CREATE_INTERFACE_PARAMETER_TYPE

(1)
```
SDS_CREATE_INTERFACE_PARAMETER_TYPE (
    sds                 : Sds_designator,
    local_name          : [Name],
    interface_type      : Interface_type_nominator
)
    new_parameter       : Interface_parameter_type_nominator
```

(2) SDS_CREATE_INTERFACE_PARAMETER_TYPE creates a new parameter that is bound to support the interface type *interface_type*.

(3) The operation creates a "constrained_to_interface_type" link from *new_parameter* to *interface_type*.

Errors

(4) ACCESS_ERRORS (*local_name*, ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*interface_type*, ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(7) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(8) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(9) SDS_IS_UNKNOWN (*sds*)

(10) TYPE_IS_ALREADY_CONSTRAINED (*sds*, *interface_type*)

(11) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *interface_type*)

G.3.2.5 SDS_CREATE_INTERFACE_TYPE

(1)
```
SDS_CREATE_INTERFACE_TYPE(
    sds                 : Sds_designator,
    local_name          : [Name],
    parents             : Interface_type_nominators_in_sds,
    new_operations      : Operation_type_nominators_in_sds
)
    new_interface       : Interface_type_nominator_in_sds
```

(2) SDS_CREATE_INTERFACE_TYPE creates a new interface type *new_interface* and its associated interface type in SDS *new_interface_in_sds* in the SDS *sds*.

(3) The type identifier of *new_interface* is set to an implementation-defined value which identifies the interface within a PCTE installation.

(4) The operation creates a "definition" link from *sds* to *new_interface_in_sds*; the key of the link is the system-assigned type identifier of *new_interface.* The operation also creates an "of_type" link from *new_interface_in_sds* to *new_interface.*

(5) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_interface_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link. "parent_interface" links are created from *new_interface* to each of *parents*, together with their reverse "child_interface" link.

(6) The three definition mode attributes of *new_interface_in_sds* are set to 1, representing CREATE_MODE, and its creation and importation time is set to the system time. If *local_name* is supplied, the annotation of *new_interface_in_sds* is set to the complete name of the created interface; otherwise it is set to the empty string.

(7) The new objects reside in 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) An "in_operation_set" link is created from *new_interface_in_sds* to each operation type in SDS of *new_operations*, with key composed of the local name of the operation type in SDS and a number initially 1 and incremented by 1 for each link.

(10) A "has_operation" link is created from *interface_type* to *operation_type*, with a system generated key.

(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 (elements of *parents*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(13) ACCESS_ERRORS (elements of *new_operations*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(14) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(15) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_interface_in_sds*)

(16) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(17) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(18) SDS_IS_UNKNOWN

(19) TYPE_IS_UNKNOWN_IN_SDS (*sds*, element of *parents*)

(20) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(21) TYPE_NAME_IS_INVALID (*local_name*)

G.3.2.6 SDS_CREATE_OBJECT_PARAMETER_TYPE

(1)
```
SDS_CREATE_OBJECT_PARAMETER_TYPE (
    sds                 : Sds_designator,
    local_name          : [ Name ],
    object_type         : Object_type_nominator
)
    new_parameter       : Object_parameter_type_nominator
```

(2) SDS_CREATE_OBJECT_PARAMETER_TYPE creates a new parameter that is bound to support the object type *object_type*.

(3) The operation creates a "constrained_to_object_type" link from *new_parameter* to *object_type*.

Errors

(4) ACCESS_ERRORS (*local_name,* ATOMIC, MODIFY, APPEND_LINKS)

(5) ACCESS_ERRORS (*object_type,* ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*sds,* ATOMIC, READ, NAVIGATE)

(7) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(8) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(9) SDS_IS_UNKNOWN (*sds*)

(10) TYPE_IS_ALREADY_CONSTRAINED (*sds, object_type*)

(11) TYPE_IS_UNKNOWN_IN_SDS (*sds, object_type*)

G.3.2.7 SDS_CREATE_OPERATION_TYPE

(1)
```
SDS_CREATE_OPERATION_TYPE(
    sds             : Sds_designator,
    local_name      : [ Name ],
    parameters      : Parameter_type_nominators,
    return_value    : Parameter_type_nominator
)
    new_operation : Operation_type_nominator_in_sds
```

(2) SDS_CREATE_OPERATION_TYPE creates a new operation type *new_operation* and its associated operation type in SDS *new_operation_in_sds* in the SDS *sds*.

(3) The type identifier of *new_operation* is set to an implementation-defined value which identifies the interface within a PCTE installation.

(4) The operation creates a "definition" link from *sds* to *new_operation_in_sds*; the key of the link is the system-assigned type identifier of *new_operation.* The operation also creates an "of_type" link from *new_operation_in_sds* to *new_operation.*

(5) If *local_name* is supplied, a "named_definition" link is created from *sds* to *new_operation_in_sds* with *local_name* as key, together with its reverse "named_in_sds" link.

(6) The three definition mode attributes of *new_operation_in_sds* are set to 1, representing CREATE_MODE, and its creation and importation time is set to the system time. If *local_name* is supplied, the annotation of *new_operation_in_sds* is set to the complete name of the created operation; otherwise it is set to the empty string.

(7) The new objects reside in the same volume as *sds*. Their ACLs 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.

(10) The operation creates a "has_parameter" link from the operation *new_operation* to each of the parameters in the *parameters* sequence. The key of each link is the position in the sequence and an implementation-defined name of the parameter type.

Errors

(11) ACCESS_ERRORS (elements of *parameters*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(12) ACCESS_ERRORS (*return_value*, ATOMIC, CHANGE, APPEND_IMPLICIT)

(13) LIMIT_WOULD_BE_EXCEEDED (MAX_DEFINITION_NAME_SIZE)

(14) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*new_operation_in_sds*)

(15) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(16) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(17) SDS_IS_UNKNOWN (*sds*)

(18) TYPE_NAME_IN_SDS_IS_DUPLICATE (*sds*, *local_name*)

(19) TYPE_NAME_IS_INVALID (*local_name*)

G.3.2.8 SDS_IMPORT_INTERFACE_TYPE

(1)
```
SDS_IMPORT_INTERFACE_TYPE(
    to_sds        : Sds_designator,
    from_sds      : Sds_designator,
    type          : Interface_type_nominator_in_sds,
    local_name    : [ Name ],
    import_scope  : Interface_scope
)
```

(2) SDS_IMPORT_INTERFACE_TYPE imports the interface type *type* from the SDS *from_sds* to the SDS *to_sds*.

(3) The importation of an interface type implies the implicit importation of all its ancestor types if not already in *to_sds*. The operations applied to the explicitly or implicitly imported types are not imported, nor is the notion of their application, unless *import_scope* is set to ALL_OPERATIONS. The interfaces implicitly imported do not have a local name assigned to them within *to_sds*.

(4) The importation of an interface type (either implicitly or explicitly) results in the creation of an interface 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 interface 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 interface 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 *from_sds*.

(6) Each of the three definition mode attributes of each new type in SDS is set to the export mode for 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 in the same volume as *to_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) 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 (interface type in SDS associated with type in *from_sds*, ATOMIC, READ, EXPLOIT_SCHEMA)

(15) ACCESS_ERRORS (an imported type, ATOMIC, CHANGE, APPEND_IMPLICIT)

(16) For each ancestor interface type A of *type* not already present in *to_sds*:
ACCESS_ERRORS (A, ATOMIC, CHANGE, APPEND_IMPLICIT)

(17) If *sds* has OWNER granted or denied:
OWNER_PROPAGATION_ERRORS_ON_COMPONENT_CREATION (*type_in_sds*)

(18) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(19) SDS_IS_UNKNOWN (*to_sds*)

(20) SDS_IS_UNKNOWN (*from_sds*)

(21) TYPE_IS_ALREADY_KNOWN_IN_SDS (*type*, *to_sds)*

(22) If *local_name* is supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, *local_name*)

(23) If *local_name* is not supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, local name of *type* in *from_sds*)
TYPE_IS_UNKNOWN_IN_SDS (*from_sds*, *type)*
TYPE_NAME_IS_INVALID (*local_name)*

G.3.2.9 SDS_IMPORT_OPERATION_TYPE

(1)
```
SDS_IMPORT_OPERATION_TYPE (
    to_sds          : Sds_designator,
    from_sds        : Sds_designator,
    type            : Operation_type_nominator_in_sds,
    local_name      : [ Name ]
)
```

(2) SDS_IMPORT_OPERATION_TYPE imports the operation type *type* from the SDS *from_sds* to the SDS *to_sds*.

(3) The operation creates an operation type in SDS *type_in_sds* in *to_sds* associated with *type*. 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.

(4) An "of_type" link from each new type in SDS to its associated type and its reverse "has_type_in_sds" link are created.

(5) 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 it is the local name of *type* in *from_sds*.

(6) Each of the three definition mode attributes of *type_in_sds* is set to the export mode for 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 in the same volume as *to_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) 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 (operation type in SDS associated with type in *from_sds*, ATOMIC, READ, EXPLOIT_SCHEMA)

(15) ACCESS_ERRORS (an imported type, ATOMIC, CHANGE, APPEND_IMPLICIT)

(16) If *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_UNKNOWN (*to_sds*)

(20) SDS_IS_UNKNOWN (*from_sds*)

(21) TYPE_IS_ALREADY_KNOWN_IN_SDS (*type, to_sds)*

(22) If *local_name* is supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, *local_name*)

(23) If *local_name* is not supplied:
TYPE_NAME_IN_SDS_IS_DUPLICATE (*to_sds*, local name of *type* in *from_sds*)
TYPE_IS_UNKNOWN_IN_SDS (*from_sds*, *type)*
TYPE_NAME_IS_INVALID (*local_name*)

G.3.2.10 SDS_UNAPPLY_INTERFACE_TYPE

(1)
```
SDS_UNAPPLY_INTERFACE_TYPE(
    sds             : Sds_designator,
    interface_type  : Interface_type_nominator_in_sds
    type            : Object_type_nominator_in_sds
)
```

(2) SDS_UNAPPLY_INTERFACE_TYPE removes the application of the interface type in the SDS *interface_type_in_sds* associated with the type *interface_type* in the SDS *sds* from the type in SDS *type_in_sds* associated with the interface type *type* in *sds*.

(3) The "supports_interface" link between *type_in_sds* and *operation_type_in_sds* and its reverse "applies_to" link are deleted.

(4) Write locks of the default mode are obtained on the deleted links.

Errors

(5) ACCESS_ERRORS (*type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*interface_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(7) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(8) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(9) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(10) SDS_IS_UNKNOWN (*sds*)

(11) TYPE_IS_ALREADY_APPLIED (*sds,interface_type,type*)

(12) TYPE_IS_UNKNOWN_IN_SDS (*sds, interface_type*)

(13) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type*)

G.3.2.11 SDS_UNAPPLY_OPERATION_TYPE

(1)
```
SDS_UNAPPLY_OPERATION_TYPE(
    sds              : Sds_designator,
    operation_type   : Operation_type_nominator_in_sds
    type             : Interface_type_nominator_in_sds
)
```

(2) SDS_UNAPPLY_OPERATION_TYPE remove the application of the operation type in SDS *operation_type_in_sds* associated with the operation type *operation_type* in the SDS *sds* from the type in SDS *type_in_sds associated* with the interface type *type* in *sds*.

(3) The "in_operation_set" link between *type_in_sds* and *operation_type_in_sds* and its reverse "is_operation_of" link are deleted.

(4) Write locks of the default mode are obtained on the deleted links.

Errors

(5) ACCESS_ERRORS (*type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(6) ACCESS_ERRORS (*operation_type_in_sds*, ATOMIC, MODIFY, APPEND_LINKS)

(7) ACCESS_ERRORS (*sds*, ATOMIC, READ, NAVIGATE)

(8) PRIVILEGE_IS_NOT_GRANTED (PCTE_SCHEMA_UPDATE)

(9) SDS_IS_IN_A_WORKING_SCHEMA (*sds*)

(10) SDS_IS_UNKNOWN (*sds*)

(11) TYPE_IS_ALREADY_APPLIED (*sds,attribute_type,type*)

(12) TYPE_IS_UNKNOWN_IN_SDS (*sds, attribute_type*)

(13) TYPE_IS_UNKNOWN_IN_SDS (*sds*, *type)*

G.3.3 Modified SDS operations (see 10.2.23)

G.3.3.1 SDS_REMOVE_TYPE

(1)
```
SDS_REMOVE_TYPE (
    sds     : Sds_designator,
    type    : Type_nominator_in_sds
)
```

Additional errors

(2) If the conditions for the deletion of the "type" object T associated with *type* are satisfied:

(3) If T is an interface type, for each parent interface P of T:
ACCESS_ERRORS (P, ATOMIC, CHANGE, WRITE_IMPLICIT)

G.4 DDL extensions (see annex B)

G.4.1 SDSs and clauses

Syntax

(1) import type = '**object**', '**type**' | '**attribute**', '**type**' | '**link**', '**type**' | '**enumeral**', '**type**' | '**interface**', '**type**' | '**operation**', '**type**';

Meaning

(2) Import types are extended to allow importation of interface and operation types.

G.4.2 Object types

Syntax

(1)
```
object type declaration =
local name, ':', [ type mode declaration ], [ 'child', 'type', 'of', object type list ], [ 'with',
[ 'contents', contents type indication, ';' ],
[ 'attribute',
    attribute indication list, ';' ],
[ 'link',
    link indication list, ';' ],
[ 'interface'
    interface indication list, ';'],
[ 'component',
    component indication list, ';' ],
'end', local name ];
```

(2)
```
object type extension =
'extend', 'object', 'type', local name,'with',
[ 'attribute',
    attribute indication list, ';' ],
[ 'link',
     link indication list, ';' ],
[ 'interface'
    interface indication list, ';'],
[ 'component',
    component indication list, ';' ],
'end', local name;
```

(3) interface indication list = interface indication list item, { ';', interface indication list item };

(4) interface indication list item = interface type name | interface type definition;

Constraints

(5) Each interface type name in an interface indication list must be the local name of an interface type introduced earlier in the specification by an interface type declaration or a type importation.

(6) All the interface types in the list must be different.

Meaning

(7) Object type declarations and extensions are extended to include optional interface types. Each interface indication list item defines an interface supported by the object type.

G.4.3 Interface types

Syntax

(1)
```
interface type definition =
    local name, ':', ['child', 'type', 'of', interface type list], ['with',
    ['operation'
        operation indication list, ';'],
    ['applied'
        object type list, ';'],
    'end', local name ];
```

(2)
```
interface type declaration =
    interface, interface type definition;
```

(3)
```
interface type extension =
    'extend', 'interface', 'type', local name, 'with'
    [ 'operation'
        operation indication list, ';' ],
    'end', local name;
```

(4) operation indication list = operation indication list item, { ';', operation indication list item };

(5) operation indication list item = operation type name | operation type definition;

Constraints

(6) The local name after '**end**' in an interface type definition or interface type extension, if present, must be the same as the first local name of that interface type definition or interface type extension.

(7) In an interface type definition the local name must be distinct from the local names of all other types defined in the same SDS as the interface type definition.

(8) In an interface type extension the local name must be the name of an interface type introduced earlier in the SDS by an interface type definition or a type importation.

(9) Each operation type name in an operation indication list must be the local name of an operation type introduced earlier in the specification by an operation type definition or a type importation.

(10) Each object type name in an object type list after the keyword '**applied**' must be the local name of an object type introduced earlier in the specification by an object type declaration or a type importation.

Meaning

(11) An *interface type definition* defines an interface type, and an interface type in SDS in the current SDS with the local name within that SDS. The new interface type has the following characteristics (see 8.3).

(12) - The operation types are all those defined by the operation indications in the operation indication list after '**operation**'.

(13) - The parent interfaces are all those in the interface type list after '**child type of**'; the interface type is added to the child interfaces of all its parent interfaces. The interface type has no child interfaces initially.

(14) The new interface type in SDS has the following characteristics (see 8.7).

(15) - The applied object types are all those in the object type list after '**applied**'.

G.4.4 Operation types

Syntax

(1)
```
operation type definition =
    operation kind,
    local name, [ 'with',
    [ 'parameter',
        parameter indication list, ';' ]
    [ 'return', parameter type name, ';' ]
    'end', local name ];
```

(2)
```
operation type declaration =
    'operation', operation type definition;
```

(3)
```
operation kind = 'normal' | 'oneway';
```

(4)
```
parameter indication list = parameter indication list item, { ';', parameter indication list item };
```

Constraints

(5) The local name after '**end**' in an operation type definition, if present, must be the same as the first local name of that operation type definition.

(6) In an definition the local name must be distinct from the local names of all other types defined in the same SDS as the operation type definition.

(7) Each parameter type name in a parameter indication list must be the local name of a parameter type introduced earlier in the specification by a parameter type declaration.

Meaning

(8) An *operation type definition* defines an operation type, and an operation type in SDS in the current SDS with the local name within that SDS. The new operation type has the following characteristics (see 8.4).

(9) - The 'used in interface' interface types are all those interface types for which the operation occurs in the operation indication list of the interface type declaration or extension.

(10) - The sequence of parameter types is defined by the parameter indication list after '**parameter**'.

(11) - The kind is defined by the operation kind.

(12) - The return value parameter type is defined by the parameter type name after '**return**'.

G.4.5 Parameter types

Syntax

(1) parameter indication list item = [parameter mode], ':', parameter type name;

(2) parameter mode = '**in**' | '**out**' | '**inout**';

(3) parameter type name = object type name | interface type name | attribute type name;

Meaning

(4) A *parameter indication list item* defines a parameter type, and its associated mode.

G.4.6 Names

Syntax

(1) interface type name = global name | local name;

(2) interface type list = interface type name, {',', interface type name};

(3) operation type name = local name;

(4) operation type list = operation type name, {',', operation type name};

Constraints

(5) An interface or operation type name without an SDS name must occur in an interface type declaration or an operation type definition, respectively, within the local specification.

(6) The local name of an interface type name with an SDS name must occur in an interface type declaration, respectively, within the specification with that SDS name.

Meaning

(7) See B.8.

Index of Operations

Index of Error Conditions

Grouped errors

Individual errors

Index of Technical Terms

The entries in this index are technical terms defined in clauses 8 to 24. Page references are given to points of definition; these are as follows:

- VDM type definitions: e.g. Attribute_designator
- VDM field names: e.g. APPLIED_LINK_TYPES
- DDL type names: e.g. type_in_sds
- other technical terms (defined in the text): e.g. *ancestor types*

The original form (capital letters and underscores) is retained as a guide to finding the definition, but in running text the form of VDM-SL and DDL terms is modified for readability: see clauses A.3 and B.8.

Printed copies can be ordered from:

ECMA
114 Rue du Rhône
CH-1204 Geneva
Switzerland

Fax: +41 22 849.60.01
Internet: documents@ecma.ch

Files can be downloaded from our FTP site, **ftp.ecma.ch,** logging in as **anonymous** and giving your E-mail address as **password**. This Standard is available from library **ECMA-ST** as a compacted, self-expanding file in MSWord 6.0 format (file E149-DOC.EXE) and as two Acrobat PDF files (file E149-V1.PDF and E149-V2.PDF). File E149-EXP.TXT gives a short presentation of the Standard.

Our web site, http://www.ecma.ch, gives full information on ECMA, ECMA activities, ECMA Standards and Technical Reports.

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