Naming Service Specification 3

3.1 Service Description

Note – EDITING INSTRUCTIONS - This chapter is a replacement for the CORBA Services Specification Chapter 3. Changebars are relative to ptc/99-09-02

3.1.1 Overview

A name-to-object association is called a *name binding*. A name binding is always defined relative to a *naming context*. A naming context is an object that contains a set of name bindings in which each name is unique. Different names can be bound to an object in the same or different contexts at the same time. There is no requirement, however, that all objects must be named.

To *resolve a name* is to determine the object associated with the name in a given context. To *bind a name* is to create a name binding in a given context. A name is always resolved relative to a context — there are no absolute names.

Because a context is like any other object, it can also be bound to a name in a naming context. Binding contexts in other contexts creates a *naming graph* — a directed graph with nodes and labeled edges where the nodes are contexts. A naming graph allows more complex names to reference an object. Given a context in a naming graph, a sequence of names can reference an object. This sequence of names (called a *compound name*) defines a path in the naming graph to navigate the resolution process. Figure 3-1 shows an example of a naming graph.

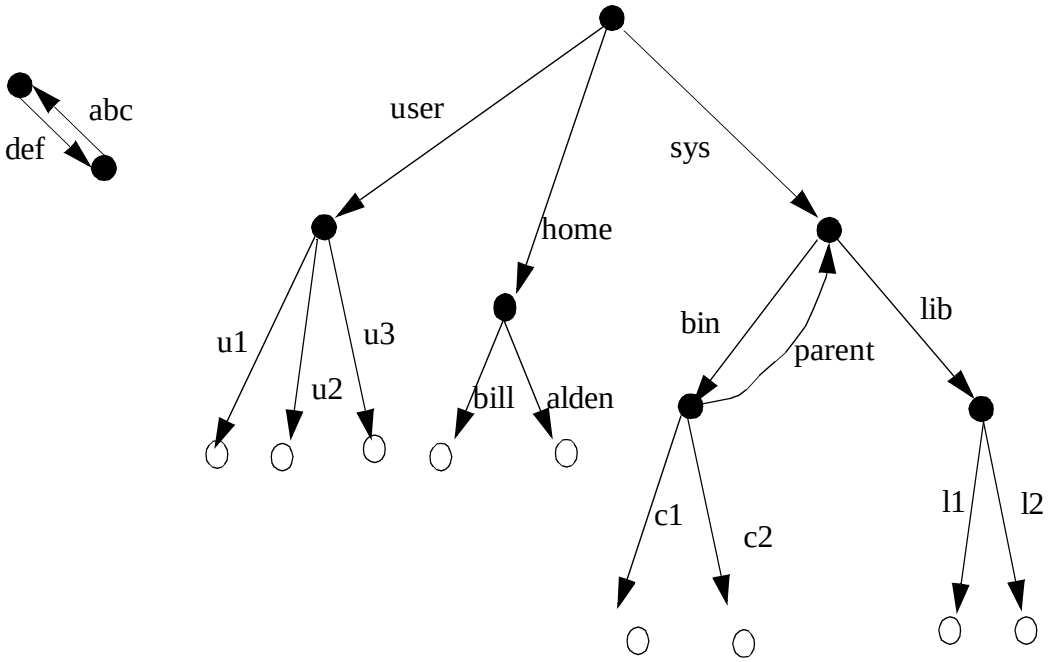

Figure 3-1 A Naming Graph

3.1.2 Names

Many of the operations defined on a naming context take names as parameters. Names have structure. A name is an ordered sequence of *components*.

A name with a single component is called a *simple name*; a name with multiple components is called a *compound name*. Each component except the last is used to name a context; the last component denotes the bound object. The notation:

```
component1/component2/component3
```

indicates a sequence of components.

Note – The slash (/) characters are simply a notation used here and are not intended to imply that names are sequences of characters separated by slashes.

A name component consists of two attributes: the ***`id`*** *attribute* and the ***`kind`*** *attribute*. Both the `id` attribute and the `kind` attribute are represented as IDL strings.

The `kind` attribute adds descriptive power to names in a syntax-independent way. Examples of the value of the `kind` attribute include *c_source*, *object_code*, *executable*, *postscript*, or “ ”. The naming system does not interpret, assign, or manage these values in any way. Higher levels of software may make policies about the use and management of these values. This feature addresses the needs of applications that

use syntactic naming conventions to distinguish related objects. For example Unix uses suffixes such as `.c` and `.o`. Applications (such as the C compiler) depend on these syntactic convention to make name transformations (for example, to transform `foo.c` to `foo.o`).

A sequence of `id` and `kind` pairs forming a name can be expressed as a single string using the syntax described in section 3.5. This allows names to be written down easily or to be presented as a strings in user interfaces. In addition, section 3.6 describes a way to express a name relative to a particular naming context in URL format. The URL representation provides a human-readable form of an object reference that is named in some naming context.

3.1.3 Example Scenarios

This section provides two short scenarios that illustrate how the naming service specification can be used by two fairly different kinds of systems -- systems that differ in the kind of implementations used to build the Naming Service and that differ in models of how clients might use the Naming Service with other object services to locate objects.

In one system, the Naming Service is implemented using an underlying enterprise-wide naming server such as DCE CDS. The Naming Service is used to construct large, enterprise-wide naming graphs where NamingContexts model "directories" or "folders" and other names identify "document" or "file" kinds of objects. In other words, the naming service is used as the backbone of an enterprise-wide filing system. In such a system, non-object-based access to the naming service may well be as commonplace as object-based access to the naming service.

The Naming Service provides the principal mechanism through which most clients of an ORB-based system locate objects that they intend to use (make requests of). Given an initial naming context, clients navigate naming contexts retrieving lists of the names bound to that context. In conjunction with properties and security services, clients look for objects with certain "externally visible" characteristics, for example, for objects with recognized names or objects with a certain time-last-modified (all subject to security considerations). All objects used in such a scheme register their externally visible characteristics with other services (a name service, a properties service, and so on).

Conventions are employed in such a scheme that meaningfully partition the name space. For example, individuals are assigned naming contexts for personal use, groups of individuals may be assigned shared naming contexts while other contexts are organized in a public section of the naming graph. Similarly, conventions are used to identify contexts that list the names of services that are available in the system (e.g., that locate a translation or printing service).

In an alternative system, the Naming Service can be used in a more limited role and can have a less sophisticated implementation. In this model, naming contexts represent the types and locations of services that are available in the system and a much shallower naming graph is employed. For example, the Naming Service is used to register the object references of a mail service, an information service, a filing service.

Given a handful of references to "root objects" obtained from the Naming Service, a client uses the Relationship and Query Services to locate objects contained in or managed by the services registered with the Naming Service. In such a system, the Naming Service is used sparingly and instead clients rely on other services such as query services to navigate through large collections of objects. Also, objects in this scheme rarely register "external characteristics" with another service - instead they support the interfaces of Query or Relationship Services.

Of course, nothing precludes the Naming Service presented here from being used to provide both models of use at the same time. These two scenarios demonstrate how this specification is suitable for use in two fairly different kinds of systems with potentially quite different kinds of implementations. The service provides a basic building block on which higher-level services impose the conventions and semantics which determine how frameworks of application and facilities objects locate other objects.

3.1.4 Design Principles

Several principles have driven the design of the Naming Service:

1. The design imparts no semantics or interpretation of the names themselves; this is up to higher-level software.
2. The design supports distributed, heterogeneous implementation and administration of names and name contexts.
3. Naming service clients need not be aware of the physical site of name servers in a distributed environment, or which server interprets what portion of a compound name, or of the way that servers are implemented.
4. The Naming Service is a fundamental object service, with no dependencies on other interfaces.
5. Name contexts of arbitrary and unknown implementation may be utilized together as nested graphs of nodes that cooperate in resolving names for a client. No “universal” root is needed for a name hierarchy.
6. Existing name and directory services employed in different network computing environments can be transparently encapsulated using name contexts. All of the above features contribute to making this possible.
7. The design does not address namespace administration. It is the responsibility of higher-level software to administer the namespace.

3.2 The `CosNaming` Module

The `CosNaming` module is a collection of interfaces that together define the Naming Service. This module contains three interfaces:

- The `NamingContext` interface
- The `BindingIterator` interface

- The `NamingContextExt` interface

This section describes these interfaces and their operations in detail.

The `CosNaming` module is shown below.

Note – `Istring` was a “placeholder for a future IDL internationalized string data type” in the original specification. It is maintained solely for compatibility reasons.

```
// File: CosNaming.idl
#ifndef _COSNAMING_IDL_
#define _COSNAMING_IDL_

#pragma prefix "omg.org"

module CosNaming {
      typedef string Istring;

      struct NameComponent {
            Istring id;
            Istring kind;
      };
      typedef sequence<NameComponent> Name;

      enum BindingType { nobject, ncontext };

      struct Binding {
            Name          binding_name;
            BindingType     binding_type;
      };

      // Note: In struct Binding, binding_name is incorrectly defined
      // as a Name instead of a NameComponent. This definition is
      // unchanged for compatibility reasons.
      typedef sequence <Binding> BindingList;

      interface BindingIterator;

      interface NamingContext {
            enum NotFoundReason {
               missing_node, not_context, not_object
            };

            exception NotFound {
                  NotFoundReason  why;
                  Name                rest_of_name;
            };

            exception CannotProceed {
                  NamingContext   cxt;
                  Name                rest_of_name;
            };

            exception InvalidName{};


            exception AlreadyBound {};

            exception NotEmpty{};
```

```
        void    bind(in Name n, in Object obj)
                        raises(
                              NotFound, CannotProceed,
                              InvalidName, AlreadyBound
                        );

        void    rebind(in Name n, in Object obj)
                        raises(NotFound, CannotProceed, InvalidName);

        void    bind_context(in Name n, in NamingContext nc)
                        raises(
                           NotFound, CannotProceed,
                           InvalidName, AlreadyBound
                        );

        void    rebind_context(in Name n, in NamingContext nc)
                        raises(NotFound, CannotProceed, InvalidName);

        Object  resolve (in Name n)
                        raises(NotFound, CannotProceed, InvalidName);

        void    unbind(in Name n)
                    raises(NotFound, CannotProceed, InvalidName);

        NamingContext   new_context();
        NamingContext   bind_new_context(in Name n)
                                raises(
                                      NotFound, AlreadyBound,
                                      CannotProceed, InvalidName
                                );

        void    destroy() raises(NotEmpty);

        void    list(
                        in unsigned long       how_many,
                        out BindingList        bl,
                        out BindingIterator    bi
                    );

};

interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long how_many, out BindingList bl);
        void    destroy();
};

interface NamingContextExt: NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;
```

```
            StringName      to_string(in Name n) raises(InvalidName);
            Name            to_name(in StringName sn)
                                raises(InvalidName);

            exception InvalidAddress {};

            URLString       to_url(in Address addr, in StringName sn)
                                raises(InvalidAddress, InvalidName);

            Object          resolve_str(in StringName n)
                                raises(
                                    NotFound, CannotProceed,
                                    InvalidName
                                );
        };
};
#endif // _COSNAMING_IDL_
```

Resolution of Compound Names

In this specification operations that are performed on compound names recursively perform the *equivalent* of a `resolve` operation on all but the last component of a name before performing the operation on the final name component. The general form is defined as follows:

ctx->op(<c1; c2; ...; cn>) equiv

ctx->resolve(<c1>)->resolve(<c2; cn-1>)->op(<cn>)

where ctx is a naming context, <c1; ...; cn> a compound name, and op a naming context operation.

Note – The intermediate components, <c1: ...; cn> of the compound name must have been bound using `bind_context` or `rebind_context` to take part in the resolve.

3.3 `NamingContext` Interface

The following sections describe the naming context data types and interface in detail.

3.3.1 Structures

NameComponent

```
struct NameComponent {
    Istring Id;
    Istring kind;
};
```

A name component consists of two attributes: the identifier attribute, `id`, and the kind attribute, `kind`.

Both of these attributes are arbitrary-length strings of ISO Latin-1 characters, excluding the ASCII `NUL` character.

When comparing two `NameComponents` for equality both the `id` and the `kind` field must match in order for two `NameComponents` to be considered identical. This applies for zero-length (empty) fields as well. Name comparisons are case sensitive.

An implementation may place limitations on the characters that may be contained in a name component, as well as the length of a name component. For example, an implementation may disallow certain characters, may not accept the empty string as a legal name component, or may limit name components to some maximum length.

Name

A name is a sequence of `NameComponent`s. The empty sequence is not a legal name. An implementation may limit the length of the sequence to some maximum. When comparing `Name`s for equality, each `NameComponent` in the first name must match the corresponding `NameComponent` in the second `Name` for the names to be considered identical.

Binding

```
enum BindingType { nobject, ncontext };
struct Binding {
    Name binding_name;
    BindingType binding_type;
};
typedef sequence<Binding> BindingList;
```

This types are used by the `NamingContext::list`, `BindingIterator::next_n` and `BindingIterator::next_one` operations. A `Binding` contains a `Name` in the member `binding_name`, together with the `BindingType` of that `Name` in the member `binding_type`.

Note – The `binding_name` member is incorrectly typed as a `Name` instead of a `NameComponent`. For compatibility with the original `CosNaming` specification this incorrect definition has been retained. The `binding_name` is used as a `NameComponent` and will always be a `Name` with length of 1.

The value of *`binding_type`* is *`ncontext`* if a `Name` denotes a binding created with one of the following operations:

- `bind_context`
- `rebind_context`
- `bind_new_context`

For bindings created with any other operation, the value of `BindingType` is `nobject`.

3.3.2 Exceptions

The Naming Service exceptions are defined below.

NotFound

```
exception NotFound {
        NotFoundReason  why;
        Name rest_of_name;
};
```

This exception is raised by operations when a component of a name does not identify a binding or the type of the binding is incorrect for the operation being performed. The `why` member explains the reason for the exception and the `rest_of_name` member contains the remainder of the non-working name:

- `missing_node`

  The first name component in `rest_of_name` denotes a binding that is not bound under that name within its parent context.

- `not_context`

  The first name component in `rest_of_name` denotes a binding with a type of `nobject` when the type `ncontext` was required.

- `not_object`

  The first name component in `rest_of_name` denotes a binding with a type of `ncontext` when the type `nobject` was required.

CannotProceed

```
exception CannotProceed {
        NamingContext  cxt;
        Name rest_of_name;
};
```

This exception is raised when an implementation has given up for some reason. The client, however, may be able to continue the operation at the returned naming context.

The `cxt` member contains the context that the operation may be able to retry from.

The `rest_of_name` member contains the remainder of the non-working name.

InvalidName

```
exception InvalidName {};
```

This exception is raised if a `Name` is invalid. A name of length zero is invalid (containing no name components). Implementations may place further limitations on what constitutes a legal name and raise this exception to indicate a violation.

AlreadyBound

```
exception AlreadyBound {};
```

Indicates an object is already bound to the specified name. Only one object can be bound to a particular `Name` in a context.

NotEmpty

```
exception NotEmpty {};
```

This exception is raised by `destroy` if the `NamingContext` contains bindings. A `NamingContext` must be empty to be destroyed.

3.3.3 Binding Objects

The binding operations name an object in a naming context. Once an object is bound, it can be found with the `resolve` operation. The Naming Service supports four operations to create bindings: `bind`, `rebind`, `bind_context` and `rebind_context`. `bind_new_context` also creates a binding, see section 3.3.6.

```
void bind(in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind(in Name n, in Object obj)
        raises(NotFound, CannotProceed, InvalidName);
void bind_context(in Name n, in NamingContext nc)
        raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
void rebind_context(in Name n, in NamingContext nc)
        raises(NotFound, CannotProceed, InvalidName);
```

bind

Creates an `nobject` binding in the naming context.

rebind

Creates an `nobject` binding in the naming context even if the name is already bound in the context.

If already bound, the previous binding must be of type `nobject`; otherwise, a `NotFound` exception with a `why` reason of `not_object` is raised.

bind_context

Creates an `ncontext` binding in the parent naming context. Attempts to bind a nil context raise a `BAD_PARAM` exception.

rebind_context

Creates an `ncontext` binding in the naming context even if the name is already bound in the context.

If already bound, the previous binding must be of type `ncontext`; otherwise, a `NotFound` exception with a `why` reason of `not_context` will be raised.

Usage

If a binding with the specified name already exists, `bind` and `bind_context` raise an `AlreadyBound` exception.

If an implementation places limits on the number of bindings within a context, `bind` and `bind_context` raise the IMP_LIMIT system exception if the new binding cannot be created.

Naming contexts bound using `bind_context` and `rebind_context` participate in name resolution when compound names are passed to be resolved; naming contexts bound with `bind` and `rebind` do not.

Use of `rebind_context` may leave a potential orphaned context (one that is unreachable within an instance of the Name Service). Policies and administration tools regarding potential orphan contexts are implementation-specific.

If `rebind` or `rebind_context` raise a `NotFound` exception because an already existing binding is of the wrong type, the `rest_of_name` member of the exception has a sequence length of 1.

3.3.4 Resolving Names

The `resolve` operation is the process of retrieving an object bound to a name in a given context. The given name must exactly match the bound name. The naming service does not return the type of the object. Clients are responsible for "narrowing" the object to the appropriate type. That is, clients typically cast the returned object from `Object` to a more specialized interface. The IDL definition of the `resolve` operation is:

```
Object resolve (in Name n)
        raises (NotFound, CannotProceed, InvalidName);
```

Names can have multiple components; therefore, name resolution can traverse multiple contexts. These contexts can be federated between different Naming Service instances.

3.3.5 Unbinding Names

The `unbind` operation removes a name binding from a context. The definition of the `unbind` operation is:

```
void unbind(in Name n)
        raises (NotFound, CannotProceed, InvalidName);
```

3.3.6 Creating Naming Contexts

The Naming Service supports two operations to create new contexts: `new_context` and `bind_new_context`.

```
NamingContext new_context();
NamingContext bind_new_context(in Name n)
            raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
```

new_context

This operation returns a new naming context. The new context is not bound to any name.

`bind_new_context`

This operation creates a new context and creates an `ncontext` binding for it using the name supplied as an argument.

Usage

If an implementation places limits on the number of naming contexts, both `new_context` and `bind_new_context` can raise the `IMP_LIMIT` system exception if the context cannot be created. `bind_new_context` can also raise `IMP_LIMIT` if the bind would cause an implementation limit on the number of bindings in a context to be exceeded.

3.3.7 Deleting Contexts

The `destroy` operation deletes a naming context.

```
void destroy()
        raises(NotEmpty);
```

This operation destroys its naming context. If there are bindings denoting the destroyed context, these bindings are *not* removed. If the naming context contains bindings, the operation raises `NotEmpty`.

3.3.8 Listing a Naming Context

The `list` operation allows a client to iterate through a set of bindings in a naming context.

```
void list (in unsigned long how_many,
                out BindingList bl, out BindingIterator bi);
};
```

`list` returns the bindings contained in a context in the parameter `bl`. The `bl` parameter is a sequence where each element is a `Binding` containing a `Name` of length 1 representing a single `NameComponent`.

The `how_many` parameter determines the maximum number of bindings to return in the parameter `bl`, with any remaining bindings to be accessed through the returned `BindingIterator bi`.

- A non-zero value of `how_many` guarantees that `bl` contains at most `how_many` elements. The implementation is free to return fewer than the number of bindings requested by `how_many`. However, for a non-zero value of `how_many`, it may not return a `bl` sequence with zero elements unless the context contains no bindings.
- If `how_many` is set to zero, the client is requesting to use only the `BindingIterator bi` to access the bindings and `list` returns a zero length sequence in `bl`.
- The parameter `bi` returns a reference to an iterator object.

- If the `bi` parameter returns a non-nil reference, this indicates that the call to `list` may not have returned all of the bindings in the context and that the remaining bindings (if any) must be retrieved using the iterator. This applies for all values of `how_many`.
- If the `bi` parameter returns a nil reference, this indicates that the `bl` parameter contains all of the bindings in the context. This applies for all values of `how_many`.

3.4 The `BindingIterator` Interface

The `BindingIterator` interface allows a client to iterate through the bindings using the `next_one` or `next_n` operations:

If a context is modified in between calls to `list`, `next_one`, or `next_n`, the behavior of further calls to `next_one` or `next_n` is implementation-dependent.

```
interface BindingIterator {
    boolean next_one(out Binding b);
    boolean next_n(in unsigned long how_many,
                   out BindingList bl);
   void destroy();
};
```

next_one

The `next_one` operation returns the next binding. It returns true if it is returning a binding, false if there are no more bindings to retrieve. If `next_one` returns false, the returned `Binding` is indeterminate

Further calls to `next_one` after it has returned false have undefined behavior.

next_n

`next_n` returns, in the parameter `bl`, bindings not yet retrieved with `list` or previous calls to `next_n` or `next_one`. It returns true if `bl` is a non-zero length sequence; it returns false if there are no more bindings and `bl` is a zero-length sequence.

The `how_many` parameter determines the maximum number of bindings to return in the parameter `bl`:

- A non-zero value of `how_many` guarantees that `bl` contains at most `how_many` elements. The implementation is free to return fewer than the number of bindings requested by `how_many`. However, it may not return a `bl` sequence with zero elements unless there are no bindings to retrieve.
- A zero value of `how_many` is illegal and raises a `BAD_PARAM` system exception.

`next_n` returns false with a `bl` parameter of length zero once all bindings have been retrieved. Further calls to `next_n` after it has returned a zero-length sequence have undefined behavior.

destroy

The `destroy` operation destroys its iterator. If a client invokes any operation on an iterator after calling `destroy`, the operation raises `OBJECT_NOT_EXIST`.

3.4.1 Garbage Collection of Iterators

Clients that create iterators but never call `destroy` can cause an implementation to permanently run out of resources. To protect itself against this scenario, an implementation is free to destroy an iterator object at any time without warning, using whatever algorithm it considers appropriate to choose iterators for destruction. In order to be robust in the presence of garbage collection, clients should be written to expect `OBJECT_NOT_EXIST` from calls to an iterator and handle this exception gracefully.

3.5 Stringified Names

Names are sequences of name components. This representation makes it difficult for applications to conveniently deal with names for I/O purposes, human or otherwise. This specification defines a syntax for stringified names and provides operations to convert a name in stringified form to its equivalent sequence form and vice-versa (see section 3.6.4).

A stringified name represents one and only one `CosNaming::Name`. If two names are equal, their stringified representations are equal (and vice-versa).

The stringified name representation reserves use of the characters '/', '.', and '\'. The forward slash '/' is a name component separator; the dot '.' separates `id` and `kind` fields. The backslash '\' is an escape character (see section 3.5.2).

3.5.1 Basic Representation of Stringified Names

A stringified name consists of the name components of a name separated by a '/' character. For example, a name consisting of the components "a", "b", and "c" (in that order) is represented as

```
a/b/c
```

Stringified names use the '.' character to separate `id` and `kind` fields in the stringified representation. For example, the stringified name

```
a.b/c.d/.
```

represents the `CosNaming::Name`:

Index	id	kind
0	a	b
1	c	d
2	<empty>	<empty>

The single '.' character is the only representation of a name component with empty `id` and `kind` fields.

If a name component in a stringified name does not contain a '.' character, the entire component is interpreted as the `id` field, and the `kind` field is empty. For example:

```
a/./c.d/.e
```

corresponds to the `CosNaming::Name`:

Index	id	kind
0	a	<empty>
1	<empty>	<empty>
2	c	d
3	<empty>	e

If a name component has a non-empty `id` field and an empty `kind` field, the stringified representation consists only of the `id` field. A trailing '.' character is not permitted.

3.5.2 Escape Mechanism

The backslash '\' character escapes the reserved meaning of '/', '.', and '\' in a stringified name. The meaning of any other character following a '\' is reserved for future use.

NameComponent Separators

If a name component contains a '/' slash character, the stringified representation uses the '\' character as an escape. For example, the stringified name

```
a/x\/y\/z/b
```

represents the name consisting of the name components "a", "x/y/z", and "b".

`id` and `kind` Fields

The backslash escape mechanism is also used for '.', so `id` and `kind` fields can contain a literal '.'. To illustrate, the stringified name

```
a\.b.c\.d/e.f
```

represents the `CosNaming::Name`:

Index	id	kind
0	`a.b`	`c.d`
1	`e`	`f`

The Escape Character

The escape character '\' must be escaped if it appears in a name component. For example, the stringified name:

```
a/b\\/c
```

represents the name consisting of the components "`a`", "`b\`", and "`c`".

3.6 URL schemes

This section describes the Uniform Resource Locator (URL) schemes available to represent a CORBA object and a CORBA object bound in a `NamingContext`.

3.6.1 IOR

The string form of an IOR (**IOR:<hex_octets**>) is a valid URL. The scheme name is **IOR** and the text after the ':' is defined in the CORBA 2.3 specification, Section 13.6.6. The IOR URL is robust and insulates the client from the encapsulated transport information and object key used to reference the object. This URL format is independent of Naming Service.

3.6.2 corbaloc

It is difficult for humans to exchange IORs through non-electronic means because of their length and the text encoding of binary information. The `corbaloc` URL scheme provides URLs that are familiar to people and similar to `ftp` or `http` URLs.

The `corbaloc` URL is described in the CORBA 2.3 Specification, Section 13.6.6. This URL format is independent of the Naming Service.

3.6.3 corbaname

A `corbaname` URL is similar to a `corbaloc` URL. However a `corbaname` URL also contains a stringified name that identifies a binding in a naming context.

`corbaname` Examples

```
corbaname::555xyz.com/dev/NContext1#a/b/c
```

This example denotes a naming context that can be contacted in the same manner as a `corbaloc` URL at 555xyz.com with a key of “dev/NContext1”. The “#” character denotes the start of the stringified name ,“`a/b/c` “. This name is resolved against the context to yield the final object.

```
corbaname::555xyz.com#a/b/c
```

When an object key is not specified, as in the above example, the default key of “ NameService” is used to contact the naming context.

```
corbaname:rir:#a/b/c
```

This URL will resolve the stringified name “a/b/c” against the naming context returned by resolve_initial_references(“NameService”).

```
corbaname:rir:
```

```
corbaname:rir:/NameService
```

The above URLs are equivalent to `corbaloc:rir:` and reference the naming context returned by resolve_initial_references(“ NameService”).

```
corbaname:atm:00033...#a/b/c
```

```
corbaname::55xyz.com,atm:00033.../dev/NCtext#a/b/c
```

These last URLs illustrate support of multiple protocols as allowed by `corbaloc` URLs. `atm`: is an example only and is not a defined URL protocol at this time.

Note – Unlike stringified names, `corbaname`s cannot be compared directly for equality as the address specification can differ for `corbaname` URLs with the same meaning.

`corbaname` Syntax

The full `corbaname` BNF is:

```
<corbaname>     = “corbaname:”<corbaloc_obj>[“#”<string_name>]
<corbaloc_obj>  = <obj_addr_list> [“/”<key_string>]
<obj_addr_list> = as defined in a corbaloc URL
<key_string>    = as defined in a corbaloc URL
<string_name>= stringified  Name | empty_string
```

Where:

corbaloc_obj: portion of a corbaname URL that identifies the naming context. The syntax is identical to its use in a corbaloc URL.

obj_addr_list: as defined in a corbaloc URL

key_string: as defined in a corbaloc URL.

string_name: a stringified Name with URL escapes as defined below.

`corbaname` Character Escapes

`corbaname` URLs use the escape mechanism described in the Internet Engineering Task Force (IETF) RFC 2396. These escape rules insure that URLs can be transferred via a variety of transports without undergoing changes. The character escape rules for the stringified name portion of an `corbaname` are:

US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

";" | "/" | ":" | "?" | "@" | "&" | "=" | "+" | "$" |

"," | "-" | "_" | "." | "!" | "~" | "*" | "'" | "(" | ")"

`corbaname` Escape Mechanism

The percent '%' character is used as an escape. If a character that requires escaping is present in a name component it is encoded as two hexadecimal digits following a "%" character to represent the octet. (The first hexadecimal character represent the high-order nibble of the octet, the second hexadecimal character represents the low-order nibble.) If a '%' is not followed by two hex digits, the stringified name is syntactically invalid.

Examples

Table 3-1

Stringified Name	After URL Escapes	Comment
a.b/c.d	a.b/c.d	URL form identical
<a>.b/c.d	%3ca%3e.b/c.d	Escaped "<" and ">"
a.b/ c.d	a.b/%20%20c.d	Escaped two " " spaces
a%b/c%d	a%25b/c%25d	Escaped two "%" percents
a\\b/c.d	a%5c%5c/c.d	Escaped "\" character, which is already escaped in the stringified name

`corbaname` Resolution

`corbaname` resolution can be implemented as a simple extension to `corbaloc` URL processing. Given a `corbaname`:

```
corbaname:<corbaloc_obj>["#" <string_name>]
```

The `corbaname` is resolved by:

1. First constructing an `corbaloc` URL of the form: `corbaloc:<corabloc_obj>`.

   If the <corbaloc_obj> does not contain a key string, a default key of "NameService" is used.

2. This is converted to a naming context object reference with `CORBA::ORB::string_to_object`.
3. The `<string_name>` is converted to a `CosNaming::Name`.
4. The resulting name is passed to a `resolve` operation on the naming context.
5. The object reference returned by the `resolve` is the result.

Implementations are not required to use the method described and may make optimizations appropriate to their environment.

3.6.4 Converting between CosNames, Stringified Names, and URLs

The `NamingContextExt` interface, derived from `NamingContext`, provides the operations required to use URLs and stringified names.

```
module CosNaming {
    // ...
    interface NamingContextExt: NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

        StringName     to_string(in Name n) raises(InvalidName);
        Name           to_name(in StringName sn)
                           raises(InvalidName);

        exception InvalidAddress {};

        URLString      to_url(in Address addrkey, in StringName sn)
                           raises(InvalidAddress, InvalidName);

        Object         resolve_str(in StringName n)
                           raises(
                               NotFound, CannotProceed,
                               InvalidName
                           );
    };
};
```

to_string

This operation accepts a `Name` and returns a stringified name. If the `Name` is invalid, an `InvalidName` exception is raised.

to_name

This operation accepts a stringified name and returns a `Name`. If the stringified name is syntactically malformed or violates an implementation limit, an `InvalidName` exception is raised.

resolve_str

This is a convenience operation that performs a resolve in the same manner as `NamingContext::resolve`. It accepts a stringified name as an argument instead of a `Name`.

to_url

This operation takes a corbaloc URL `<address>` and <key_string> component such as

- `:myhost.555xyz.com`
- `:myhost.555xyz.com/a/b/c`
- `atm:00002112...,:myhost.xyz.com/a/b/c`

for the first parameter, and a stringified name for the second. It then performs any escapes necessary on the parameters and returns a fully formed URL string. An exception is raised if either the corbaloc address and key parameter or name parameter are malformed.

It is legal for the stringified_name to be empty. If the address is empty, an InvalidAddress exception is raised.

URL to Object Reference

Conversions from URLs to objects are handled by `CORBA::ORB::string_to_object` as described in the CORBA 2.3 Specification, Section 13.6.6.

3.7 Initial Reference to a NamingContextExt

An initial reference to an instance of this interface can be obtained by calling `resolve_initial_references` with an `ObjectID` of `NameService`.

3.8 Conformance Requirements

3.8.1 Optional Interfaces

There are no optional interfaces in this specification. A compliant implementation must implement all of the functionality and interfaces described.

3.8.2 Documentation Requirements

A compliant implementation must document all of the following:

- any limitations to the character values or character sequences that may be used in a name component
- any limitations to the length (including minimum or maximum) of a name component
- any limitations to number of name components in a name
- any limitations to the maximum number of bindings in a context
- any limitations to the total number of bindings (implementation-wide)
- any limitations to the maximum number of contexts
- the means provided to deal with orphaned contexts and bindings
- Any policy for dealing with potentially orphaned naming contexts. Orphaned contexts are contexts that are not bound in any other context within a naming server.
- Any policy for destroying binding iterators that are considered to be no longer in use.
- Under what circumstances, if any, a `CannotProceed` exception is raised.

ORB Interface 4

Note – This chapter is the CORBA 2.3 Specification Chapter 4 with a new Section 4.8, "Configuring Initial Service References". The new section is in blue and marked with changebars. Changebars outside of 4.8 are *not* for the Interoperable Naming Service submission.

The ORB Interface chapter has been updated based on the CORE changes from (ptc/98-09-04) and the Objects by Value documents (ptc/98-07-06) and (orbos/98-01-18). Changes from RTF 2.4 (ptc/99-03-01) and policy management related material from the Messaging specification (orbos/98-05-05) have also been incorporated.

Contents

This chapter contains the following sections.

Section Title	Page
"Policy Object"	4-47
"Management of Policy Domains"	4-54
"Thread-Related Operations"	4-60

4.1 Overview

This chapter introduces the operations that are implemented by the ORB core, and describes some basic ones, while providing reference to the description of the remaining operations that are described elsewhere. The ORB interface is the interface to those ORB functions that do not depend on which object adapter is used. These operations are the same for all ORBs and all object implementations, and can be performed either by clients of the objects or implementations. The Object interface contains operations that are implemented by the ORB, and are accessed as implicit operations of the Object Reference. The ValueBase interface contains operations that are implemented by the ORB, and are accessed as implicit operations of the ValueBase Reference.

Because the operations in this section are implemented by the ORB itself, they are not in fact operations on objects, although they are described that way for the Object or ValueBase interface operations and the language binding will, for consistency, make them appear that way.

4.2 The ORB Operations

The ORB interface contains the operations that are available to both clients and servers. These operations do not depend on any specific object adapter or any specific object reference.

```
module CORBA {

    interface NVList;           // forward declaration
    interface OperationDef;     // forward declaration
    interface TypeCode;         // forward declaration

    typedef short PolicyErrorCode;
    // for the definition of consts see "PolicyErrorCode" on page 4-49

    interface Request;                  // forward declaration
    typedef sequence <Request> RequestSeq;

    native AbstractBase;

    exception PolicyError {PolicyErrorCode reason;};

    typedef string RepositoryId;
```

```
        typedef string Identifier;

        // StructMemberSeq defined in Chapter 10
        // UnionMemberSeq defined in Chapter 10
        // EnumMemberSeq defined in Chapter 10

        typedef unsigned short ServiceType;
        typedef unsigned long ServiceOption;
        typedef unsigned long ServiceDetailType;

        const ServiceType Security = 1;

        struct ServiceDetail {
            ServiceDetailType service_detail_type;
            sequence <octet> service_detail;
        };

        struct ServiceInformation {
            sequence <ServiceOption> service_options;
            sequence <ServiceDetail> service_details;
        };

        native ValueFactory;

        interface ORB {                                          // PIDL
    #pragma version ORB 2.3

            typedef string ObjectId;
            typedef sequence <ObjectId> ObjectIdList;

            exception InvalidName {};

            string object_to_string (
                in Object           obj
            );

            Object string_to_object (
                in string           str
            );

            // Dynamic Invocation related operations

            void create_list (
                in long             count,
                out NVList          new_list
            );

            void create_operation_list (
                in OperationDef     oper,
                out NVList          new_list
            );
```

```
void get_default_context (
    out Context          ctx
);

void send_multiple_requests_oneway(
    in RequestSeq      req
);

void send_multiple_requests_deferred(
    in RequestSeq      req
);

boolean poll_next_response();

void get_next_response(
    out Request          req
);

// Service information operations

boolean get_service_information (
    in ServiceType service_type,
    out ServiceInformation service_information
);

ObjectIdList list_initial_services ();

// Initial reference operation

Object resolve_initial_references (
    in ObjectId identifier
) raises (InvalidName);

// Type code creation operations

TypeCode create_struct_tc (
    in RepositoryId id,
    in Identifier name,
    in StructMemberSeq members
);

TypeCode create_union_tc (
    in RepositoryId id,
    in Identifier name,
    in TypeCode discriminator_type,
    in UnionMemberSeq members
);

TypeCode create_enum_tc (
    in RepositoryId id,
    in Identifier name,
```

```
        in EnumMemberSeq members
);

TypeCode create_alias_tc (
        in RepositoryId id,
        in Identifier name,
        in TypeCode original_type
);

TypeCode create_exception_tc (
        in RepositoryId id,
        in Identifier name,
        in StructMemberSeq members
);

TypeCode create_interface_tc (
        in RepositoryId id,
        in Identifier name
);

TypeCode create_string_tc (
        in unsigned long bound
);

TypeCode create_wstring_tc (
        in unsigned long bound
);

TypeCode create_fixed_tc (
        in unsigned short digits,
        in short scale
);

TypeCode create_sequence_tc (
        in unsigned long bound,
        in TypeCode element type
);

TypeCode create_recursive_sequence_tc // deprecated
        in unsigned long bound,
        in unsigned long offset
);

TypeCode create_array_tc (
        in unsigned long length,
        in TypeCode element_type
);

TypeCode create_value_tc (
        in RepositoryId         id,
        in Identifier           name,
```

```
        in ValueModifier        type_modifier,
        in TypeCode             concrete_base,
        in ValueMembersSeq      members
    );

    TypeCode create_value_box_tc (
        in RepositoryId         id,
        in Identifier           name,
        in TypeCode             boxed_type
    );

    TypeCode create_native_tc (
        in RepositoryId         id,
        in Identifier           name
    );

    TypeCode create_recursive_tc(
        in RepositoryId         id
    );

    TypeCode create_abstract_interface_tc(
        in RepositoryId         id,
        in Identifier           name
    );

    // Thread related operations

    boolean work_pending( );

    void perform_work();

    void run();

    void shutdown(
        in boolean              wait_for_completion
    );

    void destroy();

    // Policy related operations

    Policy create_policy(
        in PolicyType       type,
        in any              val
    ) raises (PolicyError);

    // Dynamic Any related operations deprecated and removed
    // from primary list of ORB operations

    // Value factory operations
```

```
        ValueFactory register_value_factory(
            in RepositoryId id,
            in ValueFactory factory
        );

        void unregister_value_factory(in RepositoryId id);

        ValueFactory lookup_value_factory(in RepositoryId id);
    };
};
```

All types defined in this chapter are part of the CORBA module. When referenced in OMG IDL, the type names must be prefixed by "**CORBA::**".

The operations **object_to_string** and **string_to_object** are described in "Converting Object References to Strings" on page 4-31.

For a description of the **create_list** and **create_operation_list** operations, see Section 7.4, "List Operations," on page 7-10. The **get_default_context** operation is described in the section Section 7.6.1, "get_default_context," on page 7-14. The **send_multiple_requests_oneway** and **send_multiple_requests_deferred** operations are described in the section Section 7.3.2, "send_multiple_requests," on page 7-9. The **poll_next_response** and **get_next_response** operations are described in the section Section 7.3.5, "get_next_response," on page 7-10.

The **list_intial_services** and **resolve_initial_references** operations are described in "Obtaining Initial Object References" on page 4-42.

The Type code creation operations with names of the form **create_<type>_tc** are described in Section 10.7.3, "Creating TypeCodes," on page 10-53.

The **work_pending**, **perform_work**, **shutdown**, **destroy** and **run** operations are described in "Thread-Related Operations" on page 4-60.

The **create_policy** operations is described in "Create_policy" on page 4-50.

The **register_value_factory**, **unregister_value_factory** and **lookup_value_factory** operations are described in Section 5.4.3, "Language Specific Value Factory Requirements," on page 5-9.

4.2.1 Converting Object References to Strings

4.2.1.1 object_to_string

```
string object_to_string (
    in Object           obj
);
```

4.2.1.2 string_to_object

```
Object string_to_object (
    in string               str
);
```

Because an object reference is opaque and may differ from ORB to ORB, the object reference itself is not a convenient value for storing references to objects in persistent storage or communicating references by means other than invocation. Two problems must be solved: allowing an object reference to be turned into a value that a client can store in some other medium, and ensuring that the value can subsequently be turned into the appropriate object reference.

An object reference may be translated into a string by the operation **object_to_string**. The value may be stored or communicated in whatever ways strings may be manipulated. Subsequently, the **string_to_object** operation will accept a string produced by **object_to_string** and return the corresponding object reference.

To guarantee that an ORB will understand the string form of an object reference, that ORB's **object_to_string** operation must be used to produce the string. For all conforming ORBs, if **obj** is a valid reference to an object, then **string_to_object(object_to_string(obj))** will return a valid reference to the same object, if the two operations are performed on the same ORB. For all conforming ORB's supporting IOP, this remains true even if the two operations are performed on different ORBs.

4.2.2 Getting Service Information

4.2.2.1 get_service_information

```
boolean get_service_information (
    in ServiceType service_type;
    out ServiceInformation service_information;
);
```

The **get_service_information** operation is used to obtain information about CORBA facilities and services that are supported by this ORB. The service type for which information is being requested is passed in as the in parameter **service_type**, the values defined by constants in the CORBA module. If service information is available for that type, that is returned in the out parameter **service_information**, and the operation returns the value TRUE. If no information for the requested services type is available, the operation returns FALSE (i.e., the service is not supported by this ORB).

4.3 Object Reference Operations

There are some operations that can be done on any object. These are not operations in the normal sense, in that they are implemented directly by the ORB, not passed on to the object implementation. We will describe these as being operations on the object

reference, although the interfaces actually depend on the language binding. As above, where we used interface **Object** to represent the object reference, we define an interface for **Object**:

```
module CORBA {

    interface DomainManager;          // forward declaration
    typedef sequence <DomainManager> DomainManagersList;

    interface Policy;                 // forward declaration
    typedef sequence <Policy> PolicyList;
    typedef unsigned long PolicyType;

    interface Context;                // forward declaration

    typedef string Identifier;
    interface Request;                // forward declaration
    interface NVList;                 // forward declaration
    struct NamedValue{};              // an implicitly well known type
    typedef unsigned long Flags;
    interface InterfaceDef;

    enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

    interface Object {                                    // PIDL

        InterfaceDef get_interface ();

        boolean is_nil();

        Object duplicate ();

        void release ();

        boolean is_a (
            in string             logical_type_id
        );

        boolean non_existent();

        boolean is_equivalent (
            in Object             other_object
        );

        unsigned long hash(
            in unsigned long      maximum
        );

        void create_request (
            in Context            ctx
```

```
            in Identifier           operation,
            in NVList               arg_list,
            inout NamedValue        result,
            out Request             request,
            in Flags                req_flag
        );

        Policy get_policy (
            in PolicyType           policy_type
        );

        DomainManagersList get_domain_managers ();

        Object set_policy_overrides(
            in PolicyList           policies,
            in SetOverrideType      set_add
        );
    };
};
```

The **create_request** operation is part of the Object interface because it creates a pseudo-object (a Request) for an object. It is described with the other Request operations in the section Section 7.2, “Request Operations,” on page 7-4.

Unless otherwise stated below, the operations in the IDL above do not require access to remote information.

4.3.1 Determining the Object Interface

4.3.1.1 get_interface

```
InterfaceDef get_interface();
```

An operation on the object reference, **get_interface**, returns an object in the Interface Repository, which provides type information that may be useful to a program. See the Interface Repository chapter for a definition of operations on the Interface Repository. The implementation of this operation may involve contacting the ORB that implements the target object.

4.3.2 Duplicating and Releasing Copies of Object References

4.3.2.1 duplicate

```
Object duplicate();
```

4.3.2.2 release

```
void release();
```

Because object references are opaque and ORB-dependent, it is not possible for clients or implementations to allocate storage for them. Therefore, there are operations defined to copy or release an object reference.

If more than one copy of an object reference is needed, the client may create a duplicate. Note that the object implementation is not involved in creating the duplicate, and that the implementation cannot distinguish whether the original or a duplicate was used in a particular request.

When an object reference is no longer needed by a program, its storage may be reclaimed by use of the **release** operation. Note that the object implementation is not involved, and that neither the object itself nor any other references to it are affected by the **release** operation.

4.3.3 Nil Object References

4.3.3.1 is_nil

```
boolean is_nil();
```

An object reference whose value is **OBJECT_NIL** denotes no object. An object reference can be tested for this value by the **is_nil** operation. The object implementation is not involved in the nil test.

4.3.4 Equivalence Checking Operation

4.3.4.1 is_a

```
boolean is_a(
    in RepositoryId         logical_type_id
);
```

An operation is defined to facilitate maintaining type-safety for object references over the scope of an ORB.

The **logical_type_id** is a string denoting a shared type identifier (**RepositoryId**). The operation returns true if the object is really an instance of that type, including if that type is an ancestor of the “most derived” type of that object.

Determining whether an object's type is compatible with the **logical_type_id** may require contacting a remote ORB or interface repository. Such an attempt may fail at either the local or the remote end. If **is_a** cannot make a reliable determination of type compatibility due to failure, it raises an exception in the calling application code. This enables the application to distinguish among the **TRUE**, **FALSE**, and indeterminate cases.

This operation exposes to application programmers functionality that must already exist in ORBs which support “type safe narrow” and allows programmers working in environments that do not have compile time type checking to explicitly maintain type safety.

4.3.5 Probing for Object Non-Existence

4.3.5.1 non_existent

```
boolean non_existent ();
```

The **non_existent** operation may be used to test whether an object (e.g., a proxy object) has been destroyed. It does this without invoking any application level operation on the object, and so will never affect the object itself. It returns true (rather than raising CORBA::OBJECT_NOT_EXIST) if the ORB knows authoritatively that the object does not exist; otherwise, it returns false.

Services that maintain state that includes object references, such as bridges, event channels, and base relationship services, might use this operation in their “idle time” to sift through object tables for objects that no longer exist, deleting them as they go, as a form of garbage collection. In the case of proxies, this kind of activity can cascade, such that cleaning up one table allows others then to be cleaned up.

Probing for object non-existence may require contacting the ORB that implements the target object. Such an attempt may fail at either the local or the remote end. If non-existent cannot make a reliable determination of object existence due to failure, it raises an exception in the calling application code. This enables the application to distinguish among the true, false, and indeterminate cases.

4.3.6 Object Reference Identity

In order to efficiently manage state that include large numbers of object references, services need to support a notion of object reference identity. Such services include not just bridges, but relationship services and other layered facilities.

Two identity-related operations are provided. One maps object references into disjoint groups of potentially equivalent references, and the other supports more expensive pairwise equivalence testing. Together, these operations support efficient maintenance and search of tables keyed by object references.

4.3.6.1 Hashing Object Identifiers

hash

```
unsigned long hash(
    in unsigned long        maximum
);
```

Object references are associated with ORB-internal identifiers which may indirectly be accessed by applications using the **hash** operation. The value of this identifier does not change during the lifetime of the object reference, and so neither will any hash function of that identifier.

The value of this operation is not guaranteed to be unique; that is, another object reference may return the same hash value. However, if two object references hash differently, applications can determine that the two object references are *not* identical.

The **maximum** parameter to the **hash** operation specifies an upper bound on the hash value returned by the ORB. The lower bound of that value is zero. Since a typical use of this feature is to construct and access a collision chained hash table of object references, the more randomly distributed the values are within that range, and the cheaper those values are to compute, the better.

For bridge construction, note that proxy objects are themselves objects, so there could be many proxy objects representing a given "real" object. Those proxies would not necessarily hash to the same value.

4.3.6.2 *Equivalence Testing*

is_equivalent

```
boolean is_equivalent(
    in Object               other_object
);
```

The **is_equivalent** operation is used to determine if two object references are equivalent, so far as the ORB can easily determine. It returns TRUE if the target object reference is known to be equivalent to the other object reference passed as its parameter, and FALSE otherwise.

If two object references are identical, they are equivalent. Two different object references which in fact refer to the same object are also equivalent.

ORBs are allowed, but not required, to attempt determination of whether two distinct object references refer to the same object. In general, the existence of reference translation and encapsulation, in the absence of an omniscient topology service, can make such determination impractically expensive. This means that a FALSE return from **is_equivalent** should be viewed as only indicating that the object references are distinct, and not necessarily an indication that the references indicate distinct objects.

A typical application use of this operation is to match object references in a hash table. Bridges could use it to shorten the lengths of chains of proxy object references. Externalization services could use it to "flatten" graphs that represent cyclical relationships between objects. Some might do this as they construct the table, others during idle time.

4.3.7 Getting Policy Associated with the Object

4.3.7.1 get_policy

The **get_policy** operation returns the policy object of the specified type (see "Policy Object" on page 4-47), which applies to this object. It returns the *effective* **Policy** for the object reference. The effective **Policy** is the one that would be used if a request were made. This **Policy** is determined first by obtaining the *effective override* for the **PolicyType** as returned by **get_client_policy**. The effective override is then compared with the **Policy** as specified in the IOR. The effective **Policy** is the intersection of the values allowed by the effective override and the IOR-specified **Policy**. If the intersection is empty, the system exception INV_POLICY is raised. Otherwise, a **Policy** with a value legally within the intersection is returned as the effective **Policy**. The absence of a **Policy** value in the IOR implies that any legal value may be used. Invoking **non_existent** on an object reference prior to **get_policy** ensures the accuracy of the returned effective **Policy**. If **get_policy** is invoked prior to the object reference being bound, the returned effective **Policy** is implementation dependent. In that situation, a compliant implementation may do any of the following: raise the system exception BAD_INV_ORDER, return some value for that **PolicyType** which may be subject to change once a binding is performed, or attempt a binding and then return the effective **Policy**. Note that if the effective **Policy** may change from invocation to invocation due to transparent rebinding.

```
Policy get_policy (
    in PolicyType       policy_type
);
```

Parameter(s)

policy_type - The type of policy to be obtained.

Return Value

A **Policy** object of the type specified by the **policy_type** parameter.

Exception(s)

CORBA::INV_POLICY - raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

The implementation of this operation may involve remote invocation of an operation (e.g. **DomainManager::get_domain_policy** for some security policies) for some policy types.

4.3.8 Overriding Associated Policies on an Object Reference

4.3.8.1 set_policy_overrides

The **set_policy_overrides** operation returns a new object reference with the new policies associated with it. It takes two input parameters. The first parameter **policies** is a sequence of references to **Policy** objects. The second parameter **set_add** of type **SetOverrideType** indicates whether these policies should be added onto any other overrides that already exist (**ADD_OVERRIDE**) in the object reference, or they should be added to a clean override free object reference (**SET_OVERRIDE**). This operation associates the policies passed in the first parameter with a newly created object reference that it returns. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception.

```
enum SetOverrideType {SET_OVERRIDE, ADD_OVERRIDE};

Object set_policy_overrides(
    in PolicyList           policies,
    in SetOverrideType      set_add
);
```

Parameter(s)

policies - a sequence of **Policy** objects that are to be associated with the new copy of the object reference returned by this operation

set_add - whether the association is in addition to (**ADD_OVERRIDE**) or as replacement of (**SET_OVERRIDE**) any existing overrides already associated with the object reference.

Return Value

A copy of the object reference with the overrides from **policies** associated with it in accordance with the value of **set_add**.

Exception(s)

CORBA::NO_PERMISSION - raised when an attempt is made to override any policy that cannot be overridden.

4.3.9 Getting the Domain Managers Associated with the Object

4.3.9.1 get_domain_managers

The get_domain_managers operation allows administration services (and applications) to retrieve the domain managers (see "Management of Policy Domains" on page 4-54), and hence the security and other policies applicable to individual objects that are members of the domain.

```
typedef sequence <DomainManager> DomainManagersList;

DomainManagersList get_domain_managers ();
```

Return Value

The list of immediately enclosing domain managers of this object. At least one domain manager is always returned in the list since by default each object is associated with at least one domain manager at creation.

The implementation of this operation may involve contacting the ORB that implements the target object.

4.4 ValueBase Operations

ValueBase serves a similar role for value types that **Object** serves for interfaces. Its mapping is language-specific and must be explicitly specified for each language.

Typically it is mapped to a concrete language type which serves as a base for all value types. Any operations that are required to be supported for all values are conceptually defined on **ValueBase**, although in reality their actual mapping depends upon the specifics of any particular language mapping.

Analogous to the definition of the **Object** interface for implicit operations of object references, the implicit operations of **ValueBase** are defined on a pseudo-**valuetype** as follows:

```
module CORBA {
    valuetype ValueBase{                                    PIDL
        ValueDef get_value_def();
    };
};
```

The **get_value_def()** operation returns a description of the value's definition as described in the interface repository (Section 10.5.24, "ValueDef," on page 10-34).

4.5 ORB and OA Initialization and Initial References

Before an application can enter the CORBA environment, it must first:

- Be initialized into the ORB and possibly the object adapter (POA) environments.
- Get references to ORB pseudo-object (for use in future ORB operations) and perhaps other objects (including the root POA or some Object Adapter objects).

The following operations are provided to initialize applications and obtain the appropriate object references:

- Operations providing access to the ORB. These operations reside in the CORBA module, but not in the ORB interface and are described in Section 4.6, "ORB Initialization," on page 4-41.

- Operations providing access to Object Adapters, Interface Repository, Naming Service, and other Object Services. These operations reside in the ORB interface and are described in Section 4.7, "Obtaining Initial Object References," on page 4-42.

4.6 ORB Initialization

When an application requires a CORBA environment it needs a mechanism to get the ORB pseudo-object reference and possibly an OA object reference (such as the root POA). This serves two purposes. First, it initializes an application into the ORB and OA environments. Second, it returns the ORB pseudo-object reference and the OA object reference to the application for use in future ORB and OA operations.

The ORB and OA initialization operations must be ordered with ORB occurring before OA: an application cannot call OA initialization routines until ORB initialization routines have been called for the given ORB. The operation to initialize an application in the ORB and get its pseudo-object reference is not performed on an object. This is because applications do not initially have an object on which to invoke operations. The ORB initialization operation is an application's bootstrap call into the CORBA world. The **ORB_init** call is part of the CORBA module but not part of the ORB interface.

Applications can be initialized in one or more ORBs. When an ORB initialization is complete, its pseudo reference is returned and can be used to obtain other references for that ORB.

In order to obtain an ORB pseudo-object reference, applications call the **ORB_init** operation. The parameters to the call comprise an identifier for the ORB for which the pseudo-object reference is required, and an **arg_list**, which is used to allow environment-specific data to be passed into the call. PIDL for the ORB initialization is as follows:

```
// PIDL
module CORBA {
        typedef string ORBid;
        typedef sequence <string> arg_list;
        ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

The identifier for the ORB will be a name of type **CORBA::ORBid**. All **ORBid** strings other than the empty string are allocated by ORB administrators and are not managed by the OMG. **ORBid** strings other than the empty string are intended to be used to uniquely identify each ORB used within the same address space in a multi-ORB application. These special **ORBid** strings are specific to each ORB implementation and the ORB administrator is responsible for ensuring that the names are unambiguous.

If an empty **ORBid** string is passed to **ORB_init**, then the **arg_list** arguments shall be examined to determine if they indicate an ORB reference that should be returned. This is achieved by searching the **arg_list** parameters for one preceded by "**-ORBid**" for example, "**-ORBid example_orb**" (the white space after the "**-ORBid**" tag is ignored) or "**-ORBidMyFavoriteORB**" (with no white space following the "**-ORBid**"

tag). Alternatively, two sequential parameters with the first being the string "**-ORBid**" indicates that the second is to be treated as an **ORBid** parameter. If an empty string is passed and no **arg_list** parameters indicate the ORB reference to be returned, the default ORB for the environment will be returned.

Other parameters of significance to the ORB can also be identified in **arg_list**, for example, "**Hostname**," "**SpawnedServer**," and so forth. To allow for other parameters to be specified without causing applications to be re-written, it is necessary to specify the parameter format that ORB parameters may take. In general, parameters shall be formatted as either one single **arg_list** parameter:

–ORB<suffix><optional white space> <value>

or as two sequential **arg_list** parameters:

-ORB<suffix>

<value>

Regardless of whether an empty or non-empty **ORBid** string is passed to **ORB_init**, the **arg_list** arguments are examined to determine if any ORB parameters are given. If a non-empty **ORBid** string is passed to **ORB_init**, all **ORBid** parameters in the **arg_list** are ignored. All other **-ORB<suffix>** parameters in the **arg_list** may be of significance during the ORB initialization process.

Before **ORB_init** returns, it will remove from the **arg_list** parameter all strings that match the **-ORB<suffix>** pattern described above and that are recognized by that ORB implementation, along with any associated sequential parameter strings. If any strings in **arg_list** that match this pattern are not recognized by the ORB implementation, **ORB_init** will raise the BAD_PARAM system exception instead.

The **ORB_init** operation may be called any number of times and shall return the same ORB reference when the same **ORBid** string is passed, either explicitly as an argument to **ORB_init** or through the **arg_list**. All other **-ORB<suffix>** parameters in the **arg_list** may be considered on subsequent calls to **ORB_init**.

4.7 Obtaining Initial Object References

Applications require a portable means by which to obtain their initial object references. References are required for the root POA, POA Current, Interface Repository and various Object Services instances. (The POA is described in the Portable Object Adaptor chapter; the Interface Repository is described in the Interface Repository chapter; Object Services are described in *CORBAservices: Common Object Services Specification*.) The functionality required by the application is similar to that provided by the Naming Service. However, the OMG does not want to mandate that the Naming Service be made available to all applications in order that they may be portably initialized. Consequently, the operations shown in this section provide a simplified, local version of the Naming Service that applications can use to obtain a small, defined set of object references which are essential to its operation. Because only a small well-

defined set of objects are expected with this mechanism, the naming context can be flattened to be a single-level name space. This simplification results in only two operations being defined to achieve the functionality required.

Initial references are not obtained via a new interface; instead two operations are provided in the ORB pseudo-object interface, providing facilities to list and resolve initial object references.

list_initial_services

```
typedef string ObjectId;
typedef sequence <ObjectId> ObjectIdList;
ObjectIdList list_initial_services ();
```

resolve_initial_references

```
exception InvalidName {};

Object resolve_initial_references (
    in ObjectId identifier
) raises (InvalidName);
```

The **resolve_initial_references** operation is an operation on the ORB rather than the Naming Service's **NamingContext**. The interface differs from the Naming Service's resolve in that **ObjectId** (a string) replaces the more complex Naming Service construct (a sequence of structures containing string pairs for the components of the name). This simplification reduces the name space to one context.

ObjectIds are strings that identify the object whose reference is required. To maintain the simplicity of the interface for obtaining initial references, only a limited set of objects are expected to have their references found via this route. Unlike the ORB identifiers, the **ObjectId** name space requires careful management. To achieve this, the OMG may, in the future, define which services are required by applications through this interface and specify names for those services.

Currently, reserved **ObjectIds** are **RootPOA**, **POACurrent**, **InterfaceRepository,** **NameService**, **TradingService**, **SecurityCurrent**, **TransactionCurrent,** and **DynAnyFactory.**

To allow an application to determine which objects have references available via the initial references mechanism, the **list_initial_services** operation (also a call on the ORB) is provided. It returns an **ObjectIdList,** which is a sequence of **ObjectIds**. **ObjectIds** are typed as strings. Each object, which may need to be made available at initialization time, is allocated a string value to represent it. In addition to defining the id, the type of object being returned must be defined (i.e., "**InterfaceRepository**" returns an object of type **Repository**, and "**NameService**" returns a **CosNamingContext** object).

The application is responsible for narrowing the object reference returned from **resolve_initial_references** to the type which was requested in the ObjectId. For example, for InterfaceRepository the object returned would be narrowed to **Repository** type.

In the future, specifications for Object Services (in *CORBAservices: Common Object Services Specification*) will state whether it is expected that a service's initial reference be made available via the **resolve_initial_references** operation or not (i.e., whether the service is necessary or desirable for bootstrap purposes).

4.8 Configuring Initial Service References

4.8.1 ORB-specific Configuration

It is required that an ORB can be administratively configured to return an arbitrary object reference from `CORBA::ORB::resolve_initial_references` for non-locality-constrained objects.

In addition to this required implementation-specific configuration, two `CORBA::ORB_init` arguments are provided to override the ORB initial reference configuration.

4.8.2 ORBInitRef

The ORB initial reference argument, `-ORBInitRef`, allows specification of an arbitrary object reference for an initial service. The format is:

-ORBInitRef <ObjectID>=<ObjectURL>

Examples of use are:

-ORBInitRef NameService=IOR:00230021AB...

-ORBInitRef NotificationService=corbaloc::555objs.com/NotificationService

-ORBInitRef TradingService=corbaname::555objs.com/Dev/Trader

`<ObjectID>` represents the well-known `ObjectID` for a service defined in the CORBA specification, such as `NameService`. This mechanism allows an ORB to be configured with new initial service Object IDs that were not defined when the ORB was installed.

`<ObjectURL>` can be any of the URL schemes supported by `CORBA::ORB::string_to_object` (Sections 13.6.6 to 13.6.7 CORBA 2.3 Specification). If a URL is syntactically malformed or can be determined to be invalid in an implementation defined manner, `ORB_init` raises a `BAD_PARAM` exception.

4.8.3 ORBDefaultInitRef

The ORB default initial reference argument, `-ORBDefaultInitRef`, assists in resolution of initial references not explicitly specified with `-ORBInitRef`. `-ORBDefaultInitRef` requires a URL that, after appending a slash '/' character and a stringified object key, forms a new URL to identify an initial object reference. For example:

-ORBDefaultInitRef corbaloc::555objs.com

A call to `resolve_initial_references("NotificationService")` with this argument results in a new URL:

corbaloc::555objs.com/NotificationService

That URL is passed to `CORBA::ORB::string_to_object` to obtain the initial reference for the service.

Another example is:

-ORBDefaultInitRef corbaname::555ResolveRefs.com,:555Backup.com/Prod/Local

After calling `resolve_initial_references("NameService")`, one of the `corbaname` URLs

```
corbaname::555ResolveRefs.com/Prod/Local/NameService
```

or

```
corbaname::555Backup411.com/Prod/Local/NameService
```

is used to obtain an object reference from `string_to_object`. (In this example, `Prod/Local/NameService` represents a stringified `CosNaming::Name`).

Section 13.6.7 provides details of the `corbaloc` and `corbaname` URL schemes. The `-ORBDefaultInitRef` argument naturally extends to URL schemes that may be defined in the future, provided the final part of the URL is an object key.

4.8.4 Configuration Effect on `resolve_initial_references`

4.8.4.1 Default Resolution Order

The default order for processing a call to `CORBA::ORB::resolve_initial_references` for a given `<ObjectID>` is:

1. Resolve with `-ORBInitRef` for this `<ObjectID>` if possible
2. Resolve with an `-ORBDefaultInitRef` entry if possible
3. Resolve with pre-configured ORB settings.

4.8.4.2 ORB Configured Resolution Order

There are cases where the default resolution order may not be appropriate for all services and use of `-ORBDefaultInitRef` may have unintended resolution side effects. For example, an ORB may use a proprietary service, such as `ImplementationRepository`, for internal purposes and may want to prevent a client from unknowingly diverting the ORB's reference to an implementation repository from another vendor. To prevent this, an ORB is allowed to ignore the `-ORBDefaultInitRef` argument for any or all `<ObjectID>`s for those services that are not OMG-specified services with a well-known service name as accepted by `resolve_initial_references`. An ORB can only ignore the `-ORBDefaultInitRef` argument but must always honor the `-ORBInitRef` argument.

4.8.5 Configuration Effect on `list_initial_services`

The `<ObjectID>`s of all `-ORBInitRef` arguments to `ORB_init` appear in the list of tokens returned by `list_initial_services` as well as all ORB-configured `<ObjectID>`s. Any other tokens that may appear are implementation-dependent.

The list of `<ObjectID>`s returned by `list_initial_services` can be a subset of the `<ObjectID>`s recognized as valid by `resolve_initial_reference`s.

4.9 Current Object

ORB and CORBA services may wish to provide access to information (context) associated with the thread of execution in which they are running. This information is accessed in a structured manner using interfaces derived from the **Current** interface defined in the CORBA module.

Each ORB or CORBA service that needs its own context derives an interface from the CORBA module's **Current**. Users of the service can obtain an instance of the appropriate **Current** interface by invoking **ORB::resolve_initial_references**. For example the Security service obtains the **Current** relevant to it by invoking

ORB::resolve_initial_references("SecurityCurrent")

A CORBA service does not have to use this method of keeping context but may choose to do so.

```
module CORBA {
    // interface for the Current object
    interface Current {
    };
};
```

Operations on interfaces derived from **Current** access state associated with the thread in which they are invoked, not state associated with the thread from which the **Current** was obtained. This prevents one thread from manipulating another thread's state, and avoids the need to obtain and narrow a new **Current** in each method's thread context.

Current objects must not be exported to other processes, or externalized with **ORB::object_to_string**. If any attempt is made to do so, the offending operation will raise a MARSHAL system exception. **Current**s are per-process singleton objects, so no destroy operation is needed.

4.10 Policy Object

4.10.1 Definition of Policy Object

An ORB or CORBA service may choose to allow access to certain choices that affect its operation. This information is accessed in a structured manner using interfaces derived from the **Policy** interface defined in the CORBA module. A CORBA service does not have to use this method of accessing operating options, but may choose to do so. The *Security Service* in particular uses this technique for associating *Security Policy* with objects in the system.

```
module CORBA {
    typedef unsigned long PolicyType;

    // Basic IDL definition
    interface Policy {
        readonly attribute PolicyType policy_type;
        Policy copy();
        void destroy();
    };

    typedef sequence <Policy> PolicyList;
};
```

PolicyType defines the type of **Policy** object. In general the constant values that are allocated are defined in conjunction with the definition of the corresponding **Policy** object. The values of **PolicyTypes** for policies that are standardized by OMG are allocated by OMG. Additionally, vendors may reserve blocks of 4096 PolicyType values identified by a 20 bit *Vendor PolicyType Valueset ID* (**VPVID**) for their own use.

PolicyType which is an unsigned long consists of the 20-bit **VPVID** in the high order 20 bits, and the vendor assigned policy value in the low order 12 bits. The **VPVID**s *0 through \xf* are reserved for OMG. All values for the standard **PolicyTypes** are allocated within this range by OMG. Additionally, the **VPVID**s *\xfffff* is reserved for experimental use and **OMGVMCID** (Section 3.17.1, “Standard Exception Definitions,” on page 3-52) is reserved for OMG use. These will not be allocated to anybody. Vendors can request allocation of **VPVID** by sending mail to tag-*request@omg.org*.

When a **VMCID** (Section 3.17, “Standard Exceptions,” on page 3-51) is allocated to a vendor automatically the same value of **VPVID** is reserved for the vendor and vice versa. So once a vendor gets either a **VMCID** or a **VPVID** registered they can use that value for both their minor codes and their policy types.

4.10.1.1 Copy

Policy copy();

Return Value

This operation copies the policy object. The copy does not retain any relationships that the policy had with any domain, or object.

4.10.1.2 Destroy

void destroy();

This operation destroys the policy object. It is the responsibility of the policy object to determine whether it can be destroyed.

Exception(s)

CORBA::NO_PERMISSION - raised when the policy object determines that it cannot be destroyed.

4.10.1.3 Policy_type

readonly attribute policy_type

Return Value

This readonly attribute returns the constant value of type **PolicyType** that corresponds to the type of the **Policy** object.

4.10.2 Creation of Policy Objects

A generic ORB operation for creating new instances of Policy objects is provided as described in this section.

```
module CORBA {

    typedef short PolicyErrorCode;
    const PolicyErrorCode BAD_POLICY = 0;
    const PolicyErrorCode UNSUPPORTED_POLICY = 1;
    const PolicyErrorCode BAD_POLICY_TYPE = 2;
    const PolicyErrorCode BAD_POLICY_VALUE = 3;
    const PolicyErrorCode UNSUPPORTED_POLICY_VALUE = 4;
```

```
        exception PolicyError {PolicyErrorCode reason;};

        interface ORB {

            .....

            Policy create_policy(
                in PolicyType type,
                in any val
            ) raises(PolicyError);
        };
};
```

4.10.2.1 *PolicyErrorCode*

A request to create a **Policy** may be invalid for the following reasons:

BAD_POLICY - the requested **Policy** is not understood by the ORB.

UNSUPPORTED_POLICY - the requested **Policy** is understood to be valid by the ORB, but is not currently supported.

BAD_POLICY_TYPE - The type of the value requested for the **Policy** is not valid for that **PolicyType**.

BAD_POLICY_VALUE - The value requested for the **Policy** is of a valid type but is not within the valid range for that type.

UNSUPPORTED_POLICY_VALUE - The value requested for the **Policy** is of a valid type and within the valid range for that type, but this valid value is not currently supported.

4.10.2.2 *PolicyError*

```
exception PolicyError {PolicyErrorCode reason;};
```

PolicyError exception is raised to indicate problems with parameter values passed to the **ORB::create_policy** operation. Possible reasons are described above.

4.10.2.3 *INV_POLICY*

```
exception INV_POLICY
```

Due to an incompatibility between **Policy** overrides, the invocation cannot be made. This is a standard system exception that can be raised from any invocation.

4.10.2.4 Create_policy

The ORB operation **create_policy** can be invoked to create new instances of policy objects of a specific type with specified initial state. If **create_policy** fails to instantiate a new **Policy** object due to its inability to interpret the requested type and content of the policy, it raises the PolicyError exception with the appropriate reason as described in “PolicyErrorCode” on page 4-49.

```
Policy create_policy(
    in PolicyType type,
    in any val
) raises(PolicyError);
```

Parameter(s)

type - the **PolicyType** of the policy object to be created.

val - the value that will be used to set the initial state of the **Policy** object that is created.

ReturnValue

Reference to a newly created **Policy** object of type specified by the **type** parameter and initialized to a state specified by the **val** parameter.

Exception(s)

PolicyError - raised when the requested policy is not supported or a requested initial state for the policy is not supported.

When new policy types are added to CORBA or CORBA Services specification, it is expected that the IDL type and the valid values that can be passed to **create_policy** also be specified.

4.10.3 Usages of Policy Objects

Policy Objects are used in general to encapsulate information about a specific policy, with an interface derived from the policy interface. The type of the Policy object determines how the policy information contained within it is used. Usually a Policy object is associated with another object to associate the contained policy with that object.

Objects with which policy objects are typically associated are Domain Managers, POA, the execution environment, both the process/capsule/ORB instance and thread of execution (Current object) and object references. Only certain types of policy object can be meaningfully associated with each of these types of objects.

These relationships are documented in sections that pertain to these individual objects and their usages in various core facilities and object services. The use of Policy Objects with the POA are discussed in the *Portable Object Adaptor* chapter. The use of Policy objects in the context of the Security services, involving their association with Domain Managers as well as with the Execution Environment are discussed in *CORBAservices, Security Service* chapter.

In the following section the association of Policy objects with the Execution Environment is discussed. In "Management of Policy Domains" on page 4-54 the use of Policy objects in association with Domain Managers is discussed.

4.10.4 Policy Associated with the Execution Environment

Certain policies that pertain to services like security (e.g., QOP, Mechanism, invocation credentials etc.) are associated by default with the process/capsule(RM-ODP)/ORB instance (hereinafter referred to as "capsule") when the application is instantiated together with the capsule. By default these policies are applicable whenever an invocation of an operation is attempted by any code executing in the said capsule. The Security service provides operations for modulating these policies on a per-execution thread basis using operations in the **Current** interface. Certain of these policies (e.g., invocation credentials, qop, mechanism etc.) which pertain to the invocation of an operation through a specific object reference can be further modulated at the client end, using the **set_policy_overrides** operation of the **Object** reference. For a description of this operation see "Overriding Associated Policies on an Object Reference" on page 4-39. It associates a specified set of policies with a newly created object reference that it returns.

The association of these overridden policies with the object reference is a purely local phenomenon. These associations are never passed on in any IOR or any other marshaled form of the object reference. the associations last until the object reference in the capsule is destroyed or the capsule in which it exists is destroyed.

The policies thus overridden in this new object reference and all subsequent duplicates of this new object reference apply to all invocations that are done through these object references. The overridden policies apply even when the default policy associated with **Current** is changed. It is always possible that the effective policy on an object reference at any given time will fail to be successfully applied, in which case the invocation attempt using that object reference will fail and return a CORBA::NO_PERMISSION exception. Only certain policies that pertain to the invocation of an operation at the client end can be overridden using this operation. These are listed in the Security specification. Attempts to override any other policy will result in the raising of the CORBA::NO_PERMISSION exception.

In general the policy of a specific type that will be used in an invocation through an specific object reference using a specific thread of execution is determined first by determining if that policy type has been overridden in that object reference. if so then the overridden policy is used. if not then if the policy has been set in the thread of execution then that policy is used. If not then the policy associated with the capsule is used. For policies that matter, the ORB ensures that there is a default policy object of each type that matters associated with each capsule (ORB instance). Hence, in a correctly implemented ORB there is no case when a required type policy is not available to use with an operation invocation.

4.10.5 Specification of New Policy Objects

When new **PolicyType**s are added to CORBA specifications, the following details must be defined. It must be clearly stated which particular uses of a new policy are legal and which are not:

- Specify the assigned **CORBA::PolicyType** and the policy's interface definition.
- If the **Policy** can be created through **CORBA::ORB::create_policy**, specify the allowable values for the any argument 'val' and how they correspond to the initial state/behavior of that **Policy** (such as initial values of attributes). For example, if a Policy has multiple attributes and operations, it is most likely that create_policy will receive some complex data for the implementation to initialize the state of the specific policy:

```
//IDL
struct MyPolicyRange {
      long low;
      long high;
};

const CORBA::PolicyType MY_POLICY_TYPE = 666;
interface MyPolicy : Policy {
      readonly attribute long low;
      readonly attribute long high;
};
```

  If this sample **MyPolicy** can be constructed via create_policy, the specification of **MyPolicy** will have a statement such as: "When instances of **MyPolicy** are created, a value of type **MyPolicyRang**e is passed to **CORBA::ORB::create_policy** and the resulting MyPolicy's attribute 'low' has the same value as the **MyPolicyRange** member 'low' and attribute 'high' has the same value as the **MyPolicyRange** member 'high'.

- If the **Policy** can be passed as an argument to **POA::create_POA**, specify the effects of the new policy on that **POA**. Specifically define incompatibilities (or inter-dependencies) with other **POA** policies, effects on the behavior of invocations on objects activated with the **POA**, and whether or not presence of the POA policy implies some **IOR** profile/component contents for object references created with that **POA**. If the **POA** policy implies some addition/modification to the object reference it is marked as "client-exposed" and the exact details are specified including which profiles are affected and how the effects are represented.
- If the component which is used to carry this information. can be set within a client to tune the client's behavior, specify the policy's effects on the client specifically with respect to (a) establishment of connections and reconnections for an object reference; (b) effects on marshaling of requests; (c) effects on insertion of service contexts into requests; (d) effects upon receipt of service contexts in replies. In addition, incompatibilities (or inter-dependencies) with other client-side policies are stated. For policies that cause service contexts to be added to requests, the exact details of this addition are given.

- If the **Policy** can be used with **POA** creation to tune **IOR** contents and can also be specified (overridden) in the client, specify how to reconcile the policy's presence from both the client and server. It is strongly recommended to avoid this case! As an exercise in completeness, most **POA** policies can probably be extended to have some meaning in the client and vice versa, but this does not help make usable systems, it just makes them more complicated without adding really useful features. There are very few cases where a policy is really appropriate to specify in both places, and for these policies the interaction between the two must be described.
- Pure client-side policies are assumed to be immutable. This allows efficient processing by the runtime that can avoid re-evaluating the policy upon every invocation and instead can perform updates only when new overrides are set (or policies change due to rebind). If the newly specified policy is mutable, it must be clearly stated what happens if non-readonly attributes are set or operations are invoked that have side-effects.
- For certain policy types, override operations may be disallowed. If this is the case, the policy specification must clearly state what happens if such overrides are attempted.

4.10.6 Standard Policies

Table 4-1 below lists the standard policy types that are defined by various parts of CORBA and CORBA Services in this version of CORBA.

Table 4-1 Standard Policy Types

Policy Type	Policy Interface	Defined in Sect./Page	Uses create_ policy
SecClientInvocationAccess	SecurityAdmin::AccessPolicy	Security Service	No
SecTargetInvocationAccess	SecurityAdmin::AccessPolicy	Security Service	No
SecApplicationAccess	SecurityAdmin::AccessPolicy	Security Service	No
SecClientInvocationAudit	SecurityAdmin::AuditPolicy	Security Service	No
SecTargetInvocationAudit	SecurityAdmin::AuditPolicy	Security Service	No
SecApplicationAudit	SecurityAdmin::AuditPolicy	Security Service	No
SecDelegation	SecurityAdmin::DelegationPolicy	Security Service	No
SecClientSecureInvocation	SecurityAdmin::SecureInvocationPolicy	Security Service	No
SecTargetSecureInvocation	SecurityAdmin::SecureInvocationPolicy	Security Service	No
SecNonRepudiation	NRService::NRPolicy	Security Service	No
SecConstruction	CORBA::SecConstruction	CORBA Core - ORB Interface chapter	No
SecMechanismPolicy	SecurityLevel2::MechanismPolicy	Security Service	Yes
SecInvocationCredentialsPolicy	SecurityLevel2::InvocationCredentialsPolicy	Security Service	Yes
SecFeaturesPolicy	SecurityLevel2::FeaturesPolicy	Security Service	Yes

Table 4-1 Standard Policy Types

Policy Type	Policy Interface	Defined in Sect./Page	Uses create_ policy
SecQOPPolicy	SecurityLevel2::QOPPolicy	Security Service	Yes
THREAD_POLICY_ID	PortableServer::ThreadPolicy	CORBA Core - Portable Object Adapter chapter	Yes
LIFESPAN_POLICY_ID	PortableServer::LifespanPolicy	CORBA Core - Portable Object Adapter chapter Core Chapter 11	Yes
ID_UNIQUENESS_POLICY_ID	PortableServer::IdUniquenessPolicy	CORBA Core - Portable Object Adapter chapter Core Chapter 11	Yes
ID_ASSIGNMENT_POLICY_ID	PortableServer::IdAssignmentPolicy	CORBA Core - Portable Object Adapter chapter	Yes
IMPLICIT_ACTIVATION_POLICY_ID	PortableServer::ImplicitActivationPolicy	CORBA Core - Portable Object Adapter chapter	Yes
SERVENT_RETENTION_POLICY_ID	PortableServer::ServentRetentionPolicy	CORBA Core - Portable Object Adapter chapter	Yes
REQUEST_PROCESSING_POLICY_ID	PortableServer::RequestProcessingPolicy	CORBA Core - Portable Object Adapter chapter	Yes
BIDIRECTIONAL_POLICY_TYPE	BiDirPolicy::BidirectionalPolicy	CORBA Core - General Inter-ORB Protocol chapter	Yes
SecDelegationDirectivePolicy	SecurityLevel2::DelegtionDirectivePolicy	Security Service	Yes
SecEstablishTrustPolicy	SecurityLevel2::EstablishTrustPolicy	Security Service	Yes

4.11 Management of Policy Domains

4.11.1 Basic Concepts

This section describes how policies, such as security policies, are associated with objects that are managed by an ORB. The interfaces and operations that facilitate this aspect of management is described in this section together with the section describing **Policy** objects.

4.11.1.1 Policy Domain

A policy domain is a set of objects to which the policies associated with that domain apply. These objects are the domain members. The policies represent the rules and criteria that constrain activities of the objects which belong to the domain. On object reference creation, the ORB implicitly associates the object reference with one or more policy domains. Policy domains provide leverage for dealing with the problem of scale in policy management by allowing application of policy at a domain granularity rather than at an individual object instance granularity.

4.11.1.2 Policy Domain Manager

A policy domain includes a unique object, one per policy domain, called the domain manager, which has associated with it the policy objects for that domain. The domain manager also records the membership of the domain and provides the means to add and remove members. The domain manager is itself a member of a domain, possibly the domain it manages.

4.11.1.3 Policy Objects

A policy object encapsulates a policy of a specific type. The policy encapsulated in a policy object is associated with the domain by associating the policy object with the domain manager of the policy domain.

There may be several policies associated with a domain, with a policy object for each. There is at most one policy of each type associated with a policy domain. The policy objects are thus shared between objects in the domain, rather than being associated with individual objects. Consequently, if an object needs to have an individual policy, then it must be a singleton member of a domain.

4.11.1.4 Object Membership of Policy Domains

Since the only way to access objects is through object references, associating object references with policy domains, implicitly associates the domain policies with the object associated with the object reference. Care should be taken by the application that is creating object references using POA operations to ensure that object references to the same object are not created by the server of that object with different domain associations. Henceforth whenever the concept of “object membership” is used, it actually means the membership of an object reference to the object in question.

An object can simultaneously be a member of more than one policy domain. In that case the object is governed by all policies of its enclosing domains. The reference model allows an object to be a member of multiple domains, which may overlap for the same type of policy (for example, be subject to overlapping access policies). This would require conflicts among policies defined by the multiple overlapping domains to be resolved. The specification does not include explicit support for such overlapping domains and, therefore, the use of policy composition rules required to resolve conflicts at policy enforcement time.

Policy domain managers and policy objects have two types of interfaces:

- The operational interfaces used when enforcing the policies. These are the interfaces used by the ORB during an object invocation. Some policy objects may also be used by applications, which enforce their own policies.

  The caller asks for the policy of a particular type (e.g., the delegation policy), and then uses the policy object returned to enforce the policy. The caller finding a policy and then enforcing it does not see the domain manager objects and the domain structure.

- The administrative interfaces used to set policies (e.g., specifying which events to audit or who can access objects of a specified type in this domain). The administrator sees and navigates the domain structure, so he is aware of the scope of what he is administering.

Note – This specification does not include any explicit interfaces for managing the policy domains themselves: creating and deleting them; moving objects between them; changing the domain structure and adding, changing, and removing policies applied to the domains.

4.11.1.5 Domains Association at Object Reference Creation

When a new object reference is created, the ORB implicitly associates the object reference (and hence the object that it is associated with) with the following elements forming its environment:

- One or more *Policy Domains*, defining all the policies to which the object associated with the object reference is subject.
- The *Technology Domains*, characterizing the particular variants of mechanisms (including security) available in the ORB.

The ORB will establish these associations when one of the object reference creation operations of the POA is called. Some or all of these associations may subsequently be explicitly referenced and modified by administrative or application activity, which might be specifically security-related but could also occur as a side-effect of some other activity, such as moving an object to another host machine.

In some cases, when a new object reference is created, it needs to be associated with a new domain. Within a given domain a construction policy can be associated with a specific object type thus causing a new domain (i.e., a domain manager object) to be created whenever an object reference of that type is created and the newly created object reference associated with the new domain manager. This construction policy is enforced at the same time as the domain membership (i.e., by the POA when it creates an object reference).

4.11.1.6 Implementor's View of Object Creation

For policy domains, the construction policy of the application or factory creating the object proceeds as follows. The application (which may be a generic factory) calls one of the object reference creation operations of the POA to create the new object reference. The ORB obtains the construction policy associated with the creating object, or the default domain absent a creating object.

By default, the new object reference that is created is made a member of the domain to which the parent belongs. Non-object applications on the client side are associated with a default, per-ORB instance policy domain by the ORB.

Each domain manager has a construction policy associated with it, which controls whether, in addition to creating the specified new object reference, a new domain manager is created with it. This object provides a single operation **make_domain_manager** which can be invoked with the **constr_policy** parameter set to TRUE to indicate to the ORB that new object references of the specified type are to be associated their own separate domains. Once such a construction policy is set, it can be reversed by invoking **make_domain_manager** again with the **constr_policy** parameter set to FALSE.

When creating an object reference of the type specified in the **make_domain_manager** call with **constr_policy** set to TRUE, the ORB must also create a new domain for the newly created object reference. If a new domain is needed, the ORB creates both the requested object reference and a domain manager object. A reference to this domain manager can be found by calling **get_domain_managers** on the newly created object reference.

While the management interface to the construction policy object is standardized, the interface from the ORB to the policy object is assumed to be a private one, which may be optimized for different implementations.

If a new domain is created, the policies initially applicable to it are the policies of the enclosing domain. The ORB will always arrange to provide a default enclosing domain with default ORB policies associated with it, in those cases where there would be no such domain as in the case of a non-object client invoking object creation operations.

The calling application, or an administrative application later, can change the domains to which this object belongs, using the domain management interfaces, which will be defined in the future.

Since the ORB has control only over domain associations with object references, it is the responsibility of the creator of new object to ensure that the object references that are created to the new object are associated meaningfully with domains.

4.11.2 Domain Management Operations

This section defines the interfaces and operations needed to find domain managers and find the policies associated with these. However, it does not include operations to manage domain membership, structure of domains, or to manage which policies are associated with domains.

This section also includes the interface to the construction policy object, as that is relevant to domains. The basic definitions of the interfaces and operations related to these are part of the CORBA module, since other definitions in the CORBA module depend on these.

```
module CORBA {
    interface DomainManager {
        Policy get_domain_policy (
            in PolicyType policy_type
        );
    };

    const PolicyType SecConstruction = 11;

    interface ConstructionPolicy: Policy{
        void make_domain_manager(
            in CORBA::InterfaceDef object_type,
            in boolean constr_policy
        );
    };

    typedef sequence <DomainManager> DomainManagersList;
};
```

4.11.2.1 Domain Manager

The domain manager provides mechanisms for:

- Establishing and navigating relationships to superior and subordinate domains.
- Creating and accessing policies.

There should be no unnecessary constraints on the ordering of these activities, for example, it must be possible to add new policies to a domain with a preexisting membership. In this case, some means of determining the members that do not conform to a policy that may be imposed is required. It should be noted that interfaces for adding new policies to domains or for changing domain memberships have not currently been standardized.

All domain managers provide the **get_domain_policy** operation. By virtue of being an object, the Domain Managers also have the **get_policy** and **get_domain_managers** operations, which is available on all objects (see “Getting Policy Associated with the Object” on page 4-38 and “Getting the Domain Managers Associated with the Object” on page 4-39).

CORBA::DomainManager::get_domain_policy

This returns the policy of the specified type for objects in this domain.

```
Policy get_domain_policy (
    in PolicyType policy_type
);
```

Parameter(s)

policy_type - The type of policy for objects in the domain which the application wants to administer. For security, the possible policy types are described in *CORBAservices: Common Object Services Specification*, Security chapter, Security Policies Introduction section.

Return Value

A reference to the policy object for the specified type of policy in this domain.

Exception(s)

CORBA::INV_POLICY - raised when the value of policy type is not valid either because the specified type is not supported by this ORB or because a policy object of that type is not associated with this Object.

4.11.2.2 Construction Policy

The construction policy object allows callers to specify that when instances of a particular object reference are created, they should be automatically assigned membership in a newly created domain at creation time.

CORBA::ConstructionPolicy::make_domain_manager

This operation enables the invoker to set the construction policy that is to be in effect in the domain with which this **ConstructionPolicy** object is associated. Construction Policy can either be set so that when an object reference of the type specified by the input parameter is created, a new domain manager will be created and the newly created object reference will respond to **get_domain_managers** by returning a reference to this domain manager. Alternatively the policy can be set to associate the newly created object reference with the domain associated with the creator. This policy is implemented by the ORB during execution of any one of the object reference creation operations of the POA, and results in the construction of the application-specified object reference and a Domain Manager object if so dictated by the policy in effect at the time of the creation of the object reference.

```
void make_domain_manager (
    in InterfaceDef object_type,
    in boolean constr_policy
);
```

Parameter(s)

object_type - The type of the object references for which Domain Managers will be created. If this is nil, the policy applies to all object references in the domain.

constr_policy - If TRUE the construction policy is set to create a new domain manager associated with the newly created object reference of this type in this domain. If FALSE construction policy is set to associate the newly created object references with the domain of the creator or a default domain as described above.

4.12 Thread-Related Operations

To support single-threaded ORBs, as well as multi-threaded ORBs that run multi-thread-unaware code, several operations are included in the ORB interface. These operations can be used by single-threaded and multi-threaded applications. An application that is a pure ORB client would not need to use these operations. Both the **ORB::run** and **ORB::shutdown** are useful in fully multi-threaded programs.

Note – These operations are defined on the ORB rather than on an object adapter to allow the main thread to be used for all kinds of asynchronous processing by the ORB. Defining these operations on the ORB also allows the ORB to support multiple object adapters, without requiring the application main to know about all the object adapters. The interface between the ORB and an object adapter is not standardized.

4.12.1 work_pending

boolean work_pending( );

This operation returns an indication of whether the ORB needs the main thread to perform some work.

A result of TRUE indicates that the ORB needs the main thread to perform some work and a result of FALSE indicates that the ORB does not need the main thread.

4.12.2 perform_work

void perform_work();

If called by the main thread, this operation performs an implementation-defined unit of work; otherwise, it does nothing.

It is platform-specific how the application and ORB arrange to use compatible threading primitives.

The **work_pending()** and **perform_work()** operations can be used to write a simple polling loop that multiplexes the main thread among the ORB and other activities. Such a loop would most likely be needed in a single-threaded server. A multi-threaded server would need a polling loop only if there were both ORB and other code that required use of the main thread.

Here is an example of such a polling loop:

```
// C++
for (;;) {
   if (orb->work_pending()) {
      orb->perform_work();
   };
   // do other things
   // sleep?
```

```
};
```

Once the ORB has shutdown, **work_pending** and **perform_work** will raise the BAD_INV_ORDER exception with minor code 4. An application can detect this exception to determine when to terminate a polling loop.

4.12.3 run

```
void run();
```

This operation provides execution resources to the ORB so that it can perform its internal functions. Single threaded ORB implementations, and some multi-threaded ORB implementations, need the use of the main thread in order to function properly. For maximum portability, an application should call either **run** or **perform_work** on its main thread. **run** may be called by multiple threads simultaneously.

This operation will block until the ORB has completed the shutdown process, initiated when some thread calls **shutdown**.

4.12.4 shutdown

```
void shutdown(
    in boolean          wait_for_completion
);
```

This operation instructs the ORB to shut down, that is, to stop processing in preparation for destruction.

Shutting down the ORB causes all object adapters to be destroyed, since they cannot exist in the absence of an ORB. Shut down is complete when all ORB processing (including request processing and object deactivation or other operations associated with object adapters) has completed and the object adapters have been destroyed. In the case of the **POA**, this means that all object etherealizations have finished and root **POA** has been destroyed (implying that all descendent **POA**s have also been destroyed).

If the **wait_for_completion** parameter is **TRUE**, this operation blocks until the shut down is complete. If an application does this in a thread that is currently servicing an invocation, the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, since blocking would result in a deadlock.

If the **wait_for_completion** parameter is **FALSE**, then **shutdown** may not have completed upon return. An ORB implementation may require the application to call (or have a pending call to) **run** or **perform_work** after **shutdown** has been called with its parameter set to **FALSE**, in order to complete the shutdown process.

While the ORB is in the process of shutting down, the ORB operates as normal, servicing incoming and outgoing requests until all requests have been completed. An implementation may impose a time limit for requests to complete while a **shutdown** is pending.

Once an ORB has shutdown, only object reference management operations(**duplicate**, **release** and **is_nil**) may be invoked on the ORB or any object reference obtained from it. An application may also invoke the destroy operation on the ORB itself. Invoking any other operation will raise the BAD_INV_ORDER system exception with the OMG minor code 4.

4.12.5 destroy

void destroy();

This operation destroys the ORB so that its resources can be reclaimed by the application. Any operation invoked on a destroyed ORB reference will raise the OBJECT_NOT_EXIST exception. Once an ORB has been destroyed, another call to **ORB_init** with the same **ORBid** will return a reference to a newly constructed ORB.

If **destroy** is called on an ORB that has not been shut down, it will start the shut down process and block until the ORB has shut down before it destroys the ORB. If an application calls **destroy** in a thread that is currently servicing an invocation, the BAD_INV_ORDER system exception will be raised with the OMG minor code 3, since blocking would result in a deadlock.

For maximum portability and to avoid resource leaks, an application should always call **shutdown** and **destroy** on all ORB instances before exiting.

ORB Interoperability Architecture 13

Note – This is the CORBA 2.3 Specification Chapter 13 with a new Section 13.6.7, "Object URLs". The new section is in blue and marked with changebars. Changebars outside of 13.6.7 are *not* for the Interoperable Naming submission.

The ORB Interoperability Architecture chapter has been updated based on CORE changes from ptc/98-09-04 and the Objects by Value documents (orbos/98-01-18 and ptc/98-07-06).

Contents

This chapter contains the following sections.

13.1 Overview

The original Request for Proposal on Interoperability (OMG Document 93-9-15) defines interoperability as the ability for a client on ORB A to invoke an OMG IDL-defined operation on an object on ORB B, where ORB A and ORB B are independently developed. It further identifies general requirements including in particular:

- Ability for two vendors' ORBs to interoperate without prior knowledge of each other's implementation.
- Support of all ORB functionality.
- Preservation of content and semantics of ORB-specific information across ORB boundaries (for example, security).

In effect, the requirement is for invocations between client and server objects to be independent of whether they are on the same or different ORBs, and not to mandate fundamental modifications to existing ORB products.

13.1.1 Domains

The CORBA Object Model identifies various distribution transparencies that must be supported within a single ORB environment, such as location transparency. Elements of ORB functionality often correspond directly to such transparencies. Interoperability can be viewed as extending transparencies to span multiple ORBs.

In this architecture a *domain* is a distinct scope, within which certain common characteristics are exhibited and common rules are observed over which a distribution transparency is preserved. Thus, interoperability is fundamentally involved with transparently crossing such domain boundaries.

Domains tend to be either administrative or technological in nature, and need not correspond to the boundaries of an ORB installation. Administrative domains include naming domains, trust groups, resource management domains and other "run-time" characteristics of a system. Technology domains identify common protocols, syntaxes and similar "build-time" characteristics. In many cases, the need for technology domains derives from basic requirements of administrative domains.

Within a single ORB, most domains are likely to have similar scope to that of the ORB itself: common object references, network addresses, security mechanisms, and more. However, it is possible for there to be multiple domains of the same type supported by a given ORB: internal representation on different machine types, or security domains. Conversely, a domain may span several ORBs: similar network addresses may be used by different ORBs, type identifiers may be shared.

13.1.2 Bridging Domains

The abstract architecture describes ORB interoperability in terms of the translation required when an object request traverses domain boundaries. Conceptually, a mapping or *bridging mechanism* resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain.

The concrete architecture identifies two approaches to inter-ORB bridging:

- At application level, allowing flexibility and portability.
- At ORB level, built into the ORB itself.

13.2 ORBs and ORB Services

The ORB Core is that part of the ORB which provides the basic representation of objects and the communication of requests. The ORB Core therefore supports the minimum functionality to enable a client to invoke an operation on a server object, with (some of) the distribution transparencies required by *CORBA*.

An object request may have implicit attributes which affect the way in which it is communicated - though not the way in which a client makes the request. These attributes include security, transactional capabilities, recovery, and replication. These features are provided by "ORB Services," which will in some ORBs be layered as internal services over the core, or in other cases be incorporated directly into an ORB's core. It is an aim of this specification to allow for new ORB Services to be defined in the future, without the need to modify or enhance this architecture.

Within a single ORB, ORB services required to communicate a request will be implemented and (implicitly) invoked in a private manner. For interoperability between ORBs, the ORB services used in the ORBs, and the correspondence between them, must be identified.

13.2.1 The Nature of ORB Services

ORB Services are invoked implicitly in the course of application-level interactions. ORB Services range from fundamental mechanisms such as reference resolution and message encoding to advanced features such as support for security, transactions, or replication.

An ORB Service is often related to a particular transparency. For example, message encoding – the marshaling and unmarshaling of the components of a request into and out of message buffers – provides transparency of the representation of the request. Similarly, reference resolution supports location transparency. Some transparencies, such as security, are supported by a combination of ORB Services and Object Services while others, such as replication, may involve interactions between ORB Services themselves.

ORB Services differ from Object Services in that they are positioned below the application and are invoked transparently to the application code. However, many ORB Services include components which correspond to conventional Object Services in that they are invoked explicitly by the application.

Security is an example of service with both ORB Service and normal Object Service components, the ORB components being those associated with transparently authenticating messages and controlling access to objects while the necessary administration and management functions resemble conventional Object Services.

13.2.2 ORB Services and Object Requests

Interoperability between ORBs extends the scope of distribution transparencies and other request attributes to span multiple ORBs. This requires the establishment of relationships between supporting ORB Services in the different ORBs.

In order to discuss how the relationships between ORB Services are established, it is necessary to describe an abstract view of how an operation invocation is communicated from client to server object.

1. The client generates an operation request, using a reference to the server object, explicit parameters, and an implicit invocation context. This is processed by certain ORB Services on the client path.
2. On the server side, corresponding ORB Services process the incoming request, transforming it into a form directly suitable for invoking the operation on the server object.
3. The server object performs the requested operation.
4. Any result of the operation is returned to the client in a similar manner.

The correspondence between client-side and server-side ORB Services need not be one-to-one and in some circumstances may be far more complex. For example, if a client application requests an operation on a replicated server, there may be multiple server-side ORB service instances, possibly interacting with each other.

In other cases, such as security, client-side or server-side ORB Services may interact with Object Services such as authentication servers.

13.2.3 Selection of ORB Services

The ORB Services used are determined by:

- Static properties of both client and server objects; for example, whether a server is replicated.
- Dynamic attributes determined by a particular invocation context; for example, whether a request is transactional.
- Administrative policies (e.g., security).

Within a single ORB, private mechanisms (and optimizations) can be used to establish which ORB Services are required and how they are provided. Service selection might in general require negotiation to select protocols or protocol options. The same is true between different ORBs: it is necessary to agree which ORB Services are used, and how each transforms the request. Ultimately, these choices become manifest as one or more protocols between the ORBs or as transformations of requests.

In principle, agreement on the use of each ORB Service can be independent of the others and, in appropriately constructed ORBs, services could be layered in any order or in any grouping. This potentially allows applications to specify selective transparencies according to their requirements, although at this time CORBA provides no way to penetrate its transparencies.

A client ORB must be able to determine which ORB Services must be used in order to invoke operations on a server object. Correspondingly, where a client requires dynamic attributes to be associated with specific invocations, or administrative policies dictate, it must be possible to cause the appropriate ORB Services to be used on client and server sides of the invocation path. Where this is not possible - because, for example, one ORB does not support the full set of services required - either the interaction cannot proceed or it can only do so with reduced facilities or transparencies.

13.3 Domains

From a computational viewpoint, the OMG Object Model identifies various distribution transparencies which ensure that client and server objects are presented with a uniform view of a heterogeneous distributed system. From an engineering viewpoint, however, the system is not wholly uniform. There may be distinctions of location and possibly many others such as processor architecture, networking mechanisms and data representations. Even when a single ORB implementation is used throughout the system, local instances may represent distinct, possibly optimized scopes for some aspects of ORB functionality.

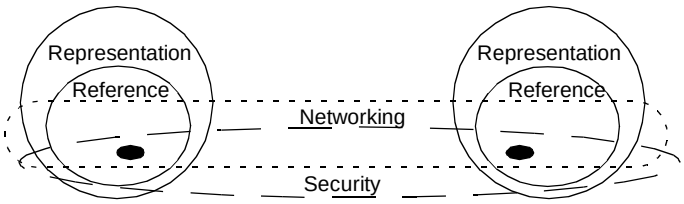

Figure 13-1 Different Kinds of Domains can Coexist.

Interoperability, by definition, introduces further distinctions, notably between the scopes associated with each ORB. To describe both the requirements for interoperability and some of the solutions, this architecture introduces the concept of *domains* to describe the scopes and their implications.

Informally, a domain is a set of objects sharing a common characteristic or abiding by common rules. It is a powerful modelling concept which can simplify the analysis and description of complex systems. There may be many types of domains (e.g., management domains, naming domains, language domains, and technology domains).

13.3.1 Definition of a Domain

Domains allow partitioning of systems into collections of components which have some characteristic in common. In this architecture a domain is a scope in which a collection of objects, said to be members of the domain, is associated with some common characteristic; any object for which the association does not exist, or is undefined, is not a member of the domain. A domain can be modelled as an object and may be itself a member of other domains.

It is the scopes themselves and the object associations or bindings defined within them which characterize a domain. This information is disjoint between domains. However, an object may be a member of several domains, of similar kinds as well as of different kinds, and so the sets of members of domains may overlap.

The concept of a domain boundary is defined as the limit of the scope in which a particular characteristic is valid or meaningful. When a characteristic in one domain is translated to an equivalent in another domain, it is convenient to consider it as traversing the boundary between the two domains.

Domains are generally either administrative or technological in nature. Examples of domains related to ORB interoperability issues are:

- Referencing domain – the scope of an object reference
- Representation domain – the scope of a message transfer syntax and protocol
- Network addressing domain – the scope of a network address
- Network connectivity domain – the potential scope of a network message
- Security domain – the extent of a particular security policy
- Type domain – the scope of a particular type identifier
- Transaction domain – the scope of a given transaction service

Domains can be related in two ways: containment, where a domain is contained within another domain, and federation, where two domains are joined in a manner agreed to and set up by their administrators.

13.3.2 Mapping Between Domains: Bridging

Interoperability between domains is only possible if there is a well-defined mapping between the behaviors of the domains being joined. Conceptually, a mapping mechanism or bridge resides at the boundary between the domains, transforming requests expressed in terms of one domain's model into the model of the destination domain. Note that the use of the term "bridge" in this context is conceptual and refers

only to the functionality which performs the required mappings between distinct domains. There are several implementation options for such bridges and these are discussed elsewhere.

For full interoperability, it is essential that all the concepts used in one domain are transformable into concepts in other domains with which interoperability is required, or that if the bridge mechanism filters such a concept out, nothing is lost as far as the supported objects are concerned. In other words, one domain may support a superior service to others, but such a superior functionality will not be available to an application system spanning those domains.

A special case of this requirement is that the object models of the two domains need to be compatible. This specification assumes that both domains are strictly compliant with the CORBA Object Model and the *CORBA* specifications. This includes the use of OMG IDL when defining interfaces, the use of the CORBA Core Interface Repository, and other modifications that were made to *CORBA*. Variances from this model could easily compromise some aspects of interoperability.

13.4 Interoperability Between ORBs

An ORB "provides the mechanisms by which objects transparently make and receive requests and responses. In so doing, the ORB provides interoperability between applications on different machines in heterogeneous distributed environments..." ORB interoperability extends this definition to cases in which client and server objects on different ORBs "transparently make and receive requests..."

Note that a direct consequence of this transparency requirement is that bridging must be bidirectional: that is, it must work as effectively for object references passed as parameters as for the target of an object invocation. Were bridging unidirectional (e.g., if one ORB could only be a client to another) then transparency would not have been provided, because object references passed as parameters would not work correctly: ones passed as "callback objects," for example, could not be used.

Without loss of generality, most of this specification focuses on bridging in only one direction. This is purely to simplify discussions, and does not imply that unidirectional connectivity satisfies basic interoperability requirements.

13.4.1 ORB Services and Domains

In this architecture, different aspects of ORB functionality - ORB Services - can be considered independently and associated with different domain types. The architecture does not, however, prescribe any particular decomposition of ORB functionality and interoperability into ORB Services and corresponding domain types. There is a range of possibilities for such a decomposition:

1. The simplest model, for interoperability, is to treat all objects supported by one ORB (or, alternatively, all ORBs of a given type) as comprising one domain. Interoperability between any pair of different domains (or domain types) is then achieved by a specific all-encompassing bridge between the domains. (This is all *CORBA* implies.)

2. More detailed decompositions would identify particular domain types - such as referencing, representation, security, and networking. A core set of domain types would be pre-determined and allowance made for additional domain types to be defined as future requirements dictate (e.g., for new ORB Services).

13.4.2 ORBs and Domains

In many respects, issues of interoperability between ORBs are similar to those which can arise with a single type of ORB (e.g., a product). For example:

- Two installations of the ORB may be installed in different security domains, with different Principal identifiers. Requests crossing those security domain boundaries will need to establish locally meaningful Principals for the caller identity, and for any Principals passed as parameters.
- Different installations might assign different type identifiers for equivalent types, and so requests crossing type domain boundaries would need to establish locally meaningful type identifiers (and perhaps more).

Conversely, not all of these problems need to appear when connecting two ORBs of a different type (e.g., two different products). Examples include:

- They could be administered to share user visible naming domains, so that naming domains do not need bridging.
- They might reuse the same networking infrastructure, so that messages could be sent without needing to bridge different connectivity domains.

Additional problems can arise with ORBs of different types. In particular, they may support different concepts or models, between which there are no direct or natural mappings. CORBA only specifies the application level view of object interactions, and requires that distribution transparencies conceal a whole range of lower level issues. It follows that within any particular ORB, the mechanisms for supporting transparencies are not visible at the application-level and are entirely a matter of implementation choice. So there is no guarantee that any two ORBs support similar internal models or that there is necessarily a straightforward mapping between those models.

These observations suggest that the concept of an ORB (instance) is too coarse or superficial to allow detailed analysis of interoperability issues between ORBs. Indeed, it becomes clear that an ORB instance is an elusive notion: it can perhaps best be characterized as the intersection or coincidence of ORB Service domains.

13.4.3 Interoperability Approaches

When an interaction takes place across a domain boundary, a mapping mechanism, or bridge, is required to transform relevant elements of the interaction as they traverse the boundary. There are essentially two approaches to achieving this: mediated bridging and immediate bridging. These approaches are described in the following subsections.

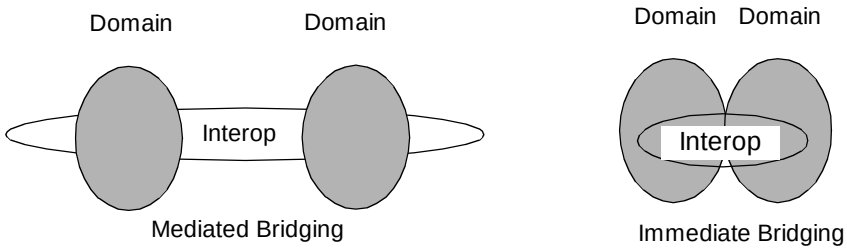

Figure 13-2 Two bridging techniques, different uses of an intermediate form agreed on between the two domains.

13.4.3.1 Mediated Bridging

With mediated bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, between the internal form of that domain and an agreed, common form.

Observations on mediated bridging are as follows:

- The scope of agreement of a common form can range from a private agreement between two particular ORB/domain implementations to a universal standard.
- There can be more than one common form, each oriented or optimized for a different purpose.
- If there is more than one possible common form, then which is used can be static (e.g., administrative policy agreed between ORB vendors, or between system administrators) or dynamic (e.g., established separately for each object, or on each invocation).
- Engineering of this approach can range from in-line specifically compiled (compare to stubs) or generic library code (such as encryption routines), to intermediate bridges to the common form.

13.4.3.2 Immediate Bridging

With immediate bridging, elements of the interaction relevant to the domain are transformed, at the boundary of each domain, directly between the internal form of one domain and the internal form of the other.

Observations on immediate bridging are as follows:

- This approach has the potential to be optimal (in that the interaction is not mediated via a third party, and can be specifically engineered for each pair of domains) but sacrifices flexibility and generality of interoperability to achieve this.
- This approach is often applicable when crossing domain boundaries which are purely administrative (i.e., there is no change of technology). For example, when crossing security administration domains between similar ORBs, it is not necessary to use a common intermediate standard.

As a general observation, the two approaches can become almost indistinguishable when private mechanisms are used between ORB/domain implementations.

13.4.3.3 Location of Inter-Domain Functionality

Logically, an inter-domain bridge has components in both domains, whether the mediated or immediate bridging approach is used. However, domains can span ORB boundaries and ORBs can span machine and system boundaries; conversely, a machine may support, or a process may have access to more than one ORB (or domain of a given type). From an engineering viewpoint, this means that the components of an inter-domain bridge may be dispersed or co-located, with respect to ORBs or systems. It also means that the distinction between an ORB and a bridge can be a matter of perspective: there is a duality between viewing inter-system messaging as belonging to ORBs, or to bridges.

For example, if a single ORB encompasses two security domains, the inter-domain bridge could be implemented wholly within the ORB and thus be invisible as far as ORB interoperability is concerned. A similar situation arises when a bridge between two ORBs or domains is implemented wholly within a process or system which has access to both. In such cases, the engineering issues of inter-domain bridging are confined, possibly to a single system or process. If it were practical to implement all bridging in this way, then interactions between systems or processes would be solely within a single domain or ORB.

13.4.3.4 Bridging Level

As noted at the start of this section, bridges may be implemented both internally to an ORB and as layers above it. These are called respectively “in-line” and “request-level” bridges.

Request-level bridges use the CORBA APIs, including the Dynamic Skeleton Interface, to receive and issue requests. However, there is an emerging class of “implicit context” which may be associated with some invocations, holding ORB Service information such as transaction and security context information, which is not at this time exposed through general purpose public APIs. (Those APIs expose only OMG IDL-defined operation parameters, not implicit ones.) Rather, the precedent set with the Transaction Service is that special purpose APIs are defined to allow bridging of each kind of context. This means that request-level bridges must be built to specifically understand the implications of bridging such ORB Service domains, and to make the appropriate API calls.

13.4.4 Policy-Mediated Bridging

An assumption made through most of this specification is that the existence of domain boundaries should be transparent to requests: that the goal of interoperability is to hide such boundaries. However, if this were always the goal, then there would be no real need for those boundaries in the first place.

Realistically, administrative domain boundaries exist because they reflect ongoing differences in organizational policies or goals. Bridging the domains will in such cases require *policy mediation*. That is, inter-domain traffic will need to be constrained, controlled, or monitored; fully transparent bridging may be highly undesirable. Resource management policies may even need to be applied, restricting some kinds of traffic during certain periods.

Security policies are a particularly rich source of examples: a domain may need to audit external access, or to provide domain-based access control. Only a very few objects, types of objects, or classifications of data might be externally accessible through a "firewall."

Such policy-mediated bridging requires a bridge that knows something about the traffic being bridged. It could in general be an application-specific policy, and many policy-mediated bridges could be parts of applications. Those might be organization-specific, off-the-shelf, or anywhere in between.

Request-level bridges, which use only public ORB APIs, easily support the addition of policy mediation components, without loss of access to any other system infrastructure that may be needed to identify or enforce the appropriate policies.

13.4.5 Configurations of Bridges in Networks

In the case of network-aware ORBs, we anticipate that some ORB protocols will be more frequently bridged to than others, and so will begin to serve the role of "backbone ORBs." (This is a role that the IIOP is specifically expected to serve.) This use of "backbone topology" is true both on a large scale and a small scale. While a

large scale public data network provider could define its own backbone ORB, on a smaller scale, any given institution will probably designate one commercially available ORB as its backbone.

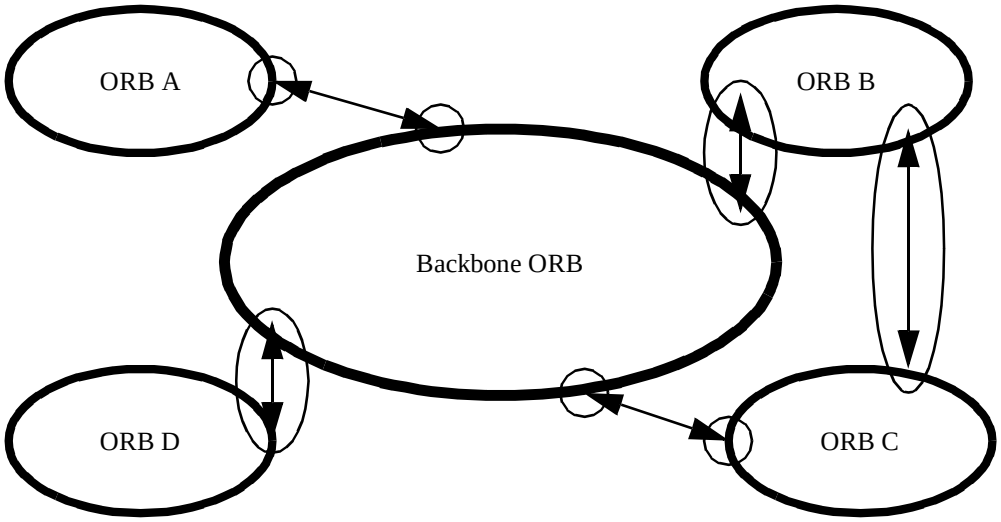

Figure 13-3 An ORB chosen as a backbone will connect other ORBs through bridges, both full-bridges and half-bridges.

Adopting a backbone style architecture is a standard administrative technique for managing networks. It has the consequence of minimizing the number of bridges needed, while at the same time making the ORB topology match typical network organizations. (That is, it allows the number of bridges to be proportional to the number of protocols, rather than combinatorial.)

In large configurations, it will be common to notice that adding ORB bridges doesn't even add any new "hops" to network routes, because the bridges naturally fit in locations where connectivity was already indirect, and augment or supplant the existing network firewalls.

13.5 Object Addressing

The Object Model (see Chapter 1, Requests) defines an object reference as an object name that reliably denotes a particular object. An object reference identifies the same object each time the reference is used in a request, and an object may be denoted by multiple, distinct references.

The fundamental ORB interoperability requirement is to allow clients to use such object names to invoke operations on objects in other ORBs. Clients do not need to distinguish between references to objects in a local ORB or in a remote one. Providing this transparency can be quite involved, and naming models are fundamental to it.

This section of this specification discusses models for naming entities in multiple domains, and transformations of such names as they cross the domain boundaries. That is, it presents transformations of object reference information as it passes through

networks of inter-ORB bridges. It uses the word "ORB" as synonymous with referencing domain; this is purely to simplify the discussion. In other contexts, "ORB" can usefully denote other kinds of domain.

13.5.1 Domain-relative Object Referencing

Since CORBA does not require ORBs to understand object references from other ORBs, when discussing object references from multiple ORBs one must always associate the object reference's domain (ORB) with the object reference. We use the notation *D0.R0* to denote an object reference *R0* from domain *D0*; this is itself an object reference. This is called "domain-relative" referencing (or addressing) and need not reflect the implementation of object references within any ORB.

At an implementation level, associating an object reference with an ORB is only important at an inter-ORB boundary; that is, inside a bridge. This is simple, since the bridge knows from which ORB each request (or response) came, including any object references embedded in it.

13.5.2 Handling of Referencing Between Domains

When a bridge hands an object reference to an ORB, it must do so in a form understood by that ORB: the object reference must be in the recipient ORB's native format. Also, in cases where that object originated from some other ORB, the bridge must associate each newly created "proxy" object reference with (what it sees as) the original object reference.

Several basic schemes to solve these two problems exist. These all have advantages in some circumstances; all can be used, and in arbitrary combination with each other, since CORBA object references are opaque to applications. The ramifications of each scheme merits attention, with respect to scaling and administration. The schemes include:

1. *Object Reference Translation Reference Embedding*: The bridge can store the original object reference itself, and pass an entirely different proxy reference into the new domain. The bridge must then manage state on behalf of each bridged object reference, map these references from one ORB's format to the other's, and vice versa.

2. *Reference Encapsulation:* The bridge can avoid holding any state at all by conceptually concatenating a domain identifier to the object name. Thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1*... *D4* it could be identified in *D4* as proxy reference *d3.d2.d1.d0.R*, where *dn* is the address of *Dn* relative to *Dn+1*.

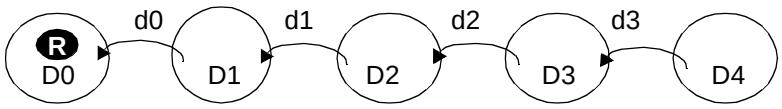

Figure 13-4 Reference encapsulation adds domain information during bridging.

3. *Domain Reference Translation:* Like object reference translation, this scheme holds some state in the bridge. However, it supports sharing that state between multiple object references by adding a domain-based route identifier to the proxy (which still holds the original reference, as in the reference encapsulation scheme). It achieves this by providing encoded domain route information each time a domain boundary is traversed; thus if a reference *D0.R*, originating in domain *D0*, traversed domains *D1...D4* it would be identified in *D4* as *(d3, x3).R*, and in *D2* as *(d1,x1).R*, and so on, where *dn* is the address of *Dn* relative to *Dn+1*, and *xn* identifies the pair *(dn-1, xn-1)*.

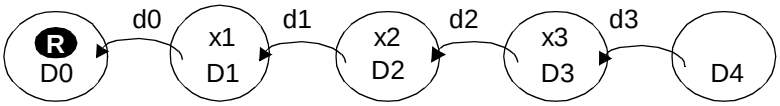

Figure 13-5 Domain Reference Translation *substitutes domain references* during bridging.

4. *Reference Canonicalization:* This scheme is like domain reference translation, except that the proxy uses a "well-known" (e.g., global) domain identifier rather than an encoded path. Thus a reference *R*, originating in domain *D0* would be identified in other domains as *D0.R*.

Observations about these approaches to inter-domain reference handling are as follows:

- Naive application of reference encapsulation could lead to arbitrarily large references. A "topology service" could optimize cycles within any given encapsulated reference and eliminate the appearance of references to local objects as alien references.
- A topology service could also optimize the chains of routes used in the domain reference translation scheme. Since the links in such chains are re-used by any path traversing the same sequence of domains, such optimization has particularly high leverage.

- With the general purpose APIs defined in *CORBA*, object reference translation can be supported even by ORBs not specifically intended to support efficient bridging, but this approach involves the most state in intermediate bridges. As with reference encapsulation, a topology service could optimize individual object references. (APIs are defined by the Dynamic Skeleton Interface and Dynamic Invocation Interface)
- The chain of addressing links established with both object and domain reference translation schemes must be represented as state within the network of bridges. There are issues associated with managing this state.
- Reference canonicalization can also be performed with managed hierarchical name spaces such as those now in use on the Internet and X.500 naming.

13.6 An Information Model for Object References

This section provides a simple, powerful information model for the information found in an object reference. That model is intended to be used directly by developers of bridging technology, and is used in that role by the IIOP, described in the *General Inter-ORB Protocol* chapter, *Object References* section.

13.6.1 What Information Do Bridges Need?

The following potential information about object references has been identified as critical for use in bridging technologies:

- *Is it null?* Nulls only need to be transmitted and never support operation invocation.
- *What type is it?* Many ORBs require knowledge of an object's type in order to efficiently preserve the integrity of their type systems.
- *What protocols are supported?* Some ORBs support objrefs that in effect live in multiple referencing domains, to allow clients the choice of the most efficient communications facilities available.
- *What ORB Services are available?* As noted in "Selection of ORB Services" on page 13-66, several different ORB Services might be involved in an invocation. Providing information about those services in a standardized way could in many cases reduce or eliminate negotiation overhead in selecting them.

13.6.2 Interoperable Object References: IORs

To provide the information above, an "Interoperable Object Reference," (IOR) data structure has been provided. This data structure need not be used internally to any given ORB, and is not intended to be visible to application-level ORB programmers. It should be used only when crossing object reference domain boundaries, within bridges.

This data structure is designed to be efficient in typical single-protocol configurations, while not penalizing multiprotocol ones.

```
module IOP {                                        // IDL

    // Standard Protocol Profile tag values

    typedef unsigned long           ProfileId;
    const ProfileId                 TAG_INTERNET_IOP = 0;
    const ProfileId                 TAG_MULTIPLE_COMPONENTS = 1;

    struct TaggedProfile {
        ProfileId                   tag;
        sequence <octet>            profile_data;
    };

    // an Interoperable Object Reference is a sequence of
    // object-specific protocol profiles, plus a type ID.

    struct IOR {
        string                      type_id;
        sequence <TaggedProfile>    profiles;
    };

    // Standard way of representing multicomponent profiles.
    // This would be encapsulated in a TaggedProfile.

    typedef unsigned long ComponentId;
    struct TaggedComponent {
        ComponentId                 tag;
        sequence <octet>            component_data;
    };
    typedef sequence <TaggedComponent> MultipleComponentProfile;
};
```

Object references have at least one *tagged profile*. Each profile supports one or more protocols and encapsulates all the basic information the protocols it supports need to identify an object. Any single profile holds enough information to drive a complete invocation using any of the protocols it supports; the content and structure of those profile entries are wholly specified by these protocols. A bridge between two domains may need to know the detailed content of the profile for those domains' profiles, depending on the technique it uses to bridge the domains[1].

Each profile has a unique numeric tag, assigned by the OMG. The ones defined here are for the IIOP (see Section 15.7.3, "IIOP IOR Profile Components," on page 15-51) and for use in "multiple component profiles." Profile tags in the range **0x80000000** through **0xffffffff** are reserved for future use, and are not currently available for assignment.

1. Based on topology and policy information available to it, a bridge may find it prudent to add or remove some profiles as it forwards an object reference. For example, a bridge acting as a firewall might remove all profiles except ones that make such profiles, letting clients that understand the profiles make routing choices.

Null object references are indicated by an empty set of profiles, and by a "Null" type ID (a string which contains only a single terminating character). A Null TypeID is the only mechanism that can be used to represent the type **CORBA::Object**. Type IDs may only be "Null" in any message, requiring the client to use existing knowledge or to consult the object, to determine interface types supported. The type ID is a Repository ID identifying the interface type, and is provided to allow ORBs to preserve strong typing. This identifier is agreed on within the bridge and, for reasons outside the scope of this interoperability specification, needs to have a much broader scope to address various problems in system evolution and maintenance. Type IDs support detection of type equivalence, and in conjunction with an Interface Repository, allow processes to reason about the relationship of the type of the object referred to and any other type.

The type ID, if provided by the server, indicates the most derived type that the server wishes to publish, at the time the reference is generated. The object's actual most derived type may later change to a more derived type. Therefore, the type ID in the IOR can only be interpreted by the client as a hint that the object supports at least the indicated interface. The client can succeed in narrowing the reference to the indicated interface, or to one of its base interfaces, based solely on the type ID in the IOR, but must not fail to narrow the reference without consulting the object via the "_is_a" or "_get_interface" pseudo-operations.

13.6.2.1 *The* **TAG_INTERNET_IOP** *Profile*

The **TAG_INTERNET_IOP** tag identifies profiles that support the Internet Inter-ORB Protocol. The **ProfileBody** of this profile, described in detail in Section 15.7.2, "IIOP IOR Profiles," on page 15-49, contains a CDR encapsulation of a structure containing addressing and object identification information used by IIOP. Version 1.1 of the **TAG_INTERNET_IOP** profile also includes a **sequence<TaggedComponent>** that can contain additional information supporting optional IIOP features, ORB services such as security, and future protocol extensions.

Protocols other than IIOP (such as ESIOPs and other GIOPs) can share profile information (such as object identity or security information) with IIOP by encoding their additional profile information as components in the **TAG_INTERNET_IOP** profile. All **TAG_INTERNET_IOP** profiles support IIOP, regardless of whether they also support additional protocols. Interoperable ORBs are not required to create or understand any other profile, nor are they required to create or understand any of the components defined for other protocols that might share the **TAG_INTERNET_IOP** profile with IIOP.

13.6.2.2 *The* **TAG_MULTIPLE_COMPONENTS** *Profile*

The **TAG_MULTIPLE_COMPONENTS** tag indicates that the value encapsulated is of type **MultipleComponentProfile**. In this case, the profile consists of a list of protocol components, indicating ORB services accessible using that protocol. ORB services are assigned component identifiers in a namespace that is distinct from the profile identifiers. Note that protocols may use the **MultipleComponentProfile** data

structure to hold profile components even without using **TAG_MULTIPLE_COMPONENTS** to indicate that particular protocol profile, and need not use a **MultipleComponentProfile** to hold sets of profile components.

13.6.2.3 IOR Components

TaggedComponents contained in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles are identified by unique numeric tags using a namespace distinct form that is used for profile tags. Component tags are assigned by the OMG.

Specifications of components must include the following information:

- *Component ID:* The compound tag that is obtained from OMG.
- *Structure and encoding:* The syntax of the component data and the encoding rules. If the component value is encoded as a CDR encapsulation, the IDL type that is encapsulated and the GIOP version which is used for encoding the value, if different than GIOP 1.0, must be specified as part of the component definition.
- *Semantics:* How the component data is intended to be used.
- *Protocols:* The protocol for which the component is defined, and whether it is intended that the component be usable by other protocols.
- *At most once:* whether more than one instance of this component can be included in a profile.

Specification of protocols must describe how the components affect the protocol. The following should be specified in any protocol definition for each **TaggedComponent** that the protocol uses:

- *Mandatory presence:* Whether inclusion of the component in profiles supporting the protocol is required (MANDATORY PRESENCE) or not required (OPTIONAL PRESENCE).
- *Droppable:* For optional presence component, whether component, if present, must be retained or may be dropped.

13.6.3 Standard IOR Components

The following are standard IOR components that can be included in **TAG_INTERNET_IOP** and **TAG_MULTIPLE_COMPONENTS** profiles, and may apply to IIOP, other GIOPs, ESIOPs, or other protocols. An ORB must not drop these components from an existing IOR.

```
module IOP {
    const ComponentId    TAG_ORB_TYPE = 0;
    const ComponentId    TAG_CODE_SETS = 1;
    const ComponentId    TAG_POLICIES = 2;
    const ComponentId    TAG_ALTERNATE_IIOP_ADDRESS = 3;

    const ComponentId    TAG_ASSOCIATION_OPTIONS = 13;
```

```
    const ComponentId    TAG_SEC_NAME = 14;
    const ComponentId    TAG_SPKM_1_SEC_MECH = 15;
    const ComponentId    TAG_SPKM_2_SEC_MECH = 16;
    const ComponentId    TAG_KerberosV5_SEC_MECH = 17;
    const ComponentId    TAG_CSI_ECMA_Secret_SEC_MECH = 18;
    const ComponentId    TAG_CSI_ECMA_Hybrid_SEC_MECH = 19;
    const ComponentId    TAG_SSL_SEC_TRANS = 20;
    const ComponentId    TAG_CSI_ECMA_Public_SEC_MECH = 21;
    const ComponentId    TAG_ GENERIC_SEC_MECH = 22;
    const ComponentId    TAG_JAVA_CODEBASE = 25;
};
```

The following additional components that can be used by other protocols are specified in the DCE ESIOP chapter of this document and *CORBAServices*, Security Service, in the Security Service for DCE ESIOP section:

```
    const ComponentId    TAG_COMPLETE_OBJECT_KEY = 5;
    const ComponentId    TAG_ENDPOINT_ID_POSITION = 6;
    const ComponentId    TAG_LOCATION_POLICY = 12;
    const ComponentId    TAG_DCE_STRING_BINDING = 100;
    const ComponentId    TAG_DCE_BINDING_NAME = 101;
    const ComponentId    TAG_DCE_NO_PIPES = 102;
    const ComponentId    TAG_DCE_SEC_MECH = 103; // Security Service
```

13.6.3.1 **TAG_ORB_TYPE** *Component*

It is often useful in the real world to be able to identify the particular kind of ORB an object reference is coming from, to work around problems with that particular ORB, or exploit shared efficiencies.

The **TAG_ORB_TYPE** component has an associated value of type **unsigned long**, encoded as a CDR encapsulation, designating an ORB type ID allocated by the OMG for the ORB type of the originating ORB. Anyone may register any ORB types by submitting a short (one-paragraph) description of the ORB type to the OMG, and will receive a new ORB type ID in return. A list of ORB type descriptions and values will be made available on the OMG web server.

The **TAG_ORB_TYPE** component can appear at most once in any IOR profile. For profiles supporting IIOP 1.1 or greater, it is optionally present and may not be dropped.

13.6.3.2 **TAG_ALTERNATE_IIOP_ADDRESS** *Component*

In cases where the same object key is used for more than one internet location, the following standard IOR Component is defined for support in IIOP version 1.2.

The **TAG_ALTERNATE_IIOP_ADDRESS** component has an associated value of type

```
    struct {
```

```
        string HostID,
        short Port
    };
```

encoded as a CDR encapsulation.

Zero or more instances of the **TAG_ALTERNATE_IIOP_ADDRESS** component type may be included in a version 1.2 **TAG_INTERNET_IOP** Profile. Each of these alternative addresses may be used by the client orb, in addition to the host and port address expressed in the body of the Profile. In cases where one or more **TAG_ALTERNATE_IIOP_ADDRESS** components are present in a **TAG_INTERNET_IOP** Profile, no order of use is prescribed by Version 1.2 of IIOP.

13.6.3.3 Other Components

The following standard components are specified in various OMG specifications:

- **TAG_CODE_SETS** (See Section 13.7.2.4, "CodeSet Component of IOR Multi-Component Profile," on page 13-99.)
- **TAG_POLICIES** (See CORBA Messaging specification - currently orbos/98-05-05, will be incorporated into CORBA 3.0).
- **TAG_SEC_NAME** (See Section 15.10.2 Mechanism Tags, Security chapter - CORBAServices).
- **TAG_ASSOCIATION_OPTIONS** (See Section 15.10.3 Tag Association Options, Security chapter - CORBAServices).
- **TAG_SSL_SEC_TRANS** (See Section 15.10.2 Mechanism Tags, Security chapter - CORBAServices).
- **TAG_GENERIC_SEC_MECH** and all other tags with names in the form **TAG_*_SEC_MECH** (See Section 15.10.2 Mechanism Tags, Security chapter - CORBAServices).
- **TAG_JAVA_CODEBASE** (See the *Java to IDL Language Mapping*, Section 1.4.9.3, "Codebase Transmission," on page 1-33).
- **TAG_COMPLETE_OBJECT_KEY** (See Section 16.5.4, "Complete Object Key Component," on page 16-19).
- **TAG_ENDPOINT_ID_POSITION** (See Section 16.5.5, "Endpoint ID Position Component," on page 16-20).
- **TAG_LOCATION_POLICY** (See Section 16.5.6, "Location Policy Component," on page 16-20).
- **TAG_DCE_STRING_BINDING** (See Section 16.5.1, "DCE-CIOP String Binding Component," on page 16-17).
- **TAG_DCE_BINDING_NAME** (See Section 16.5.2, "DCE-CIOP Binding Name Component," on page 16-18).
- **TAG_DCE_NO_PIPES** (See Section 16.5.3, "DCE-CIOP No Pipes Component," on page 16-19).

13.6.4 Profile and Component Composition in IORs

The following rules augment the preceding discussion:

1. Profiles must be independent, complete, and self-contained. Their use shall not depend on information contained in another profile.
2. Any invocation uses information from exactly one profile.
3. Information used to drive multiple inter-ORB protocols may coexist within a single profile, possibly with some information (e.g., components) shared between the protocols, as specified by the specific protocols.
4. Unless otherwise specified in the definition of a particular profile, multiple profiles with the same profile tag may be included in an IOR.
5. Unless otherwise specified in the definition of a particular component, multiple components with the same component tag may be part of a given profile within an IOR.
6. A **TAG_MULTIPLE_COMPONENTS** profile may hold components shared between multiple protocols. Multiple such profiles may exist in an IOR.
7. The definition of each protocol using a **TAG_MULTIPLE_COMPONENTS** profile must specify which components it uses, and how it uses them.
8. Profile and component definitions can be either public or private. Public definitions are those whose tag and data format is specified in OMG documents. For private definitions, only the tag is registered with OMG.
9. Public component definitions shall state whether or not they are intended for use by protocols other than the one(s) for which they were originally defined, and dependencies on other components.

The OMG is responsible for allocating and registering protocol and component tags. Neither allocation nor registration indicates any “standard” status, only that the tag will not be confused with other tags. Requests to allocate tags should be sent to tag_request@omg.org.

13.6.5 IOR Creation and Scope

IORs are created from object references when required to cross some kind of referencing domain boundary. ORBs will implement object references in whatever form they find appropriate, including possibly using the IOR structure. Bridges will normally use IORs to mediate transfers where that standard is appropriate.

13.6.6 Stringified Object References

Object references can be “stringified” (turned into an external string form) by the **ORB::object_to_string** operation, and then “destringified” (turned back into a programming environment’s object reference representation) using the **ORB::string_to_object** operation.

There can be a variety of reasons why being able to parse this string form might *not* help make an invocation on the original object reference:

- Identifiers embedded in the string form can belong to a different domain than the ORB attempting to destringify the object reference.
- The ORBs in question might not share a network protocol, or be connected.
- Security constraints may be placed on object reference destringification.

Nonetheless, there is utility in having a defined way for ORBs to generate and parse stringified IORs, so that in some cases an object reference stringified by one ORB could be destringified by another.

To allow a stringified object reference to be internalized by what may be a different ORB, a stringified IOR representation is specified. This representation instead establishes that ORBs could parse stringified object references using that format. This helps address the problem of bootstrapping, allowing programs to obtain and use object references, even from different ORBs.

The following is the representation of the stringified (externalized) IOR:

```
(1)    <oref>      ::= <prefix> <hex_Octets>
(2)    <prefix>    ::= "IOR:"
(3)    <hex_Octets>::= <hex_Octet> {<hex_Octet>}*
(4)    <hex_Octet> ::= <hexDigit> <hexDigit>
(5)    <hexDigit>  ::= <digit> | <a> | <b> | <c> | <d> | <e> | <f>
(6)    <digit>     ::= "0" | "1" | "2" | "3" | "4" | "5" |
                       |  "6" | "7" | "8" | "9"
(7)    <a>         ::= "a" | "A"
(8)    <b>         ::= "b" | "B"
(9)    <c>         ::= "c" | "C"
(10)   <d>         ::= "d" | "D"
(11)   <e>         ::= "e" | "E"
(12)   <f>         ::= "f" | "F"
```

The hexadecimal strings are generated by first turning an object reference into an IOR, and then encapsulating the IOR using the encoding rules of CDR, as specified in GIOP 1.0. (See Section 15.3, "CDR Transfer Syntax," on page 15-5 for more information.) The content of the encapsulated IOR is then turned into hexadecimal digit pairs, starting with the first octet in the encapsulation and going until the end. The high four bits of each octet are encoded as a hexadecimal digit, then the low four bits.

13.6.7 Object URLs

To address the problem of bootstrapping and allow for more convenient exchange of human-readable object references, `ORB::string_to_object` allows URLs in the `corbaloc` and `corbaname` formats to be converted into object references. If conversion fails, `string_to_object` raises a `BAD_PARAM` exception with the following minor codes:

- BadSchemeName
- BadAddress
- BadSchemeSpecificPart
- Other

13.6.7.1 corbaloc *URL*

The corbaloc URL scheme provides stringified object references that are more easily manipulated by users than IOR URLs. Currently , corbaloc URLs denote objects that can be contacted by IIOP or resolve_initial_references. Other transport protocols can be explicitly specified when they become available. Examples of IIOP and resolve_initial_references (rir:) based corbaloc URLs are:

```
corbaloc::555xyz.com/Prod/TradingService
corbaloc:iiop:1.1@555xyz.com/Prod/TradingService
corbaloc::555xyz.com,:556xyz.com:80/Dev/NameService
corbaloc:rir:/TradingService
corbaloc:rir:/NameService
```

A corbaloc URL contains one or more:

- protocol identifiers
- protocol specific components such as address and version information.

When the rir protocol is used, no other protocols are allowed.

After the addressing information, a corbaloc URL ends with a single object key.

The full syntax is:

```
<corbaloc>          = “corbaloc:”<obj_addr_list>[“/”<key_string>]
<obj_addr_list>     = [<obj_addr> “,”]* <obj_addr>
<obj_addr>          = <prot_addr> | <future_prot_addr>
<prot_addr>         = <rir_prot_addr> | <iiop_prot_addr>

<rir_prot_addr>     = <rir_prot_token>”:”
<rir_prot_token>    = “rir”

<iiop_prot_addr>    = <iiop_id><iiop_addr>
<iiop_id>           = “:” | <iiop_prot_token>”:”
<iiop_prot_token>   = “iiop”
<iiop_addr>         = defined in Section 13.6.7.3, “corbaloc:iiop
URL”

<future_prot_addr>    = <future_prot_id><future_prot_addr>
<future_prot_id>       = <future_prot_token>”:”
```

```
<future_prot_token>    = possible examples: “atm” | “dce”
<future_prot_addr>     = protocol specific address

<key_string>        = <string> | empty_string
```

Where:

obj_addr_list: comma-separated list of protocol id, version, and address information. This list is used in an implementation-defined manner to address the object An object may be contacted by any of the addresses and protocols.

Note – If the `rir` protocol is used, no other protocols are allowed.

---

obj_addr: A protocol identifier, version tag, and a protocol specific address. The comma ‘,’ and ‘/’ characters are specifically prohibited in this component of the URL.

rir_prot_addr: resolve_initial_references protocol identifier. This protocol does not have a version tag or address. See Section 13.6.7.2

iiop_prot_addr: iiop protocol identifier, version tag, and address containing a DNS-style host name or IP address. See Section 13.6.7.3, “corbaloc:iiop URL” for the iiop specific definitions.

future_prot_addr: a placeholder for future corbaloc protocols.

future_prot_id: token representing a protocol terminated with a “:”.

future_prot_token: token representing a protocol. Currently only “`iiop`” and “rir” are defined.

future_prot_addr: a protocol specific address and possibly protocol version information. An example of this for `iiop` is “`1.1@555xyz.com`”

key_string: a stringified object key

The `key_string` corresponds to the octet sequence in the `object_key` member of a GIOP `Request` or `LocateRequest` header as defined in section 15.4 of CORBA 2.3. The `key_string` uses the escape conventions described in RFC 2396 to map away from octet values that cannot directly be part of a URL. US-ASCII alphanumeric characters are not escaped. Characters outside this range are escaped, except for the following:

“;” | “/” | “?”| “:” | “@” | “&” | “=” | “+” | “$” |

“,” | “-” | “_” | ”.” | “!” | “~” | “*” | “’” | “(“ | “)”

The `key_string` is not NUL-terminated.

13.6.7.2 corbaloc:rir *URL*

The corbaloc:rir URL is defined to allow access to the ORB's configured initial references through a URL.

The protocol address syntax is:

```
<rir_prot_addr>    = <rir_prot_token>":"
<rir_prot_token>   = "rir"
```

Where:

rir_prot_addr: resolve_initial_references protocol identifier. There is no version or address information when `rir` is used.

rir_prot_token: The token "rir" identifies this protocol..

For a `corbaloc:rir` URL, the `<key_string>` is used as the argument to resolve_initial_references. An empty `<key_string>` is interpreted as the default "NameService".

The `rir` protocol can not be used with any other protocol in a URL.

13.6.7.3 corbaloc:iiop *URL*

The corbaloc:iiop URL is defined for use in TCP/IP- and DNS-centric environments The full protocol address syntax is:

```
<iiop_prot_addr>   = <iiop_id><iiop_addr>
<iiop_id>          = <iiop_default> | <iiop_prot_token>":"
<iiop_default>     = ":"
<iiop_prot_token>  = "iiop"
<iiop_addr>        = <version> <host> [":" <port>]
<host>             = DNS-style Host Name | ip_address
<version>          = <major> "." <minor> "@" | empty_string
<port>             = number
<major>            = number
<minor>            = number
```

Where:

iiop_prot_addr: iiop protocol identifier, version tag, and address containing a DNS-style host name or IP address.

iiop_id: tokens recognized to indicate an iiop protocol corbaloc.

iiop_default: default token indicating iiop protocol, ":".

iiop_prot_token: iiop protocol token, "iiop"

iiop_address: a single address

host: DNS-style host name or IP address.

version: a major and minor version number, separated by '.' and followed by '@'. If the version is absent, 1.0 is assumed.

ip_address: numeric IP address (dotted decimal notation)

port: port number the agent is listening on (see below). The default port is 2089.

13.6.7.4 `corbaloc` *Server Implementation*

The only requirements on an object advertised by a `corbaloc` URL are that there must be a software agent listening on the host and port specified by the URL. This agent must be capable of handling GIOP `Request` and `LocateRequest` messages targeted at the object key specified in the URL.

A normal CORBA server meets these criteria. It is also possible to implement lightweight object location forwarding agents that respond to GIOP `Request` messages with `Reply` messages with a `LOCATION_FORWARD` status, and respond to GIOP `LocateRequest` messages with `LocateReply` messages.

13.6.7.5 `corbaname` *URL*

The `corbaname` URL scheme is described in Chapter 3 of the CORBAservices specification. It extends the capabilities of the `corbaloc` scheme to allow URLs to denote entries in a Naming Service. Resolving `corbaname` URLs does not require a Naming Service implementation in the ORB core. Some examples are:

```
corbaname::555objs.com#a/string/path/to/obj
```

This URL specifies that at host `555objs.com`, a object of type `NamingContext` (with an object key of `NameService`) can be found, or alternatively, that an agent is running at that location which will return a reference to a `NamingContext`. The (stringified) name `a/string/path/to/obj` is then used as the argument to a `resolve_str` operation on that `NamingContext`. The URL denotes the object reference that results from that lookup.

```
corbaname:rir:#a/local/obj
```

This URL specifies that the stringified name `a/local/obj` is to be resolved relative to the naming context returned by resolve_initial_references("NameService").

13.6.7.6 Future `corbaloc` *URL Protocols*

This specification only defines use of iiop and rir with corbaloc. New protocols can be added to corbaloc as required. Each new protocol must implement the <future_prot_addr> component of the URL and define a described in Section 13.6.7.1, "corbaloc URL."

A possible example of a future corbaloc URL that incorporates an ATM address is:

corbaloc:iiop:xyz.com,atm:E.164:358.400.1234567/dev/test/objectX

13.6.7.7 Future URL Schemes

Several currently defined non-CORBA URL scheme names are reserved. Implementations may choose to provide these or other URL schemes to support additional ways of denoting objects with URLs.

Table 13-1 lists the required and some optional formats.

Table 13-1 URL formats

Scheme	Description	Status
IOR:	Standard stringified IOR format	Required
corbaloc:	Simple object reference. rir: must be supported.	Required
corbaname:	CosName URL	Required
file://	Specifies a file containing a URL/IOR	Optional
ftp://	Specifies a file containing a URL/IOR that is accessible via ftp protocol.	Optional
http://	Specifies an HTTP URL that returns an object URL/IOR.	Optional

13.6.8 Object Service Context

Emerging specifications for Object Services occasionally require service-specific context information to be passed implicitly with requests and replies. (Specifications for OMG's Object Services are contained in *CORBAservices: Common Object Service Specifications.*) The Interoperability specifications define a mechanism for identifying and passing this service-specific context information as "hidden" parameters. The specification makes the following assumptions:

- Object Service specifications that need additional context passed will completely specify that context as an OMG IDL data type.
- ORB APIs will be provided that will allow services to supply and consume context information at appropriate points in the process of sending and receiving requests and replies.
- It is an ORB's responsibility to determine when to send service-specific context information, and what to do with such information in incoming messages. It may be possible, for example, for a server receiving a request to be unable to de-encapsulate and use a certain element of service-specific context, but nevertheless still be able to successfully reply to the message.

As shown in the following OMG IDL specification, the IOP module provides the mechanism for passing Object Service–specific information. It does not describe any service-specific information. It only describes a mechanism for transmitting it in the most general way possible. The mechanism is currently used by the DCE ESIOP and could also be used by the Internet Inter-ORB protocol (IIOP) General Inter_ORB Protocol (GIOP).

Each Object Service requiring implicit service-specific context to be passed through GIOP will be allocated a unique service context ID value by OMG. Service context ID values are of type **unsigned long**. Object service specifications are responsible for describing their context information as single OMG IDL data types, one data type associated with each service context ID.

The marshaling of Object Service data is described by the following OMG IDL:

```
module IOP {        // IDL

    typedef unsigned long           ServiceId;

    struct ServiceContext {
        ServiceId           context_id;
        sequence <octet>    context_data;
    };
    typedef sequence <ServiceContext>ServiceContextList;

    const ServiceId         TransactionService = 0;
    const ServiceId         CodeSets = 1;
    const ServiceId         ChainBypassCheck = 2;
    const ServiceId         ChainBypassInfo = 3;
    const ServiceId         LogicalThreadId = 4;
    const ServiceId         BI_DIR_IIOP = 5;
    const ServiceId         SendingContextRunTime = 6;
    const ServiceId         INVOCATION_POLICIES = 7;
    const ServiceId         FORWARDED_IDENTITY = 8;
    const ServiceId         UnknownExceptionInfo = 9;
};
```

The context data for a particular service will be encoded as specified for its service-specific OMG IDL definition, and that encoded representation will be encapsulated in the **context_data** member of **IOP::ServiceContext**. (See Section 15.3.3, "Encapsulation," on page 15-13). The **context_id** member contains the service ID value identifying the service and data format. Context data is encapsulated in octet sequences to permit ORBs to handle context data without unmarshaling, and to handle unknown context data types.

During request and reply marshaling, ORBs will collect all service context data associated with the *Request* or *Reply* in a **ServiceContextList**, and include it in the generated messages. No ordering is specified for service context data within the list. The list is placed at the beginning of those messages to support security policies that may need to apply to the majority of the data in a request (including the message headers).

Each Object Service requiring implicit service-specific context to be passed through GIOP will be allocated a unique service context ID value by the OMG. Service context ID values are of type unsigned long. Object service specifications are responsible for describing their context information as single OMG IDL data types, one data type associated with each service context ID.

The high-order 20 bits of service-context ID contain a 20-bit vendor service context codeset ID (VSCID); the low-order 12 bits contain the rest of the service context ID. A vendor (or group of vendors) who wish to define a specific set of system exception minor codes should obtain a unique VSCID from the OMG, and then define a specific set of service context IDs using the VSCID for the high-order bits.

The VSCID of zero is reserved for use for OMG-defined standard service context IDs (i.e., service context IDs in the range 0-4095 are reserved as OMG standard service contexts).

The **ServiceId**s currently defined are:

- **TransactionService** identifies a CDR encapsulation of the **CosTSInteroperation::PropogationContext** defined in *CORBAservices: Common Object Services Specifications*.
- **CodeSets** identifies a CDR encapsulation of the **CONV_FRAME::CodeSetContext** defined in Section 13.7.2.5, "GIOP Code Set Service Context," on page 13-100.
- DCOM-CORBA Interworking uses three service contexts as defined in "DCOM-CORBA Interworking" in the "Interoperability with non-CORBA Systems"chapter. They are:
  - **ChainBypassCheck**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassCheck**. This is carried only in a **Request** message as described in Section 20.9.1, "CORBA Chain Bypass," on page 20-19.
  - **ChainBypassInfo**, which carries a CDR encapsulation of the **struct CosBridging::ChainBypassInfo**. This is carried only in a **Reply** message as described in Section 20.9.1, "CORBA Chain Bypass," on page 20-19.
  - **LogicalThreadId**, which carries a CDR encapsulation of the **struct CosBridging::LogicalThreadId** as described in Section 20.10, "Thread Identification," on page 20-21.
- **BI_DIR_IIOP** identifies a CDR encapsulation of the **IIOP::BiDirIIOPServiceContext** defined in Section 15.8, "Bi-Directional GIOP," on page 15-52.
- **SendingContextRunTime** identifies a CDR encapsulation of the IOR of the **SendingContext::RunTime** object (see Section 5.6, "Access to the Sending Context Run Time," on page 5-15).
- **UnknownExceptionInfo** identifies a CDR encapsulation of a marshaled instance of a **java.lang.throwable** or one of its subclasses as described in *Java to IDL Language Mapping*, Section 1.4.8.1, "Mapping of UnknownExceptionInfo Service Context," on page 1-32.
- The **profile_data** for the **TAG_INTERNET_IOP** profile is a CDR encapsulation of the **IIOP::ProfileBody_1_1** type, described in Section 15.7.2, "IIOP IOR Profiles," on page 15-49.

- The **profile_data** for the **TAG_MULTIPLE_COMPONENTS** profile is a CDR encapsulation of the **MultipleComponentProfile** type, which is a sequence of **TaggedComponent** structures, described in Section 13.6, "An Information Model for Object References," on page 13-77.
- The **component_data member** identifies a CDR encapsulation of a **BindingNameComponent** structure, described in Section 16.5.2.1, "BindingNameComponent," on page 16-18.

Note – For more information on INVOCATION_POLICIES refer to the Asynchronous Messaging specification - orbos/98-05-05. For information on FORWARDED_IDENTITY refer to the Firewall specification - orbos/98-05-04.

Service context IDs are associated with a specific version of GIOP, but will always be allocated in the OMG service context range. This allows any ORB to recognize when it is receiving a standard service context, even if it has been defined in a version of GIOP that it does not support.

The following are the rules for processing a received service context:

- The service context is in the OMG defined range:
  - If it is valid for the supported GIOP version, then it must be processed correctly according to the rules associated with it for that GIOP version level.
  - If it is not valid for the GIOP version, then it may be ignored by the receiving ORB, however it must be passed on through a bridge. No exception shall be raised.
- The service context is not in the OMG-defined range:
  - The receiving ORB may choose to ignore it, process it if it "understands" it, or raise a system exception, however it must be passed on through a bridge. If a system exception is raised, it shall be BAD_PARAM with an OMG standard minor code of 1.

The association of service contexts with GIOP versions, (along with some other supported features tied to GIOP minor version), is shown in Table 13-2.

Table 13-2 Feature Support Tied to Minor GIOP Version Number

Feature	Version 1.0	Version 1.1	Version 1.2
Transaction Service Context	yes	yes	yes
Codeset Negotiation Service Context		yes	yes
DCOM Bridging Service Contexts: **ChainBypassCheck** **ChainBypassInfo** **LogicalThreadId**			yes
Object by Value Service Context: **SendingContextRunTime**			yes

Table 13-2 Feature Support Tied to Minor GIOP Version Number *(Continued)*

Feature	Version 1.0	Version 1.1	Version 1.2
Bi-Directional IIOP Service Context: **BI_DIR_IIOP**			yes
Java Language Throwable Service Context: **UnknownExceptionInfo**			yes
IOR components in IIOP profile		yes	yes
TAG_ORB_TYPE		yes	yes
TAG_CODE_SETS		yes	yes
TAG_ALTERNATE_IIOP_ADDRESS			yes
TAG_ASSOCIATION_OPTION		yes	yes
TAG_SEC_NAME		yes	yes
TAG_SSL_SEC_TRANS		yes	yes
TAG_GENERIC_SEC_MECH		yes	yes
TAG_*_SEC_MECH		yes	yes
TAG_JAVA_CODEBASE			yes
IOR component nn			yes
Extended IDL data types		yes	yes
Bi-Directional GIOP Features			yes

13.7 Code Set Conversion

13.7.1 Character Processing Terminology

This section introduces a few terms and explains a few concepts to help understand the character processing portions of this document.

13.7.1.1 Character Set

A finite set of different characters used for the representation, organization, or control of data. In this specification, the term “character set” is used without any relationship to code representation or associated encoding. Examples of character sets are the English alphabet, Kanji or sets of ideographic characters, corporate character sets (commonly used in Japan), and the characters needed to write certain European languages.

13.7.1.2 Coded Character Set, or Code Set

A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation or numeric value. In this specification, the term “code set” is used as an abbreviation for the term “coded character set.” Examples include ASCII, ISO 8859-1, JIS X0208 (which includes Roman characters, Japanese hiragana, Greek characters, Japanese kanji, etc.) and Unicode.

13.7.1.3 Code Set Classifications

Some language environments distinguish between byte-oriented and “wide characters.” The byte-oriented characters are encoded in one or more 8-bit bytes. A typical single-byte encoding is ASCII as used for western European languages like English. A typical multi-byte encoding which uses from one to three 8-bit bytes for each character is eucJP (Extended UNIX Code - Japan, packed format) as used for Japanese workstations.

Wide characters are a fixed 16 or 32 bits long, and are used for languages like Chinese, Japanese, etc., where the number of combinations offered by 8 bits is insufficient and a fixed-width encoding is needed. A typical example is Unicode (a “universal” character set defined by the The Unicode Consortium, which uses an encoding scheme identical to ISO 10646 UCS-2, or 2-byte Universal Character Set encoding). An extended encoding scheme for Unicode characters is UTF-16 (UCS Transformation Format, 16-bit representations).

The C language has data types **`char`** for byte-oriented characters and **`wchar_t`** for wide characters. The language definition for C states that the sizes for these characters are implementation-dependent. Some environments do not distinguish between byte-oriented and wide characters (e.g., Ada and Smalltalk). Here again, the size of a character is implementation-dependent. The following table illustrates code set classifications as used in this document.

Table 13-3 Code Set Classification

Orientation	Code Element Encoding	Code Set Examples	C Data Type
byte-oriented	single-byte	ASCII, ISO 8859-1 (Latin-1), EBCDIC, ...	**`char`**
	multi-byte	UTF-8, eucJP, Shift-JIS, JIS, Big5, ...	**`char[]`**
non-byte-oriented	fixed-length	ISO 10646 UCS-2 (Unicode), ISO 10646 UCS-4, UTF-16, ...	**`wchar_t`**

13.7.1.4 Narrow and Wide Characters

Some language environments distinguish between “narrow” and “wide” characters. Typically the narrow characters are considered to be 8-bit long and are used for western European languages like English, while the wide characters are 16-bit or 32-

bit long and are used for languages like Chinese, Japanese, etc., where the number of combinations offered by 8 bits are insufficient. However, as noted above there are common encoding schemes in which Asian characters are encoded using multi-byte code sets and it is incorrect to assume that Asian characters are always encoded as "wide" characters.

Within this specification, the general terms "narrow character" and "wide character" are only used in discussing OMG IDL.

13.7.1.5 Char Data and Wchar Data

The phrase "**char** data" in this specification refers to data whose IDL types have been specified as **char** or **string**. Likewise "**wchar** data" refers to data whose IDL types have been specified as **wchar** or **wstring**.

13.7.1.6 Byte-Oriented Code Set

An encoding of characters where the numeric code corresponding to a character code element can occupy one or more bytes. A byte as used in this specification is synonymous with octet, which occupies 8 bits.

13.7.1.7 Multi-Byte Character Strings

A character string represented in a byte-oriented encoding where each character can occupy one or more bytes is called a multi-byte character string. Typically, wide characters are converted to this form from a (fixed-width) process code set before transmitting the characters outside the process (see below about process code sets). Care must be taken to correctly process the component bytes of a character's multi-byte representation.

13.7.1.8 Non-Byte-Oriented Code Set

An encoding of characters where the numeric code corresponding to a character code element can occupy fixed 16 or 32 bits.

13.7.1.9 Char Transmission Code Set (TCS-C) and Wchar Transmission Code Set (TCS-W)

These two terms refer to code sets that are used for transmission between ORBs after negotiation is completed. As the names imply, the first one is used for **char** data and the second one for **wchar** data. Each TCS can be byte-oriented or non-byte oriented.

13.7.1.10 Process Code Set and File Code Set

Processes generally represent international characters in an internal fixed-width format which allows for efficient representation and manipulation. This internal format is called a “process code set.” The process code set is irrelevant outside the process, and hence to the interoperation between CORBA clients and servers through their respective ORBs.

When a process needs to write international character information out to a file, or communicate with another process (possibly over a network), it typically uses a different encoding called a “file code set.” In this specification, unless otherwise indicated, all references to a program’s code set refer to the file code set, not the process code set. Even when a client and server are located physically on the same machine, it is possible for them to use different file code sets.

13.7.1.11 Native Code Set

A native code set is the code set which a client or a server uses to communicate with its ORB. There might be separate native code sets for **char** and **wchar** data.

13.7.1.12 Transmission Code Set

A transmission code set is the commonly agreed upon encoding used for character data transfer between a client’s ORB and a server’s ORB. There are two transmission code sets established per session between a client and its server, one for **char** data (TCS-C) and the other for **wchar** data (TCS-W). Figure 13-6 illustrates these relationships:

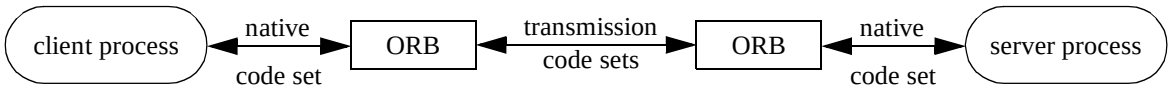

Figure 13-6 Transmission Code Sets

The intent is for TCS-C to be byte-oriented and TCS-W to be non-byte-oriented. However, this specification does allow both types of characters to be transmitted using the same transmission code set. That is, the selection of a transmission code set is orthogonal to the wideness or narrowness of the characters, although a given code set may be better suited for either narrow or wide characters.

13.7.1.13 Conversion Code Set (CCS)

With respect to a particular ORB’s native code set, the set of other or target code sets for which an ORB can convert all code points or character encodings between the native code set and that target code set. For each code set in this CCS, the ORB maintains appropriate translation or conversion procedures and advertises the ability to use that code set for transmitted data in addition to the native code set.

13.7.2 Code Set Conversion Framework

13.7.2.1 Requirements

The file code set that an application uses is often determined by the platform on which it runs. In Japan, for example, Japanese EUC is used on Unix systems, while Shift-JIS is used on PCs. Code set conversion is therefore required to enable interoperability across these platforms. This proposal defines a framework for the automatic conversion of code sets in such situations. The requirements of this framework are:

1. Backward compatibility. In previous CORBA specifications, IDL type **char** was limited to ISO 8859-1. The conversion framework should be compatible with existing clients and servers that use ISO 8859-1 as the code set for **char**.

2. Automatic code set conversion. To facilitate development of CORBA clients and servers, the ORB should perform any necessary code set conversions automatically and efficiently. The IDL type **octet** can be used if necessary to prevent conversions.

3. Locale support. An internationalized application determines the code set in use by examining the LOCALE string (usually found in the LANG environment variable), which may be changed dynamically at run time by the user. Example LOCALE strings are fr_FR.ISO8859-1 (French, used in France with the ISO 8859-1 code set) and ja_JP.ujis (Japanese, used in Japan with the EUC code set and X11R5 conventions for LOCALE). The conversion framework should allow applications to use the LOCALE mechanism to indicate supported code sets, and thus select the correct code set from the registry.

4. CMIR and SMIR support. The conversion framework should be flexible enough to allow conversion to be performed either on the client or server side. For example, if a client is running in a memory-constrained environment, then it is desirable for code set converters to reside in the server and for a Server Makes It Right (SMIR) conversion method to be used. On the other hand, if many servers are executed on one server machine, then converters should be placed in each client to reduce the load on the server machine. In this case, the conversion method used is Client Makes It Right (CMIR).

13.7.2.2 Overview of the Conversion Framework

Both the client and server indicate a native code set indirectly by specifying a locale. The exact method for doing this is language-specific, such as the XPG4 C/C++ function **`setlocale`**. The client and server use their native code set to communicate with their ORB. (Note that these native code sets are in general different from process code sets and hence conversions may be required at the client and server ends.)

The conversion framework is illustrated in Figure 13-7. The server-side ORB stores a server's code set information in a component of the IOR multiple-component profile structure (see Section 13.6.2, "Interoperable Object References: IORs," on page 13-77)[2]. The code sets actually used for transmission are carried in the service context field of an IOP (Inter-ORB Protocol) request header (see Section 13.6.8,

"Object Service Context," on page 13-89 and Section 13.7.2.5, "GIOP Code Set Service Context," on page 13-100). Recall that there are two code sets (TCS-C and TCS-W) negotiated for each session.

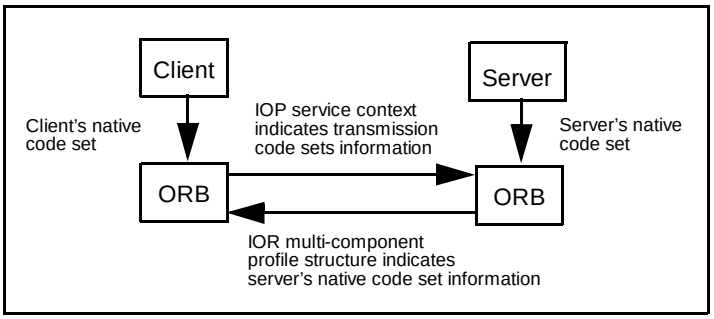

Figure 13-7 Code Set Conversion Framework Overview

If the native code sets used by a client and server are the same, then no conversion is performed. If the native code sets are different and the client-side ORB has an appropriate converter, then the CMIR conversion method is used. In this case, the server's native code set is used as the transmission code set. If the native code sets are different and the client-side ORB does not have an appropriate converter but the server-side ORB does have one, then the SMIR conversion method is used. In this case, the client's native code set is used as the transmission code set.

The conversion framework allows clients and servers to specify a native **char** code set and a native **wchar** code set, which determine the local encodings of IDL types **char** and **wchar**, respectively. The conversion process outlined above is executed independently for the **char** code set and the **wchar** code set. In other words, the algorithm that is used to select a transmission code set is run twice, once for **char** data and once for **wchar** data.

The rationale for selecting two transmission code sets rather than one (which is typically inferred from the locale of a process) is to allow efficient data transmission without any conversions when the client and server have identical representations for **char** and/or **wchar** data. For example, when a Windows NT client talks to a Windows NT server and they both use Unicode for wide character data, it becomes possible to transmit wide character data from one to the other without any conversions. Of course, this becomes possible only for those wide character representations that are well-defined, not for any proprietary ones. If a single transmission code set was mandated, it might require unnecessary conversions. (For example, choosing Unicode as the transmission code set would force conversion of all byte-oriented character data to Unicode.)

---

2. Version 1.1 of the IIOP profile body can also be used to specify the server's code set information, as this version introduces an extra field that is a sequence of tagged components.

13.7.2.3 ORB Databases and Code Set Converters

The conversion framework requires an ORB to be able to determine the native code set for a locale and to convert between code sets as necessary. While the details of exactly how these tasks are accomplished are implementation-dependent, the following databases and code set converters might be used:

- Locale database. This database defines a native code set for a process. This code set could be byte-oriented or non-byte-oriented and could be changed programmatically while the process is running. However, for a given session between a client and a server, it is fixed once the code set information is negotiated at the session's setup time.
- Environment variables or configuration files. Since the locale database can only indicate one code set while the ORB needs to know two code sets, one for **char** data and one for **wchar** data, an implementation can use environment variables or configuration files to contain this information on native code sets.
- Converter database. This database defines, for each code set, the code sets to which it can be converted. From this database, a set of "conversion code sets" (CCS) can be determined for a client and server. For example, if a server's native code set is eucJP, and if the server-side ORB has eucJP-to-JIS and eucJP-to-SJIS bilateral converters, then the server's conversion code sets are JIS and SJIS.
- Code set converters. The ORB has converters which are registered in the converter database.

13.7.2.4 CodeSet Component of IOR Multi-Component Profile

The code set component of the IOR multi-component profile structure contains:

- server's native **char** code set and conversion code sets, and
- server's native **wchar** code set and conversion code sets.

Both **char** and **wchar** conversion code sets are listed in order of preference. The code set component is identified by the following tag:

```
const IOP::ComponentID TAG_CODE_SETS = 1;
```

This tag has been assigned by OMG (See "Standard IOR Components" on page 13-80.). The following IDL structure defines the representation of code set information within the component:

```
module CONV_FRAME {                          // IDL
        typedef unsigned long CodeSetId;
        struct CodeSetComponent {
            CodeSetId               native_code_set;
            sequence<CodeSetId>     conversion_code_sets;
        };
        struct CodeSetComponentInfo {
```

```
        CodeSetComponent        ForCharData;
        CodeSetComponent        ForWcharData;
    };
};
```

Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (See "Character and Code Set Registry" on page 13-106 for further information). Data within the code set component is represented as a structure of type **CodeSetComponentInfo**, and is encoded as a CDR encapsulation. In other words, the **char** code set information comes first, then the **wchar** information, represented as structures of type **CodeSetComponent**.

A null value should be used in the **native_code_set** field if the server desires to indicate no native code set (possibly with the identification of suitable conversion code sets).

If the code set component is not present in a multi-component profile structure, then the default **char** code set is ISO 8859-1 for backward compatibility. However, there is no default **wchar** code set. If a server supports interfaces that use wide character data but does not specify the **wchar** code sets that it supports, client-side ORBs will raise exception INV_OBJREF.

13.7.2.5 GIOP Code Set Service Context

The code set GIOP service context contains:

- **char** transmission code set, and
- **wchar** transmission code set

in the form of a code set service. This service is identified by:

```
const IOP::ServiceID  CodeSets = 1;
```

The following IDL structure defines the representation of code set service information:

```
module CONV_FRAME {                         // IDL
        typedef unsigned long CodeSetId;
        struct CodeSetContext {
            CodeSetId               char_data;
            CodeSetId               wchar_data;
        };
};
```

Code sets are identified by a 32-bit integer id from the OSF Character and Code Set Registry (See "Character and Code Set Registry" on page 13-106 for further information).

Note – A server's **char** and **wchar** Code set components are usually different, but under some special circumstances they can be the same. That is, one could use the same code set for both **char** data and **wchar** data. Likewise the **CodesetId**s in the service context don't have to be different.

13.7.2.6 Code Set Negotiation

The client-side ORB determines a server's native and conversion code sets from the code set component in an IOR multi-component profile structure, and it determines a client's native and conversion code sets from the locale setting (and/or environment variables/configuration files) and the converters that are available on the client. From this information, the client-side ORB chooses **char** and **wchar** transmission code sets (TCS-C and TCS-W). For both requests and replies, the **char** TCS-C determines the encoding of **char** and **string** data, and the **wchar** TCS-W determines the encoding of **wchar** and **wstring** data.

Code set negotiation is not performed on a per-request basis, but only when a client initially connects to a server. All text data communicated on a connection are encoded as defined by the TCSs selected when the connection is established.

Figure 13-8 illustrates, there are two channels for character data flowing between the client and the server. The first, TCS-C, is used for **char** data and the second, TCS-W, is used for **wchar** data. Also note that two native code sets, one for each type of data, could be used by the client and server to talk to their respective ORBs (as noted earlier, the selection of the particular native code set used at any particular point is done via **`setlocale`** or some other implementation-dependent method).

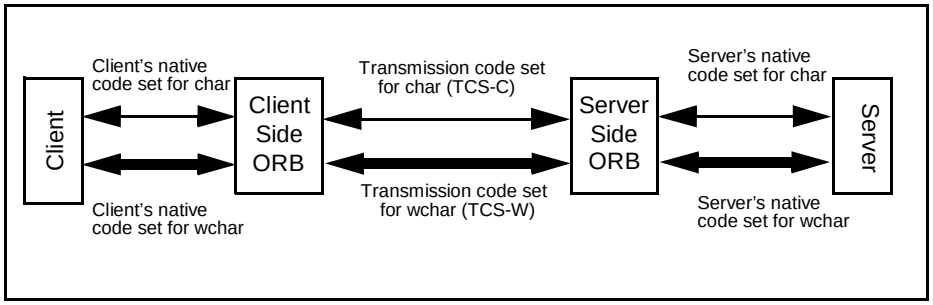

Figure 13-8 Transmission Code Set Use

Let us look at an example. Assume that the code set information for a client and server is as shown in the table below. (Note that this example concerns only **char** code sets and is applicable only for data described as **char**s in the IDL.)

	Client	**Server**
Native code set:	SJIS	eucJP
Conversion code sets:	eucJP, JIS	SJIS, JIS

The client-side ORB first compares the native code sets of the client and server. If they are identical, then the transmission and native code sets are the same and no conversion is required. In this example, they are different, so code set conversion is necessary. Next, the client-side ORB checks to see if the server's native code set, eucJP, is one of the conversion code sets supported by the client. It is, so eucJP is selected as the transmission code set, with the client (i.e., its ORB) performing conversion to and from its native code set, SJIS, to eucJP. Note that the client may first have to convert all its data described as **char**s (and possibly **wchar_t**s) from process codes to SJIS first.

Now let us look at the general algorithm for determining a transmission code set and where conversions are performed. First, we introduce the following abbreviations:

- CNCS - Client Native Code Set;
- CCCS - Client Conversion Code Sets;
- SNCS - Server Native Code Set;
- SCCS - Server Conversion Code Sets; and
- TCS - Transmission Code Set.

The algorithm is as follows:

```
if (CNCS==SNCS)
        TCS = CNCS;                 // no conversion required
else {
        if (elementOf(SNCS,CCCS))
            TCS = SNCS;   // client converts to server's native code set
        else if (elementOf(CNCS,SCCS))
            TCS = CNCS;   // server converts from client's native code set
        else if (intersection(CCCS,SCCS) != emptySet) {
            TCS = oneOf(intersection(CCCS,SCCS));
            // client chooses TCS, from intersection(CCCS,SCCS), that is
            // most preferable to server;
            // client converts from CNCS to TCS and server
            // from TCS to SNCS
        else if (compatible(CNCS,SNCS))
            TCS = fallbackCS;       // fallbacks are UTF-8 (for char data) and
                                    // UTF-16 (for wchar data)
        else
            raise CODESET_INCOMPATIBLE exception;
}
```

The algorithm first checks to see if the client and server native code sets are the same. If they are, then the native code set is used for transmission and no conversion is required. If the native code sets are not the same, then the conversion code sets are examined to see if

1. the client can convert from its native code set to the server's native code set,
2. the server can convert from the client's native code set to its native code set, or

3. transmission through an intermediate conversion code set is possible.

If the third option is selected and there is more than one possible intermediate conversion code set (i.e., the intersection of CCCS and SCCS contains more than one code set), then the one most preferable to the server is selected.[3]

If none of these conversions is possible, then the fallback code set (UTF-8 for **char** data and UTF-16 for **wchar** data— see below) is used. However, before selecting the fallback code set, a compatibility test is performed. This test looks at the character sets encoded by the client and server native code sets. If they are different (e.g., Korean and French), then meaningful communication between the client and server is not possible and a CODESET_INCOMPATIBLE exception is raised. This test is similar to the DCE compatibility test and is intended to catch those cases where conversion from the client native code set to the fallback, and the fallback to the server native code set would result in massive data loss. (See Section 13.9, "Relevant OSFM Registry Interfaces," on page 13-106 for the relevant OSF registry interfaces that could be used for determining compatibility.)

A DATA_CONVERSION exception is raised when a client or server attempts to transmit a character that does not map into the negotiated transmission code set. For example, not all characters in Taiwan Chinese map into Unicode. When an attempt is made to transmit one of these characters via Unicode, an ORB is required to raise a DATA_CONVERSION exception.

In summary, the fallback code set is UTF-8 for **char** data (identified in the Registry as 0x05010001, "X/Open UTF-8; UCS Transformation Format 8 (UTF-8)"), and UTF-16 for **wchar** data (identified in the Registry as 0x00010109, "ISO/IEC 10646-1:1993; UTF-16, UCS Transformation Format 16-bit form"). As mentioned above the fallback code set is meaningful only when the client and server character sets are compatible, and the fallback code set is distinguished from a default code set used for backward compatibility.

If a server's native **char** code set is not specified in the IOR multi-component profile, then it is considered to be ISO 8859-1 for backward compatibility. However, a server that supports interfaces that use wide character data is required to specify its native **wchar** code set; if one is not specified, then the client-side ORB raises exception INV_OBJREF.

Similarly, if no **char** transmission code set is specified in the code set service context, then the **char** transmission code set is considered to be ISO 8859-1 for backward compatibility. If a client transmits wide character data and does not specify its **wchar** transmission code set in the service context, then the server-side ORB raises exception BAD_PARAM.

3.Recall that server conversion code sets are listed in order of preference.

To guarantee “out-of-the-box” interoperability, clients and servers must be able to convert between their native **char** code set and UTF-8 and their native **wchar** code set (if specified) and Unicode. Note that this does not require that all server native code sets be mappable to Unicode, but only those that are exported as native in the IOR. The server may have other native code sets that aren’t mappable to Unicode and those can be exported as SCCSs (but not SNCSs). This is done to guarantee out-of-the-box interoperability and to reduce the number of code set converters that a CORBA-compliant ORB must provide.

ORB implementations are strongly encouraged to use widely-used code sets for each regional market. For example, in the Japanese marketplace, all ORB implementations should support Japanese EUC, JIS and Shift JIS to be compatible with existing business practices.

13.7.3 Mapping to Generic Character Environments

Certain language environments do not distinguish between byte-oriented and wide characters. In such environments both **char** and **wchar** are mapped to the same “generic” character representation of the language. **String** and **wstring** are likewise mapped to generic strings in such environments. Examples of language environments that provide generic character support are Smalltalk and Ada.

Even while using languages that do distinguish between wide and byte-oriented characters (e.g., C and C++), it is possible to mimic some generic behavior by the use of suitable macros and support libraries. For example, developers of Windows NT and Windows 95 applications can write portable code between NT (which uses Unicode strings) and Windows 95 (which uses byte-oriented character strings) by using a set of macros for declaring and manipulating characters and character strings. Appendix A in this chapter shows how to map wide and byte-oriented characters to these generic macros.

Another way to achieve generic manipulation of characters and strings is by treating them as abstract data types (ADTs). For example, if strings were treated as abstract data types and the programmers are required to create, destroy, and manipulate strings only through the operations in the ADT interface, then it becomes possible to write code that is representation-independent. This approach has an advantage over the macro-based approach in that it provides portability between byte-oriented and wide character environments even without recompilation (at runtime the string function calls are bound to the appropriate byte-oriented/wide library). Another way of looking at it is that the macro-based genericity gives compile-time flexibility, while ADT-based genericity gives runtime flexibility.

Yet another way to achieve generic manipulation of character data is through the ANSI C++ Strings library defined as a template that can be parameterized by **`char`**, **`wchar_t`**, or other integer types.

Given that there can be several ways of treating characters and character strings in a generic way, this standard cannot, and therefore does not, specify the mapping of **char**, **wchar, string**, and **wstring** to all of them. It only specifies the following normative requirements which are applicable to generic character environments:

- **wchar** must be mapped to the generic character type in a generic character environment.
- **wstring** must be mapped to a string of such generic characters in a generic character environment.
- The language binding files (i.e., stubs) generated for these generic environments must ensure that the generic type representation is converted to the appropriate code sets (i.e., CNCS on the client side and SNCS on the server side) before character data is given to the ORB runtime for transmission.

13.7.3.1 Describing Generic Interfaces

To describe generic interfaces in IDL we recommend using **wchar** and **wstring**. These can be mapped to generic character types in environments where they do exist and to wide characters where they do not. Either way interoperation between generic and non-generic character type environments is achieved because of the code set conversion framework.

13.7.3.2 Interoperation

Let us consider an example to see how a generic environment can interoperate with a non-generic environment. Let us say there is an IDL interface with both **char** and **wchar** parameters on the operations, and let us say the client of the interface is in a generic environment while the server is in a non-generic environment (for example the client is written in Smalltalk and the server is written in C++).

Assume that the server's (byte-oriented) native **char** code set (SNCS) is eucJP and the client's native **char** code set (CNCS) is SJIS. Further assume that the code set negotiation led to the decision to use eucJP as the **char** TCS-C and Unicode as the **wchar** TCS-W.

As per the above normative requirements for mapping to a generic environment, the client's Smalltalk stubs are responsible for converting all **char** data (however they are represented inside Smalltalk) to SJIS and all **wchar** data to the client's **wchar** code set before passing the data to the client-side ORB. Note that this conversion could be an identity mapping if the internal representation of narrow and wide characters is the same as that of the native code set(s). The client-side ORB now converts all **char** data from SJIS to eucJP and all **wchar** data from the client's **wchar** code set to Unicode, and then transmits to the server side.

The server side ORB and stubs convert the eucJP data and Unicode data into C++'s internal representation for **char**s and **wchar**s as dictated by the IDL operation signatures. Notice that when the data arrives at the server side it does not look any different from data arriving from a non-generic environment (e.g., that is just like the server itself). In other words, the mappings to generic character environments do not affect the code set conversion framework.

13.8 Example of Generic Environment Mapping

This Appendix shows how **char**, **wchar**, **string**, and **wchar** can be mapped to the generic C/C++ macros of the Windows environment. This is merely to illustrate one possibility. This section is not normative and is applicable only in generic environments. See Section 13.7.3, "Mapping to Generic Character Environments," on page 13-104.

13.8.1 Generic Mappings

Char and **string** are mapped to C/C++ **`char`** and **`char*`** as per the standard C/C++ mappings. **wchar** is mapped to the **`TCHAR`** macro which expands to either **`char`** or **`wchar_t`** depending on whether **`_UNICODE`** is defined. **wstring** is mapped to pointers to **`TCHAR`** as well as to the string class **`CORBA::Wstring_var`**. Literal strings in IDL are mapped to the **`_TEXT`** macro as in **`_TEXT(<literal>)`**.

13.8.2 Interoperation and Generic Mappings

We now illustrate how the interoperation works with the above generic mapping. Consider an IDL interface operation with a **wstring** parameter, a client for the operation which is compiled and run on a Windows 95 machine, and a server for the operation which is compiled and run on a Windows NT machine. Assume that the locale (and/or the environment variables for CNCS for **wchar** representation) on the Windows 95 client indicates the client's native code set to be SJIS, and that the corresponding server's native code set is Unicode. The code set negotiation in this case will probably choose Unicode as the TCS-W.

Both the client and server sides will be compiled with **`_UNICODE`** defined. The IDL type **wstring** will be represented as a string of **`wchar_t`** on the client. However, since the client's locale or environment indicates that the CNCS for wide characters is SJIS, the client side ORB will get the **wstring** parameter encoded as a SJIS multi-byte string (since that is the client's native code set), which it will then convert to Unicode before transmitting to the server. On the server side the ORB has no conversions to do since the TCS-W matches the server's native code set for wide characters.

We therefore notice that the code set conversion framework handles the necessary translations between byte-oriented and wide forms.

13.9 Relevant OSFM Registry Interfaces

13.9.1 Character and Code Set Registry

The OSF character and code set registry is defined in *OSF Character and Code Set Registry* (see References in the Preface) and current registry contents may be obtained directly from the Open Software Foundation (obtain via anonymous ftp to ftp.opengroup.org:/pub/code_set_registry). This registry contains two parts: character sets and code sets. For each listed code set, the set of character sets encoded by this code set is shown.

Each 32-bit code set value consists of a high-order 16-bit organization number and a 16-bit identification of the code set within that organization. As the numbering of organizations starts with 0x0001, a code set null value (0x00000000) may be used to indicate an unknown code set.

When associating character sets and code sets, OSF uses the concept of "fuzzy equality," meaning that a code set is shown as encoding a particular character set if the code set can encode "most" of the characters.

"Compatibility" is determined with respect to two code sets by examining their entries in the registry, paying special attention to the character sets encoded by each code set. For each of the two code sets, an attempt is made to see if there is at least one (fuzzy-defined) character set in common, and if such a character set is found, then the assumption is made that these code sets are "compatible." Obviously, applications which exploit parts of a character set not properly encoded in this scheme will suffer information loss when communicating with another application in this "fuzzy" scheme.

The ORB is responsible for accessing the OSF registry and determining "compatibility" based on the information returned.

OSF members and other organizations can request additions to both the character set and code set registries by email to cs-registry@opengroup.org; in particular, one range of the code set registry (**`0xf5000000`** through **`0xffffffff`**) is reserved for organizations to use in identifying sets which are not registered with the OSF (although such use would not facilitate interoperability without registration).

13.9.2 Access Routines

The following routines are for accessing the OSF character and code set registry. These routines map a code set string name to code set id and vice versa. They also help in determining character set compatibility. These routine interfaces, their semantics and their actual implementation are not normative (i.e., ORB vendors do not have to bundle the OSF registry implementation with their products for compliance).

The following routines are adopted from *RPC Runtime Support For I18N Characters - Functional Specification* (see References in the Preface).

13.9.2.1 dce_cs_loc_to_rgy

Maps a local system-specific string name for a code set to a numeric code set value specified in the code set registry.

Synopsis

```
void dce_cs_loc_to_rgy(
   idl_char *local_code_set_name,
   unsigned32 *rgy_code_set_value,
   unsigned16 *rgy_char_sets_number,
   unsigned16 **rgy_char_sets_value,
   error_status_t *status);
```

Parameters

Input

local_code_set_name - A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 data bytes plus a terminating NULL character.

Output

rgy_code_set_value 0 - The registered integer value that uniquely identifies the code set specified by local_code_set_name.

rgy_char_sets_number - The number of character sets that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter.

rgy_char_sets_value - A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL prevents this routine from returning this parameter. The routine dynamically allocates this value.

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- dce_cs_c_ok – Code set registry access operation succeeded.
- dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info.
- dce_cs_c_unknown – No code set value was not found in the registry which corresponds to the code set name specified.
- dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the name specified.

Description

The dce_cs_loc_to_rgy() routine maps operating system-specific names for character/code set encodings to their unique identifiers in the code set registry.

The dce_cs_loc_to_rgy() routine takes as input a string that holds the host-specific “local name” of a code set and returns the corresponding integer value that uniquely identifies that code set, as registered in the host's code set registry. If the integer value does not exist in the registry, the routine returns the status dce_cs_c_unknown.

The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the rgy_char_sets_number and rgy_char_sets_value[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a code set value from the code set registry can specify NULL for these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the array after it is used, since the array is dynamically allocated.

13.9.2.2 *dce_cs_rgy_to_loc*

Maps a numeric code set value contained in the code set registry to the local system-specific name for a code set.

Synopsis

```
void dce_cs_rgy_to_loc(
        unsigned32 *rgy_code_set_value,
        idl_char **local_code_set_name,
        unsigned16 *rgy_char_sets_number,
        unsigned16 **rgy_char_sets_value,
        error_status_t *status);
```

Parameters

Input

rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set.

Output

local_code_set_name - A string that specifies the name that the local host's locale environment uses to refer to the code set. The string is a maximum of 32 bytes: 31 data bytes and a terminating NULL character.

rgy_char_sets_number - The number of character sets that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value.

rgy_char_sets_value - A pointer to an array of registered integer values that uniquely identify the character set(s) that the specified code set encodes. Specifying NULL in this parameter prevents the routine from returning this value. The routine dynamically allocates this value.

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- dce_cs_c_ok – Code set registry access operation succeeded.
- dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info.
- dce_cs_c_unknown – The requested code set value was not found in the code set registry.
- dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the specific code set registry ID value. This implies that the code set is not supported in the local system environment.

Description

The dce_cs_rgy_to_loc() routine maps a unique identifier for a code set in the code set registry to the operating system-specific string name for the code set, if it exists in the code set registry.

The dce_cs_rgy_to_loc() routine takes as input a registered integer value of a code set and returns a string that holds the operating system-specific, or local name, of the code set.

If the code set identifier does not exist in the registry, the routine returns the status dce_cs_c_unknown and returns an undefined string.

The routine also returns the number of character sets that the code set encodes and the registered integer values that uniquely identify those character sets. Specifying NULL in the rgy_char_sets_number and rgy_char_sets_value[] parameters prevents the routine from performing the additional search for these values. Applications that want only to obtain a local code set name from the code set registry can specify NULL for

these parameters in order to improve the routine's performance. If the value is returned from the routine, application developers should free the rgy_char_sets_value array after it is used.

13.9.2.3 *rpc_cs_char_set_compat_check*

Evaluates character set compatibility between a client and a server.

Synopsis

```
void rpc_cs_char_set_compat_check(
        unsigned32 client_rgy_code_set_value,
        unsigned32 server_rgy_code_set_value,
        error_status_t *status);
```

Parameters

Input

client_rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set that the client is using as its local code set.

server_rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set that the server is using as its local code set.

Output

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- rpc_s_ok – Successful status.
- rpc_s_ss_no_compat_charsets – No compatible code set found. The client and server do not have a common encoding that both could recognize and convert.
- The routine can also return status codes from the dce_cs_rgy_to_loc() routine.

Description

The rpc_cs_char_set_compat_check() routine provides a method for determining character set compatibility between a client and a server; if the server's character set is incompatible with that of the client, then connecting to that server is most likely not acceptable, since massive data loss would result from such a connection.

The routine takes the registered integer values that represent the code sets that the client and server are currently using and calls the code set registry to obtain the registered values that represent the character set(s) that the specified code sets support. If both client and server support just one character set, the routine compares client and server registered character set values to determine whether or not the sets are compatible. If they are not, the routine returns the status message rpc_s_ss_no_compat_charsets.

If the client and server support multiple character sets, the routine determines whether at least two of the sets are compatible. If two or more sets match, the routine considers the character sets compatible, and returns a success status code to the caller.

13.9.2.4 *rpc_rgy_get_max_bytes*

Gets the maximum number of bytes that a code set uses to encode one character from the code set registry on a host

Synopsis

```
void rpc_rgy_get_max_bytes(
   unsigned32 rgy_code_set_value,
   unsigned16 *rgy_max_bytes,
   error_status_t *status);
```

Parameters

Input

rgy_code_set_value - The registered hexadecimal value that uniquely identifies the code set.

Output

rgy_max_bytes - The registered decimal value that indicates the number of bytes this code set uses to encode one character.

status - Returns the status code from this routine. This status code indicates whether the routine completed successfully or, if not, why not.

The possible status codes and their meanings are as follows:

- rpc_s_ok – Operation succeeded.
- dce_cs_c_cannot_allocate_memory – Cannot allocate memory for code set info.
- dce_cs_c_unknown – No code set value was not found in the registry which corresponds to the code set value specified.
- dce_cs_c_notfound – No local code set name was found in the registry which corresponds to the value specified.

Description

The rpc_rgy_get_max_bytes() routine reads the code set registry on the local host. It takes the specified registered code set value, uses it as an index into the registry, and returns the decimal value that indicates the number of bytes that the code set uses to encode one character.

This information can be used for buffer sizing as part of the procedure to determine whether additional storage needs to be allocated for conversion between local and network code sets.