Standard ECMA-255
December 1996

ECMA

Standardizing Information and Communication Systems

Portable Common Tool Environment (PCTE) - Object-Orientation Extensions - Abstract Specification

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

Standard ECMA-255
December 1996

ECMA

Standardizing Information and Communication Systems

Portable Common Tool Environment (PCTE) - Object-Orientation Extensions - Abstract Specification

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

GL Ecma-255.doc 28-02-97 12,04

Brief History

With the development of object-oriented methods and programming languages, there is an increasing demand to enhance PCTE with the ability to represent active objects, i.e. objects characterized by interfaces defining the operations which are applicable to the objects of a given type, and to define their dynamic behaviour.

In 1993, several projects addressed this problem. Two of them produced results which were made publicly available and were thereafter used as input to this Standard:

- the Portable Common Interface Set (PCIS) project of the NATO Special Working Group on APSE;
- the Object Oriented Tool Interface Set (OOTIS) project of IBM.

By the end of 1993, the US Department of Defense, the US National Institute of Standards and Technology, and the Object Management Group (OMG) decided to create an initiative, called the North American PCTE Initiative (NAPI) in order to resolve this problem (among others).

At the same time, the technical committee TC33 of ECMA decided to create a new working group, named TGOO, to add object orientation and support of fine-grain objects to PCTE. The NAPI and TGOO working groups soon decided to merge their efforts in order to do a joint specification.

In 1994, the NAPI group transformed itself into the OMG Special Interest Group on PCTE (OMG PCTE SIG) and the joint work with ECMA TC33/TGOO continued.

In September 1994, a new working group ISO/IEC JTC1/SC22/WG22 was created to manage the maintenance of the PCTE International Standard ISO/IEC 13719, which is equivalent to ECMA-149, 3rd edition. That working group participated to the review of the final drafts of this Standard.

This Standard is the result of all these collaborative efforts.

This Standard has been adopted by the ECMA General Assembly of December 1996.

Table of contents

1 Scope

(1) This ECMA Standard specifies object-orientation extensions to PCTE as defined in ECMA-149.

(2) The extensions specified in this Standard allow tools to be written in an object-oriented way, by attaching interfaces and behaviour to standard PCTE object types and exploiting them through new PCTE operations.

2 Conformance

2.1 Conformance of bindings

(1) The provisions of 2.1 of ECMA-149 are extended to cover the operations, datatypes, and error values of this Standard.

2.2 Conformance of implementations

(1) A new module is added to those defined in 2.2 of ECMA-149:

(2) - the *object-orientation module* consists of the datatypes and operations defined in clauses 8, 9, and 10, and annex B, of this Standard.

(3) A new conformance level is added to those defined in 2.2 of ECMA-149:

(4) - an implementation conforms to ECMA-149 with object-orientation if and only if it implements the core module and the object-orientation module.

3 Normative references

(1) The following standard contains provisions which, through reference in this text, constitute provisions of this Standard. At the time of publication, the edition indicated was valid. All standards are subject to revision, and parties to agreements based on this Standard are encouraged to investigate the possibility of applying the most recent edition of the standard indicated below:

(2) ECMA-149 Portable Common Tool Environment (PCTE) — Abstract Specification (3rd Edition, December 1994)

4 Definitions

4.1 Technical terms

(1) All technical terms used in this Standard, other than a few in widespread use, are defined in the text, usually in a formal notation, or in ECMA-149. All identifiers defined in VDM-SL or in DDL (see 5.2 of ECMA-149) are technical terms; apart from those, a defined technical term is printed in italics at the point of its definition.

4.2 Other terms

(1) For the purpose of this Standard, the definitions of 4.2 of ECMA-149 and the following apply:

(2) **4.2.1 operation**: a name plus a signature that is used in the context of an invocation to trigger the execution of a specific method.

(3) **4.2.2 interface**: a set of operations; interfaces are a convenient way to group operations so that they can be referred to together, e.g. to define other interfaces by inheritance.

(4) **4.2.3 method**: the set of actions triggered by an operation.

5 Formal notations

(1) The formal notations used are defined in clause 5 of ECMA-149.

(2) This Standard extends the DDL notation as defined in ECMA-149, by adding object-oriented notations — see annex A of this Standard.

6 Overview of the object-orientation extension

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

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

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

7 Outline of the Standard

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

(2) The normative definition of the object-orientation extension to PCTE is in clauses 8, 9, 10, and annexes A and B. Clause 8 defines the basic datatypes; clause 9 defines the datatypes of the module and method mapping parts of the data model, and the operations for invocation management that use them. Clause 10 defines the datatypes of the interface part of the data model, and the operations for schema management that use them. Annex A defines extensions to DDL for defining operation and interface types. Annex B defines the new error specifications.

(3) Annex C is informative. It contains the collected extensions to the predefined SDSs.

8 Object-orientation extension foundation

8.1 The state

(1) The state of ECMA-149 is not changed by this Standard.

8.2 Types

(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 from their ECMA-149 definitions 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 10.1).

8.3 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.

8.4 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.

8.5 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.

8.6 Types in SDS

(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

(7) The datatypes Type_in_sds and Type_nominator_in_sds are extended from their ECMA-149 definitions to include the new datatypes Interface_type_in_sds and Operation_type_in_sds, and their corresponding type nominator in SDS datatypes, respectively.

8.7 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.

8.8 Operation types in SDS

(1) Operation_type_in_sds :: Type_in_sds_common_part

8.9 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.

9 Object-oriented invocation management

9.1 Invocation concepts

9.1.1 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.

9.1.2 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 10.1.)

(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. The following 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.

operation id	type identifier	Attribute_1	Attribute_n	number
(17)	(18)	"user"	"Platform_1"	(19)
(20)	(21)	-	"Platform_2"	(22)
(23)	(24)	"system"	"Platform_3"	(25)

9.2 Invocation operations

9.2.1 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)

9.2.2 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*)

9.2.3 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*)

9.2.4 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*)

10 Object-oriented schema management

10.1 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.

10.2 New SDS operations

10.2.1 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)*

10.2.2 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)*

10.2.3 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*)

10.2.4 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*)

10.2.5 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*)

10.2.6 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*)

10.2.7 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*)

10.2.8 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)*

10.2.9 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*)

10.2.10 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*)

10.2.11 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)*

10.3 Modified SDS operations

10.3.1 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)

Annex A

(normative)

DDL extensions

A.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.

A.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 as defined in ECMA-149 are extended to include optional interface types. Each interface indication list item defines an interface supported by the object type.

A.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**'.

A.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 operationindication 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**'.

A.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

A *parameter indication list item* defines a parameter type, and its associated mode.

A.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 of ECMA-149.

Annex B

(normative)

Specification of new errors

(1) NUMBER_OF_PARAMETERS_IS_WRONG (*operation*) The number of parameters of the operation *operation* does not match the operation signature.

(2) OPERATION_METHOD_CANNOT_BE_FOUND (*operation*) The method related to the operation *operation* cannot be found in the method repository.

(3) OPERATION_METHOD_CANNOT_BE_ACTIVATED (*operation*) The method related to the operation *operation* cannot be activated.

(4) 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.

(5) 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*.

Annex C

(informative)

Extensions to the predefined schema definition sets

C.1 The system SDS

```
(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)  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;
```

```
(9)     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;

(10)    dispatching_context: child type of file;

(11)    extend object type process with
        link
            has_dispatching_context: (navigate) designation link to dispatching_context;
        end process;

(12)    extend object type static_context with
        link
            implementing_tool: (navigate) implicit link to tool
                reverse executable;
        end static_context;

(13)    end system;
```

C.2 The metasds SDS

```
(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;
```

ECMA

114 Rue du Rhône
CH-1204 Geneva
Switzerland

This Standard ECMA-255 is available free of charge in printed form{ and as a file}.

See inside cover page for instructions