Interworking Architecture 15

The Interworking chapters describe a specification for communication between two similar but very distinct object management systems: Microsoft's COM (including OLE) and the OMG's CORBA. An optimal specification would allow objects from either system to make their key functionality visible to clients using the other system as transparently as possible. The architecture for Interworking is designed to meet this goal.

Contents

This chapter contains the following sections.

15.1 Purpose of the Interworking Architecture

The purpose of the Interworking architecture is to specify support for two-way communication between CORBA objects and COM objects. The goal is that objects from one object model should be able to be viewed as if they existed in the other object model. For example, a client working in a CORBA model should be able to view a COM object as if it were a CORBA object. Likewise, a client working in a COM object model should be able to view a CORBA object as if it were a COM object.

There are many similarities between the two systems. In particular, both are centered around the idea that an object is a discrete unit of functionality that presents its behavior through a set of fully-described interfaces. Each system hides the details of implementation from its clients. To a large extent COM and CORBA are semantically isomorphic. Much of the COM/CORBA Interworking specification simply involves a mapping of the syntax, structure and facilities of each to the other — a straightforward task.

There are, however, differences in the CORBA and COM object models. COM and CORBA each have a different way of describing what an object is, how it is typically used, and how the components of the object model are organized. Even among largely isomorphic elements, these differences raise a number of issues as to how to provide the most transparent mapping.

15.1.1 Comparing COM Objects to CORBA Objects

From a COM point of view, an object is typically a subcomponent of an application, which represents a point of exposure to other parts of the application, or to other applications. Many OLE objects are document-centric and are often (though certainly not exclusively) tied to some visual presentation metaphor. Historically, the typical domain of an COM object is a single-user, multitasking visual desktop such as a Microsoft Windows desktop. Currently, the main goal of COM and OLE is to expedite collaboration- and information-sharing among applications using the same desktop, largely through user manipulation of visual elements (for example, drag-and-drop, cut-and-paste).

From a CORBA point of view, an object is an independent component providing a related set of behaviors. An object is expected to be available transparently to any CORBA client regardless of the location (or implementation) of either the object or the client. Most CORBA objects focus on distributed control in a heterogeneous environment. Historically, the typical domain of a CORBA object is an arbitrarily scalable distributed network. In its current form, the main goal of CORBA is to allow these independent components to be shared among a wide variety of applications (and other objects), any of which may be otherwise unrelated.

Of course, CORBA is already used to define desktop objects, and COM can be extended to work over a network. Also, both models are growing and evolving, and will probably overlap in functionally in the future. Therefore, a good interworking model must map the functionality of two systems to each other while preserving the flavor of each system as it is typically presented to a developer.

The most obvious similarity between these two systems is that they are both based architecturally on *objects*. The Interworking Object Model describes the overlap between the features of the CORBA and COM object models, and how the common features map between the two models.

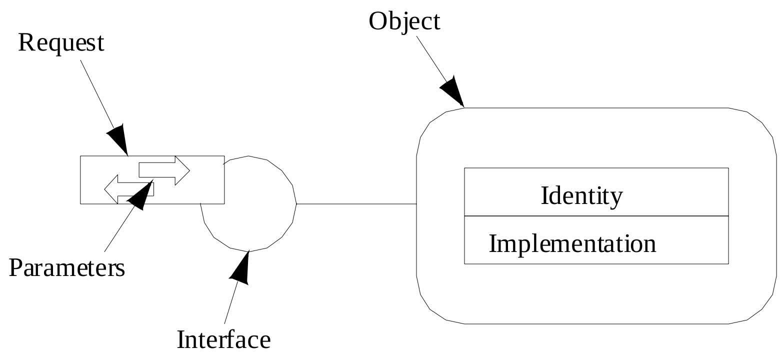

Figure 15-1 Interworking Object Model

15.2 Interworking Object Model

15.2.1 Relationship to CORBA Object Model

In the Interworking Object Model, each object is simply a discrete unit of functionality that presents itself through a published interface described in terms of a well-known, fully-described set of interface semantics. An interface (and its underlying functionality) is accessed through at least one well-known, fully described form of request. Each request in turn targets a specific object—an object instance—based on a reference to its identity. That target object is then expected to service the request by invoking the expected behavior in its own particular implementation. Request parameters are object references or nonobject data values described in the object model's data type system. Interfaces may be composed by combining other interfaces according to some well-defined composition rules. In each object system, interfaces are described in a specialized language or can be represented in some repository or library.

In CORBA, the Interworking Object Model is mapped to an architectural abstraction known as the Object Request Broker (ORB). Functionally, an ORB provides for the registration of the following:

- Types and their interfaces, as described in the OMG Interface Definition Language (OMG IDL).
- Instance identities, from which the ORB can then construct appropriate references to each object for interested clients.

A CORBA object may thereafter receive requests from interested clients that hold its object reference and have the necessary information to make a properly-formed request on the object's interface. This request can be statically defined at compile time or dynamically created at run-time based upon type information available through an interface type repository.

While CORBA specifies the existence of an implementation type description called ImplementationDef (and an Implementation Repository, which contains these type descriptions), CORBA does not specify the interface or characteristics of the Implementation Repository or the ImplementationDef. As such, implementation typing and descriptions vary from ORB to ORB and are not part of this specification.

15.2.2 Relationship to the OLE/COM Model

In OLE, the Interworking Object Model is principally mapped to the architectural abstraction known as the Component Object Model (COM). Functionally, COM allows an object to expose its interfaces in a well-defined binary form (that is, a virtual function table) so that clients with static compile-time knowledge of the interface's structure, and with a reference to an instance offering that interface, can send it appropriate requests. Most COM interfaces are described in Microsoft Interface Definition Language (MIDL).

COM supports an implementation typing mechanism centered around the concept of a COM class. A COM class has a well-defined identity and there is a repository (known as the system registry) that maps implementations (identified by class IDs) to specific executable code units that embody the corresponding implementation realizations.

COM also provides an extension called OLE Automation. Interfaces that are Automation-compatible can be described in Object Definition Language (ODL) and can optionally be registered in a binary Type Library. Automation interfaces can be invoked dynamically by a client having no compile-time interface knowledge through a special COM interface (IDispatch). Run-time type checking on invocations can be implemented when a Type Library is supplied. Automation interfaces have properties and methods, whereas COM interfaces have only methods. The data types that may be used for properties and as method parameters comprise a subset of the types supported in COM, with no support for user-defined constructed types.

Thus, use of and interoperating with objects exposing OLE Automation interfaces is considerably different from other COM objects. Although Automation is implemented through COM, for the purposes of this document, OLE Automation and COM are considered to be distinct object models. Interworking between CORBA and OLE Automation will be described separately from interworking with the basic COM model.

15.2.3 Basic Description of the Interworking Model

Viewed at this very high level, Microsoft's COM and OMG's CORBA appear quite similar. Roughly speaking, COM interfaces (including Automation interfaces) are equivalent to CORBA interfaces. In addition, COM interface pointers are very roughly equivalent to CORBA object references. Assuming that lower-level design details (calling conventions, data types, and so forth) are more or less semantically isomorphic, a reasonable level of interworking is probably possible between the two systems through straightforward mappings.

How such interworking can be practically achieved is illustrated in an Interworking Model, shown in Figure 15-2. It shows how an object in Object System B can be mapped and represented to a client in Object System A. From now on, this will be called a B/A mapping. For example, mapping a CORBA object to be visible to a COM client is a CORBA/COM mapping.

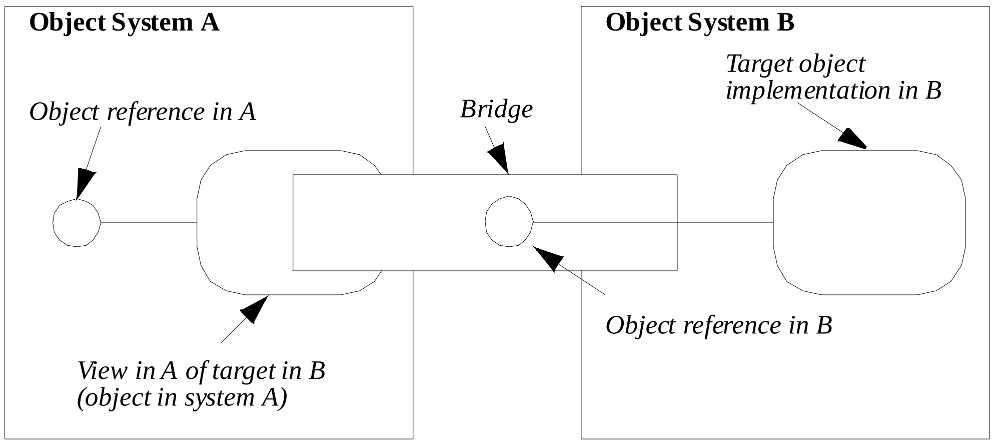
Object System A
Object System B
Object reference in A
Bridge
Target object implementation in B
Object reference in B
View in A of target in B (object in system A)

Figure 15-2 B/A Interworking Model

On the left is a client in object system A, that wants to send a request to a target object in system B, on the right. We refer to the entire conceptual entity that provides the mapping as a bridge. The goal is to map and deliver any request from the client transparently to the target.

To do so, we first provide an object in system A called a View. The View is an object in system A that presents the identity and interface of the target in system B mapped to the vernacular of system A, and is described as an A View of a B target.

The View exposes an interface, called the View Interface, which is isomorphic to the target's interface in system B. The methods of the View Interface convert requests from system A clients into requests on the target's interface in system B. The View is a component of the bridge. A bridge may be composed of many Views.

The bridge maps interface and identify forms between different object systems. Conceptually, the bridge holds a reference in B for the target (although this is not physically required). The bridge must provide a point of rendezvous between A and B, and may be implemented using any mechanism that permits communication between the two systems (IPC, RPC, network, shared memory, and so forth) sufficient to preserve all relevant object semantics.

The client treats the View as though it is the real object in system A, and makes the request in the vernacular request form of system A. The request is translated into the vernacular of object system B, and delivered to the target object. The net effect is that a request made on an interface in A is transparently delivered to the intended instance in B.

The Interworking Model works in either direction. For example, if system A is COM, and system B is CORBA, then the View is called the COM View of the CORBA target. The COM View presents the target's interface to the COM client. Similarly if system A is CORBA and system B is COM, then the View is called the *CORBA View* of the COM target. The CORBA View presents the target's interface to the CORBA client.

Figure 15-3 shows the interworking mappings discussed in the Interworking chapters. They represent the following:

- The mapping providing a COM View of a CORBA target
- The mapping providing a CORBA View of a COM target
- The mapping providing an Automation View of a CORBA target
- The mapping providing a CORBA View of an Automation target

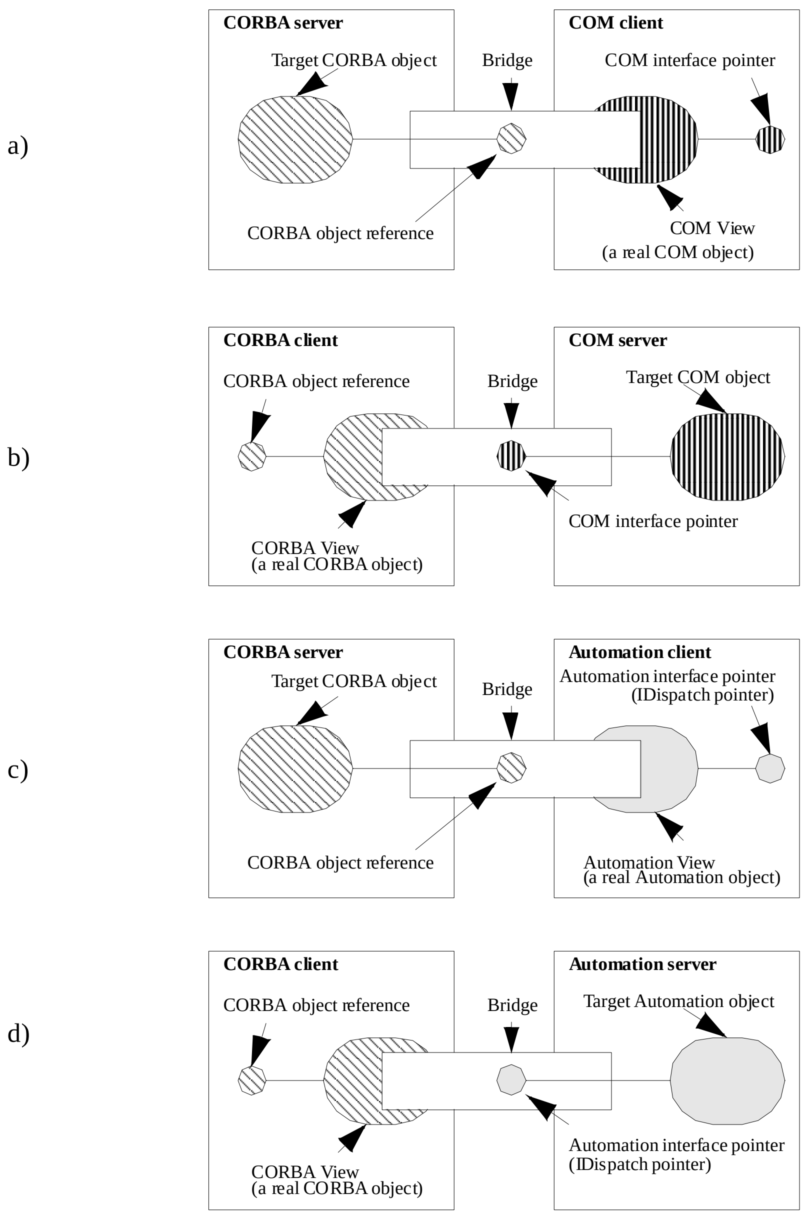

Figure 15-3 Interworking Mapping

Note that the division of the mapping process into these architectural components does not infer any particular design or implementation strategy. For example, a COM View and its encapsulated CORBA reference could be implemented in COM as a single component or as a system of communicating components on different hosts.

The architecture allows for a range of implementation strategies, including, but not limited to generic and interface-specific mapping.

- **Generic Mapping** assumes that all interfaces can be mapped through a dynamic mechanism supplied at run-time by a single set of bridge components. This allows automatic access to new interfaces as soon as they are registered with the target system. This approach generally simplifies installation and change management, but may incur the run-time performance penalties normally associated with dynamic mapping.
- **Interface-Specific Mapping** assumes that separate bridge components are generated for each interface or for a limited set of related interfaces (for example, by a compiler). This approach generally improves performance by “precompiling” request mappings, but may create installation and change management problems.

15.3 Interworking Mapping Issues

The goal of the Interworking specification is to achieve a straightforward two-way (COM/CORBA and CORBA/COM) mapping in conformance with the previously described Interworking Model. However, despite many similarities, there are some significant differences between CORBA and COM that complicate achieving this goal. The most important areas involve:

- **Interface Mapping**. A CORBA interface must be mapped to and from two distinct forms of interfaces, OLE Automation and COM.
- **Interface Composition Mapping**. CORBA multiple inheritance must be mapped to COM single inheritance/aggregation. COM interface aggregation must be mapped to the CORBA multiple inheritance model.
- **Identity Mapping**. The explicit notion of an instance identity in CORBA must be mapped to the more implicit notion of instance identity in COM.
- **Mapping Invertibility**. It may be desirable for the object model mappings to be invertible, but the Interworking specification does not guarantee invertibility in all situations.

15.4 Interface Mapping

The CORBA standard for describing interfaces is OMG IDL. It describes the requests that an object supports. OLE provides two distinct and somewhat disjointed interface models: COM and Automation. Each has its own respective request form, interface semantics, and interface syntax.

Therefore, we must consider the problems and benefits of four distinct mappings:

- CORBA/COM
- CORBA/Automation
- COM/CORBA
- Automation/CORBA

We must also consider the bidirectional impact of a third, hybrid form of interface, the Dual Interface, which supports both an Automation and a COM-like interface. The succeeding sections summarize the main issues facing each of these mappings.

15.4.1 CORBA/COM

There is a reasonably good mapping from CORBA objects to COM Interfaces; for instance:

- OMG IDL primitives map closely to COM primitives.
- Constructed data types (structs, unions, arrays, strings, and enums) also map closely.
- CORBA object references map closely to COM interface pointers.
- Inherited CORBA interfaces may be represented as multiple COM interfaces.
- CORBA attributes may be mapped to get and set operations in COM interfaces.

This mapping is perhaps the most natural way to represent the interfaces of CORBA objects in the COM environment. In practice, however, many COM clients (for example, Visual Basic applications) can only bind to Automation Interfaces and cannot bind to the more general COM Interfaces. Therefore, providing only a mapping of CORBA to the COM Interfaces would not satisfy many COM/OLE clients.

15.4.2 CORBA/Automation

There is a limited fit between OLE Automation objects and CORBA objects:

- Some OMG IDL primitives map directly to Automation primitives. However, there are primitives in both systems (for example, the OLE CURRENCY type and the CORBA unsigned integral types) that must be mapped as special cases (possibly with loss of range or precision).
- OMG IDL constructed types do not map naturally to any Automation constructs. Since such constructed types cannot be passed as argument parameters in Automation interfaces, these must be simulated by providing specially constructed interfaces (for example, viewing a struct as an OLE object with its own interface).
- CORBA Interface Repositories can be mapped dynamically to Automation Type Libraries.
- CORBA object references map to Automation interface pointers.
- There is no clean mapping for multiple inheritance to OLE Automation interfaces. All methods of the multiply-inherited interfaces could be expanded to a single Automation interface; however, this approach would require a total ordering over the methods if [dual] interfaces are to be supported. An alternative approach would be to map multiple inheritance to multiple Automation interfaces. This mapping, however, would require that an interface navigation mechanism be exposed to OLE Automation controllers. Currently OLE Automation does not provide a canonical way for clients (such as Visual Basic) to navigate between multiple interfaces.

- CORBA attributes may be mapped to get and put properties in Automation interfaces.

This form of interface mapping will place some restrictions on the types of argument passing that can be mapped, and/or the cost (in terms of run-time translations) incurred in those mappings. Nevertheless, it is likely to be the most popular form of CORBA-to-COM interworking, since it will provide dynamic access to CORBA objects from Visual Basic and other OLE Automation client development environments.

15.4.3 COM/CORBA

This mapping is similar to CORBA/COM, except for the following:

- Some COM primitive data types (for example, UNICODE long, unsigned long long, and wide char) and constructed types (for example, wide string) are not currently supported by OMG IDL. (These data types may be added to OMG IDL in the future.)
- Some unions, pointer types and the SAFEARRAY type require special handling.

The COM/CORBA mapping is somewhat further complicated, by the following issues:

- Though it is less common, COM objects may be built directly in C and C++ (without exposing an interface specification) by providing custom marshaling implementations. If the interface can be expressed precisely in some COM formalism (MIDL, ODL, or a Type Library), it must first be hand-translated to such a form before any formal mapping can be constructed. If not, the interworking mechanism (such as the View, request transformation, and so forth) must be custom-built.
- MIDL, ODL, and Type Libraries are somewhat different, and some are not supported on certain Windows platforms; for example, MIDL is not available on Win16 platforms.

15.4.4 Automation/CORBA

The Automation interface model and type system are markedly constrained, bounding the size of the problem of mapping from OLE Automation interfaces to CORBA interfaces.

- Automation interfaces and references (IDispatch pointers) map directly to CORBA interfaces and object references.
- Automation request signatures map directly into CORBA request signatures.
- Most of the Automation data types map directly to CORBA data types. Certain Automations types (for example, CURRENCY) do not have corresponding predefined CORBA types, but can easily be mapped onto isomorphic constructed types.
- Automation properties map to CORBA attributes.

15.5 Interface Composition Mappings

CORBA provides a multiple inheritance model for aggregating and extending object interfaces. Resulting CORBA interfaces are, essentially, statically defined either in OMG IDL files or in the Interface Repository. Run-time interface evolution is possible by deriving new interfaces from existing ones. Any given CORBA object reference refers to a CORBA object that exposes, at any point in time, a single most-derived interface in which all ancestral interfaces are joined. The CORBA object model does not support objects with multiple, disjoint interfaces.[1]

In contrast, COM objects expose aggregated interfaces by providing a uniform mechanism for navigating among the interfaces that a single object supports (that is, the QueryInterface method). In addition, COM anticipates that the set of interfaces that an object supports will vary at run-time. The only way to know if an object supports an interface at a particular instant is to ask the object.

OLE Automation objects typically provide all Automation operations in a single "flattened" IDispatch interface. While an analogous mechanism to QueryInterface could be supported in OLE Automation as a standard method, it is not the current use model for OLE Automation services.[2]

15.5.1 CORBA/COM

CORBA multiple inheritance maps into COM interfaces with some difficulty. Examination of object-oriented design practice indicates two common uses of interface inheritance, extending and mixing in. Inheritance may be used to extend an interface linearly, creating a specialization or new version of the inherited interface. Inheritance (particularly multiple inheritance) is also commonly used to mix in a new capability (such as the ability to be stored or displayed) that may be orthogonal to the object's basic application function.

Ideally, extension maps well into a single inheritance model, producing a single linear connection of interface elements. This usage of CORBA inheritance for specialization maps directly to COM; a unique CORBA interface inheritance path maps to a single COM interface vtable that includes all of the elements of the CORBA interfaces in the inheritance path.[3] The use of inheritance to mix in an interface maps well into COM's

1. This is established in the CORBA specification, Chapter 1, Interfaces Section, and in the Object Management Architecture Guide, Section 4.4.7.

2. One can use [dual] interfaces to expose multiple IDispatch interfaces for a given COM co-class. The "Dim A as new Z" statement in Visual Basic 4.0 can be used to invoke a Query-Interface for the Z interface. Many OLE Automation controllers, however, do not use the dual interface mechanism.

3. An ordering is needed over the CORBA operations in an interface to provide a deterministic mapping from the OMG IDL interface to a COM vtable. The current ordering is lexico-graphical by bytes in machine-collating sequence.

aggregation mechanism; each mixed-in inherited interface (or interface graph) maps to a separate COM interface, which can be acquired by invoking QueryInterface with the interface's specific UUID.

Unfortunately, with CORBA multiple inheritance there is no syntactic way to determine whether a particular inherited interface is being extended or being mixed in (or used with some other possible design intent). Therefore it is not possible to make ideal mappings mechanically from CORBA multiply-inherited interfaces to collections of COM interfaces without some additional annotation that describes the intended design. Since extending OMG IDL (and the CORBA object model) to support distinctions between different uses of inheritance is undesirable, alternative mappings require arbitrary decisions about which nodes in a CORBA inheritance graph map to which aggregated COM interfaces, and/or an arbitrary ordering mechanism. The mapping described in Section 13.5.2, Ordering Rules for the CORBA->MIDL Transformation, describes a compromise that balances the need to preserve linear interface extensions with the need to keep the number of resulting COM interfaces manageably small. It satisfies the primary requirement for interworking in that it describes a uniform, deterministic mapping from any CORBA inheritance graph to a composite set of COM interfaces.

COM/CORBA

The features of COM's interface aggregation model can be preserved in CORBA by providing a set of CORBA interfaces that can be used to manage a collection of multiple CORBA objects with different disjoint interfaces as a single composite unit. The mechanism described in OMG IDL in Section 15.4, "Interface Mapping," on page 15-8, is sufficiently isomorphic to allow composite COM interfaces to be uniformly mapped into composite OMG IDL interfaces with no loss of capability.

CORBA/Automation

OLE Automation (as exposed through the IDispatch interface) does not rely on ordering in a virtual function table. The target object implements the IDispatch interface as a mini interpreter and exposes what amounts to a flattened single interface for all operations exposed by the object. The object is not required to define an ordering of the operations it supports.

An ordering problem still exists, however, for dual interfaces. Dual interfaces are COM interfaces whose operations are restricted to the Automation data types. Since these are COM interfaces, the client can elect to call the operations directly by mapping the operation to a predetermined position in a function dispatch table. Since the interpreter is being bypassed, the same ordering problems discussed in the previous section apply for OLE Automation dual interfaces.

Automation/CORBA

OLE Automation interfaces are simple collections of operations, with no inheritance or aggregation issues. Each IDispatch interface maps directly to an equivalent OMG IDL-described interface.

15.5.2 Detailed Mapping Rules

Ordering Rules for the CORBA->MIDL Transformation

- Each OMG IDL interface that does not have a parent is mapped to an MIDL interface deriving from IUnknown.
- Each OMG IDL interface that inherits from a single parent interface is mapped to an MIDL interface that derives from the mapping for the parent interface.
- Each OMG IDL interface that inherits from multiple parent interfaces is mapped to an MIDL interface deriving from IUnknown.
- For each CORBA interface, the mapping for operations precede the mapping for attributes.
- The resulting mapping of operations within an interface are ordered based upon the operation name. The ordering is lexicographic by bytes in machine-collating order.
- The resulting mapping of attributes within an interface are ordered based upon the attribute name. The ordering is lexicographic by bytes in machine-collating order. If the attribute is not read-only, the get_<attribute name> method immediately precedes the set_<attribute name> method.

Ordering Rules for the CORBA->OLE Automation Transformation

- Each OMG IDL interface that does not have a parent is mapped to an ODL interface deriving from IDispatch.
- Each OMG IDL interface that inherits from a single parent interface is mapped to an ODL interface that derives from the mapping for the parent interface.
- Each OMG IDL interface that inherits from multiple parent interfaces is mapped to an ODL interface which derives using single inheritance from the mapping for the first parent interface. The first parent interface is defined as the first interface when the immediate parent interfaces are sorted based upon interface repository id. The order of sorting is lexicographic by bytes in machine-collating order.
- Within an interface, the mapping for operations precede the mapping for attributes.
- An OMG IDL interface’s operations are ordered in the resulting mapping based upon the operation name. The ordering is lexicographic by bytes in machine-collating order.
- An OMG IDL interface’s attributes are ordered in the resulting mapping based upon the attribute name. The ordering is lexicographic by bytes in machine-collating order. For non-read-only attributes, the [propget] method immediately precedes the [propput] method.
- For OMG IDL interfaces that multiply inherit from parent interfaces, the operations introduced in the current interface are mapped first and ordered based on the above rules. After the interface’s operations are mapped, the operations are followed by the ordered operations from the mapping of the parent interfaces (excluding the first interface which was mapped using inheritance).

15.5.3 Example of Applying Ordering Rules

Consider the OMG IDL description shown in Figure 15-4.

```
interface A {// OMG IDL
    void opA();
    attribute long val;
};
interface B : A {
    void opB();
};
interface C: A {
    void opC();
};
interface D : B, C {
    void opD();
};
interface E {
    void opE();
};
interface F : D, E {
    void opF();
};
```

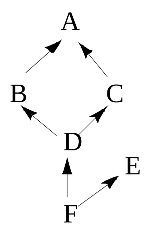

Figure 15-4 OMG IDL Description with Multiple Inheritance

Following the rules in “Detailed Mapping Rules” on page 15-13 the interface description would map to the Microsoft MIDL definition shown in Figure 15-5 and would map to the ODL definition shown in Figure 15-6.

```
[object, uuid(7fc56270-e7a7-0fa8-1d59-35b72eacbe29)]
interface IA : IUnknown{// Microsoft MIDL
    HRESULT opA();
    HRESULT get_val([out] long * val);
    HRESULT set_val([in] long val);
};
[object, uuid(9d5ed678-fe57-bcca-1d41-40957afab571)]
interface IB : IA {
    HRESULT opB();
```

$$\begin{tikzcd}
IU & IU & IU & IU & IU \\
A \arrow[u] & A \arrow[u] & D \arrow[u] & E \arrow[u] & F \arrow[u] \\
B \arrow[u] & C \arrow[u] & & &
\end{tikzcd}$$

```
};
[object,uuid(0d61f837-0cad-1d41-1d40-b84d143e1257)]
interface IC: IA {
    HRESULT opC();
};
[object, uuid(f623e75a-f30e-62bb-1d7d-6df5b50bb7b5)]
interface ID : IUnknown {
    HRESULT opD();
};
[object, uuid(3a3ea00c-fc35-332c-1d76-e5e9a32e94da)]
interface IE : IUnknown{
    HRESULT opE();
};
[object, uuid(80061894-3025-315f-1d5e-4e1f09471012)]
interface IF : IUnknown {
    HRESULT opF();
};
```

Figure 15-5 MIDL Description

```
[uuid(7fc56270-e7a7-0fa8-1dd9-35b72eacbe29),
oleautomation, dual]
interface DA : IDispatch {                                          //
Microsoft ODL
    HRESULT opA([out, optional] VARAINT* v);
    [propget]
    HRESULT val([out] long *val);
    [propset]
    HRESULT val([in] long val);
};
[uuid(9d5ed678-fe57-bcca-1dc1-40957afab571),
oleautomation,dual]
interface DB : DA {
    HRESULT opB([out, optional]VARIANT * v);
};
[uuid(0d61f837-0cad-1d41-1dc0-b84d143e1257),
oleautomation, dual]
interface DC: DA {
    HRESULT opC([out, optional]VARIANT *v);
};
[uuid(f623e75a-f30e-62bb-1dfd-6df5b50bb7b5),
oleautomation, dual]
interface DD : DB {
    HRESULT opD([out, optional]VARIANT *v);
    HRESULT opC([out, optional] VARIANT *v);
};
[uuid(3a3ea00c-fc35-332c-1df6-e5e9a32e94da),
oleautomation, dual]
interface DE : IDispatch{
    HRESULT opE([out, optional] VARIANT *v);
};
[uuid(80061894-3025-315f-1dde-4e1f09471012)
oleautomation, dual]
interface DF : DD {
    HRESULT opF([out, optional] VARIANT *v);
    HRESULT opE([out, optional] VARIANT *v);
};
```

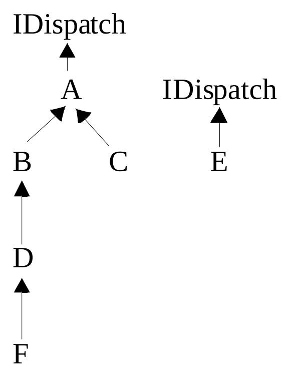
$$\begin{tikzcd}
 & \text{IDispatch} & & \\
 & A \arrow[u] & & \text{IDispatch} \\
B \arrow[ur] & & C \arrow[ul] & E \arrow[u] \\
D \arrow[u] & & & \\
F \arrow[u] & & &
\end{tikzcd}$$

Figure 15-6 Example: ODL Mapping for Multiple Inheritance

15.5.4 *Mapping Interface Identity*

This specification enables interworking solutions from different vendors to interoperate across client/server boundaries (for example, a COM View created by product A can invoke a CORBA server created with product B, given that they both share the same IDL interface). To interoperate in this way, all COM Views mapped from a particular CORBA interface must share the same COM Interface IDs. This section describes a uniform mapping from CORBA Interface Repository IDs to COM Interface IDs.

Mapping Interface Repository IDs to COM IIDs

A CORBA Repository ID is mapped to a corresponding COM Interface ID using a derivative of the RSA Data Security, Inc. MD5 Message-Digest algorithm.[4,5] The repository ID of the CORBA interface is fed into the MD5 algorithm to produce a 128-bit hash identifier. The least significant byte is byte 0 and the most significant byte is byte 8. The resulting 128 bits are modified as follows.

Note – The DCE UUID space is currently divided into four main groups:
byte 8 = 0xxxxxxx (the NCS1.4 name space)
10xxxxxx (A DCE 1.0 UUID name space)
110xxxxx (used by Microsoft)
1111xxxx (Unspecified)

For NCS1.5, the other bits in byte 8 specify a particular family. Family 29 will be assigned to ensure that the autogenerated IIDs do not interfere with other UUID generation techniques.

The upper two bits of byte 9 will be defined as follows.

```
00 unspecified
01generated COM IID
10generated Automation IID
11generated dual interface Automation ID
```

Note – These bits should never be used to determine the type of interface. They are used only to avoid collisions in the name spaces when generating IIDs for multiple types of interfaces — dual, COM, or Automation.

The other bits in the resulting key are taken from the MD5 message digest (stored in the UUID with little endian ordering).

The IID generated from the CORBA repository ID will be used for a COM view of a CORBA interface except when the repository ID is a DCE UUID and the IID being generated is for a COM interface (not Automation or dual). In this case, the DCE UUID will be used as the IID instead of the IID generated from the repository ID (this is done to allow CORBA server developers to implement existing COM interfaces).

This mechanism requires no change to IDL. However, there is an implicit assumption that repository IDs should be unique across ORBs for different interfaces and identical across ORBs for the same interface.

Note – This assumption is also necessary for IIOP to function correctly across ORBs.

4. Rivest, R. “The MD5 Message-Digest Algorithm,” RFC 1321, MIT and RSA Data Security, Inc., April 1992.

Mapping COM IIDs to CORBA Interface IDs

The mapping of a COM IID to the CORBA interface ID is vendor specific. However, the mapping should be the same as if the CORBA mapping of the COM interface were defined with the #pragma ID <interface_name> = "DCE:...".

Thus, the MIDL definition

```
[uuid(f4f2f07c-3a95-11cf-affb-08000970dac7), object]
interface A: IUnknown {
...
}
```

maps to this OMG IDL definition:

```
interface A {
#pragma ID A="DCE:f4f2f07c-3a95-11cf-affb-08000970dac7"
...
};
```

15.6 Object Identity, Binding, and Life Cycle

The interworking model illustrated in Figure 13-2 and Figure 13-3 maps a View in one object system to a reference in the other system. This relationship raises questions:

- How do the concepts of object identity and object life cycle in different object models correspond, and to the extent that they differ, how can they be appropriately mapped?
- How is a View in one system bound to an object reference (and its referent object) in the other system?

15.6.1 Object Identity Issues

COM and CORBA have different notions of what object identity means. The impact of the differences between the two object models affects the transparency of presenting CORBA objects as COM objects or COM objects as CORBA objects. The following sections discuss the issues involved in mapping identities from one system to another. They also describe the architectural mechanics of identity mapping and binding.

5. MD5 was chosen as the hash algorithm because of its uniformity of distribution of bits in the hash value and its popularity for creating unique keys for input text. The algorithm is designed such that on average, half of the output bits change for each bit change in the input. The original algorithm provides a key with uniform distribution in 128 bits. The modification used in this specification selects 118 bits. With a uniform distribution, the probability of drawing k distinct keys (using k distinct inputs) is $n!/((n-k)!*n^k)$, where n is the number of distinct key values (i.e., $n=2^{118}$). If a million (i.e., $k=10^6$) distinct interface repository IDs are passed through the algorithm, the probability of a collision in any of the keys is less than 1 in 10^{23}.

CORBA Object Identity and Reference Properties

CORBA defines an object as a combination of state and a set of methods that explicitly embodies an abstraction characterized by the behavior of relevant requests. An object reference is defined as a name that reliably and consistently denotes a particular object. A useful description of a particular object in CORBA terms is an entity that exhibits a consistency of interface, behavior, and state over its lifetime. This description may fail in many boundary cases, but seems to be a reasonable statement of a common intuitive notion of object identity.

Other important properties of CORBA objects include the following:

- Objects have opaque identities that are encapsulated in object references.
- Object identities are unique within some definable reference domain, which is at least as large as the space spanned by an ORB instance.
- Object references reliably denote a particular object; that is, they can be used to identify and locate a particular object for the purposes of sending a request.
- Identities are immutable, and persist for the lifetime of the denoted object.
- Object references can be used as request targets irrespective of the denoted object's state or location; if an object is passively stored when a client makes a request on a reference to the object, the ORB is responsible for transparently locating and activating the object.
- There is no notion of "connectedness" between object reference and object, nor is there any notion of reference counting.
- Object references may be externalized as strings and reinternalized anywhere within the ORB's reference domain.
- Two object references may be tested for equivalence (that is, to determine whether both references identify the same object instance), although only a result of TRUE for the test is guaranteed to be reliable.

COM Object Identity and Reference Properties

The notion of what it means to be "a particular COM object" is somewhat less clearly defined than under CORBA. In practice, this notion typically corresponds to an active instance of an implementation, but not a particular persistent state. A COM instance can be most precisely defined as "the entity whose interface (or rather, one of whose interfaces) is returned by an invocation of **`IClassFactory::CreateInstance`**." The following observations may be made regarding COM instances:

- COM instances are either initialized with a default "empty" state (e.g., a document or drawing with no contents), or they are initialized to arbitrary states; **`IClassFactory::CreateInstance`** has no parameters for describing initial state.
- The only inherently available identity or reference for a COM instance is its collection of interface pointers. Their usefulness for determining identity equivalence is limited to the scope and extent of the process they live in. There is no

canonical information model, visible or opaque, that defines the identity of a COM object. Individual COM class types may establish a strong notion of persistent identity, but this is not the responsibility of the COM model itself.

- There is no inherent mechanism to determine whether two interface pointers belong to the same COM class or not.
- The identity and management of state are generally independent of the identity and life cycle of COM class instances. Files that contain document state are persistent, and are identified within the file system's name space. A single COM instance of a document type may load, manipulate, and store several different document files during its lifetime; a single document file may be loaded and used by multiple COM class instances, possibly of different types. Any relationship between a COM instance and a state vector is either an artifact of the particular class type, or the user's imagination.

15.6.2 Binding and Life Cycle

The identity-related issues previously discussed emerge as practical problems in defining binding and life cycle management mechanisms in the Interworking models. Binding refers to the way in which an existing object in one system can be located by clients in the other system and associated with an appropriate View. Life cycle, in this context, refers to the way objects in one system are created and destroyed by clients in the other system.

Lifetime Comparison

The in-memory lifetime of COM (including Automation) objects is bounded by the lifetimes of its clients. That is, in COM, when there are no more clients attached to an object, it is destroyed. If clients remain, the object cannot be removed from memory. Unfortunately, a reference counted lifecycle model such as COM's has problems when applied to wide area networks, when network traffic is heavy, and when networks and routers are not fault tolerant (and thus not 100% reliable). For example, if the network connection between clients and the server object were down, the server would think that its clients had died, and would delete itself (if there were no local references to it). When the network connection was later restored, even just seconds later, the clients would then have invalid object references and would need to be restarted, or be prepared to handle invalid interface reference errors for the previously valid references. In addition, if clients exist for a server object but rarely use it, the server object is still required to be in memory. In large, long-running distributed systems, this type of memory consuming behavior is not typically acceptable.

In contrast, the CORBA Life Cycle model decouples the lifetime of the clients from the lifetime of the active (in-memory) representation of the persistent server object. The CORBA model allows clients to maintain references to CORBA server objects even when the clients are no longer running. Server objects can deactivate and remove themselves from memory whenever no clients are currently using them. This behavior avoids the problems and limitations introduced by distributed reference counting. Clients can be started and stopped without incurring expensive data reloads in the server. Servers can relinquish memory (but can later be restored) if they have not been used recently or if the network connection is down. In addition, since the client and server lifetimes are decoupled, CORBA, unlike COM, has no requirement for the servers to constantly "ping" their clients -- a requirement of distributed reference counting which can become expensive across local networks and impractical across wide area networks.

Binding Existing CORBA Objects to COM Views

COM and Automation have limited mechanisms for registering and accessing active objects. A single instance of a COM class can be registered in the active object registry. COM or Automation clients can obtain an IUnknown pointer for an active object with the COM GetActiveObject function or the Automation GetObject function. The most natural way for COM or Automation clients to access existing CORBA objects is through this (or some similar) mechanism.

Interworking solutions can, if desirable, create COM Views for any CORBA object and place them in the active object registry, so that the View (and thus, the object) can be accessed through GetActiveObject or GetObject.

The resources associated with the system registry are limited; some interworking solutions will not be able to map objects efficiently through the registry. This submission defines an interface, ICORBAFactory, which allows interworking solutions to provide their own name spaces through which CORBA objects can be made available to COM and Automation clients in a way that is similar to OLE's native mechanism (GetObject). This interface is described fully in Section 13.7.3, ICORBAFactory Interface.

Binding COM Objects to CORBA Views

As described in "Object Identity Issues" on page 15-18, COM class instances are inherently transient. Clients typically manage COM and Automation objects by creating new class instances and subsequently associating them with a desired stored state. Thus, COM objects are made available through factories. The SimpleFactory OMG IDL interface (described next in "SimpleFactory Interface" on page 15-23) is designed to map onto COM class factories, allowing CORBA clients to create (and bind to) COM objects. A single CORBA SimpleFactory maps to a single COM class factory. The manner in which a particular interworking solution maps SimpleFactories to COM class factories is not specified. Moreover, the manner in which mapped SimpleFactory objects are presented to CORBA clients is not specified.

COM View of CORBA Life Cycle

The SimpleFactory interface provides a create operation without parameters. CORBA SimpleFactory objects can be wrapped with COM IClassFactory interfaces and registered in the Windows registry. The process of building, defining, and registering the factory is implementation-specific.

To allow COM and Automation developers to benefit from the robust CORBA lifecycle model, the following rules apply to COM and Automation Views of CORBA objects. When a COM or Automation View of a CORBA object is dereferenced and there are no longer any clients for the View, the View may delete itself. It should not, however, delete the CORBA object that it refers to. The client of the View may call the **`LifeCycleObject::remove`** operation (if the interface is supported) on the CORBA object to remove it. Otherwise, the lifetime of the CORBA object is controlled by the implementation-specific lifetime management process.

COM currently provides a mechanism for client-controlled persistence of COM objects (equivalent to CORBA externalization). However, unlike CORBA, COM currently provides no general-purpose mechanism for clients to deal with server objects, such as databases, which are inherently persistent (i.e. they store their own state -- their state is not stored through an outside interface such as IPersistStorage). COM does provide monikers, which are conceptually equivalent to CORBA persistent object references. However, monikers are currently only used for OLE graphical linking. To enable COM developers to use CORBA objects to their fullest extent, the submission defines a mechanism that allows monikers to be used as persistent references to CORBA objects, and a new COM interface, IMonikerProvider, that allows clients to obtain an IMoniker interface pointer from COM and Automation Views. The resulting moniker encapsulates, stores, and loads the externalized string representation of the CORBA reference managed by the View from which the moniker was obtained. The IMonkierProvider interface and details of object reference monikers are described in "IMonikerProvider Interface and Moniker Use" on page 15-23.

CORBA View of COM/Automation Life Cycle

Initial references to COM and Automation objects can be obtained in the following way: COM IClassFactories can be wrapped with CORBA SimpleFactory interfaces. These SimpleFactory Views of COM IClassFactories can then be installed in the naming service or used via factory finders. The mechanisms used to register or dynamically look up these factories is beyond the scope of this specification.

All CORBA Views for COM and Automation objects support the LifeCycleObject interface. In order to destroy a View for a COM or Automation object, the remove method of the LifeCycleObject interface must be called. Once a CORBA View is instantiated, it must remain active (in memory) for the lifetime of the View unless the COM or Automation objects supports the IMonikerProvider interface. If the COM or Automation object supports the IMonikerProvider interface, then the CORBA View can safely be deactivated and reactivated provided it stores the object's moniker in persistent storage between activations. Interworking solutions are not required to support deactivation and activation of CORBA View objects, but are enabled to do so by the IMonikerProvider interface.

15.7 Interworking Interfaces

15.7.1 SimpleFactory Interface

CORBA allows object factories to be arbitrarily defined. In contrast, COM IClassFactory is limited to having only one object constructor and the object constructor method (called CreateInstance) has no arguments for passing data during the construction of the instance. The SimpleFactory interface allows CORBA objects to be created under the rigid factory model of COM. The interface also supports CORBA Views of COM class factories.

```
module CosLifeCycle
{
    interface SimpleFactory
    {
        Object create_object();
    };
};
```

SimpleFactory provides a generic object constructor for creating instances with no initial state. In the future, CORBA objects, which can be created with no initial state, should provide factories, which implement the SimpleFactory interface.

15.7.2 IMonikerProvider Interface and Moniker Use

COM or Automation Views for CORBA objects may support the IMonikerProvider interface. COM clients may use QueryInterface for this interface.

```
[object, uuid(ecce76fe-39ce-11cf-8e92-08000970dac7)] // MIDL
interface IMonikerProvider: IUnknown {
   HRESULT get_moniker([out] IMoniker ** val);
}
```

This allows COM clients to persistently save the object reference for later use without needing to keep the View in memory. The moniker returned by IMonikerProvider must support at least the IMoniker and IPersistStorage interfaces. To allow CORBA object reference monikers to be created with one COM/CORBA interworking solution and later restored using another, **`IPersist::GetClassID`** must return the following CLSID:

```
{a936c802-33fb-11cf-a9d1-00401c606e79}
```

In addition, the data stored by the moniker’s IPersistStorage interface must be four 0 (null) bytes followed by the length in bytes of the stringified IOR (stored as a little endian 4-byte unsigned integer value) followed by the stringified IOR itself (without null terminator).

15.7.3 ICORBAFactory Interface

All interworking solutions that expose COM Views of CORBA objects shall expose the ICORBAFactory interface. This interface is designed to support general, simple mechanisms for creating new CORBA object instances and binding to existing CORBA object references by name.

```
interface ICORBAFactory: IUnknown
{
   HRESULT CreateObject( [in] LPTSTR factoryName, [out,
retval] IUknown ** val);
   HRESULT GetObject([in] LPTSTR objectName, [out, retval]
IUknown ** val);
}
```

The UUID for the ICORBAFactory interface is:

```
{204F6240-3AEC-11cf-BBFC-444553540000}
```

A COM class implementing ICORBAFactory must be registered in the Windows System Registry on the client machine using the following class id, class id tag, and Program Id respectively:

```
   {913D82C0-3B00-11cf-BBFC-444553540000}
   DEFINE_GUID(IID_ICORBAFactory,
   0x913d82c0, 0x3b00, 0x11cf, 0xbb, 0xfc, 0x44, 0x45, 0x53,
0x54, 0x0, 0x0);
   “CORBA.Factory.COM”
```

The CORBA factory object may be implemented as a singleton object, i.e., subsequent calls to create the object may return the same interface pointer.

We define a similar interface, DICORBAFactory, that supports creating new CORBA object instances and binding to existing CORBA objects for OLE Automation clients. DICORBAFactory is an Automation Dual Interface. (For an explanation of Automation Dual interfaces, see the Mapping: OLE Automation and CORBA chapter.)

```
interface DICORBAFactory: IDispatch
{
   HRESULT CreateObject( [in] BSTR factoryName, [out,
      retval] IDispatch ** val);
   HRESULT GetObject([in] BSTR objectName, [out, retval]
      IDispatch ** val);
}
```

The UUID for the DICORBAFactory interface is:

```
{204F6241-3AEC-11cf-BBFC-444553540000}
```

An instance of this class must be registered in the Windows System Registry by calling on the client machine using the Program Id “CORBA.Factory.”

The CreateObject and GetObject methods are intended to approximate the usage model and behavior of the Visual Basic CreateObject and GetObject functions.

The first method, CreateObject, causes the following actions:

- A COM View is created. The specific mechanism by which it is created is undefined. We note here that one possible (and likely) implementation is that the View delegates the creation to a registered COM class factory.
- A CORBA object is created and bound to the View. The argument, factoryName, identifies the type of CORBA object to be created. Since the CreateObject method does not accept any parameters, the CORBA object must either be created by a null factory (a factory whose creation method requires no parameters), or the View must supply its own factory parameters internally.
- The bound View is returned to the caller.

The factoryName parameter identifies the type of CORBA object to be created, and thus implicitly identifies (directly or indirectly) the interface supported by the View. In general, the factoryName string takes the form of a sequence of identifiers separated by period characters ("."), such as "personnel.record.person". The intent of this name form is to provide a mechanism that is familiar and natural for COM and OLE Automation programmers by duplicating the form of OLE ProgIDs. The specific semantics of name resolution are determined by the implementation of the interworking solution. The following examples illustrate possible implementations:

- The factoryName sequence could be interpreted as a key to a CosNameService-based factory finder. The CORBA object would be created by invoking the factory create method. Internally, the interworking solution would map the factoryName onto the appropriate COM class ID for the View, create the View, and bind it to the CORBA object.
- The creation could be delegated directly to a COM class factory by interpreting the factoryName as a COM ProgID. The ProgID would map to a class factory for the COM View, and the View's implementation would invoke the appropriate CORBA factory to create the CORBA server object.

The GetObject method has the following behavior:

- The objectName parameter is mapped by the interworking solution onto a CORBA object reference. The specific mechanism for associating names with references is not specified. In order to appear familiar to COM and Automation users, this parameter shall take the form of a sequence of identifiers separated by periods (.), in the same manner as the parameter to CreateObject. An implementation could, for example, choose to map the objectName parameter to a name in the OMG Naming Service implementation. Alternatively, an interworking solution could choose to put precreated COM Views bound to specific CORBA object references in the active object registry, and simply delegate GetObject calls to the registry.
- The object reference is bound to an appropriate COM or Automation View and returned to the caller.

Another name form that is specialized to CORBA is a single name with a preceding period, such as ".NameService". When the name takes this form, the Interworking solution shall interpret the identifier (without the preceding period) as a name in the ORB Initialization interface. Specifically, the name shall be used as the parameter to an invocation of the **CORBA::ORB::ResolveInitialReferences** method on the ORB pseudo-object associated with the ICORBAFactory. The resulting object reference is bound to an appropriate COM or Automation View, which is returned to the caller.

15.7.4 IForeignObject Interface

As object references are passed back and forth between two different object models through a bridge, and the references are mapped through Views (as is the case in this specification), the potential exists for the creation of indefinitely long chains of Views that delegate to other Views, which in turn delegate to other Views, and so on. To avoid this, the Views of at least one object system must be able to expose the reference for the "foreign" object managed by the View. This exposure allows other Views to determine whether an incoming object reference parameter is itself a View and, if so, obtain the "foreign" reference that it manages. By passing the foreign reference directly into the foreign object system, the bridge can avoid creating View chains.

This problem potentially exists for any View representing an object in a foreign object system. The IForeignObject interface is specified to provide bridges access to object references from foreign object systems that are encapsulated in proxies.

```
typedef struct {
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [ size_is(cbMaxSize), length_is(cbLengthUsed), unique ]
long *pValue;
} objSystemIDs;
interface IForeignObject : IUnknown {
   HRESULT GetForeignReference([in[ objSystemIDs systemIDs,
      [out] long *systemID,
      [out] LPSTR* objRef);
   HRESULT GetRepositoryId([out] RepositoryId
      *repositoryId);
}
```

The UUID for IForeignObject is:

```
{204F6242-3AEC-11cf-BBFC-444553540000}
```

The first parameter (systemIDs) is an array of long values that correspond to specific object systems. These values must be positive, unique, and publicly known. The OMG will manage the allocation of identifier values in this space to guarantee uniqueness. The value for the CORBA object system is the long value 1. The systemIDs array contains a list of IDs for object systems for which the caller is interested in obtaining a reference. The order of IDs in the list indicates the caller's order of preference. If the View can produce a reference for at least one of the specified object systems, then the second parameter (systemID) is the ID of the first object system in the incoming array that it can satisfy. The objRef out parameter will contain the object reference converted to a string form. Each object system is responsible for providing a mechanism to convert its references to strings, and back into references. For the CORBA object system, the string contains the IOR string form returned by **`CORBA::ORB::object_to_string`**, as defined in the CORBA specification.

The choice of object reference strings is motivated by the following observations:

- Language mappings for object references do not prescribe the representation of object references. Therefore, it is impossible to reliably map any given ORB's object references onto a fixed OLE Automation parameter type.
- The object reference being returned from GetForeignObject may be from a different ORB than the caller. IORs in string form are the only externalized standard form of object reference supported by CORBA.

The purpose of the GetRepositoryID method is to support the ability of DICORBAAny (see "Mapping for anys" on page 17-24) when it wraps an object reference, to produce a type code for the object when asked to do so via DICORBAAny's readonly typeCode property.

It is not possible to provide a similar inverse interface exposing COM references to CORBA clients through CORBA Views, because of limitations imposed by COM's View of object identity and use of interface pointer as references.

15.7.5 ICORBAObject Interface

The ICORBAObject interface is a COM interface that is exposed by COM Views, allowing COM clients to have access to operations on the CORBA object references, defined on the **`CORBA::Object`** pseudo-interface. The ICORBAObject interface can be obtained by COM clients through QueryInterface. ICORBAObject is defined as follows:

```
interface ICORBAObject: IUnknown
{
   HRESULT GetInterface([out] IUnknown ** val);
   HRESULT GetImplementation([out] IUnknown ** val);
   HRESULT IsA([in] LPTSTR repositoryID, [out] boolean);
   HRESULT IsNil([out] boolean *val);
   HRESULT IsEquivalent([in] IUnknown* obj,[out] boolean *
val);
   HRESULT NonExistent([out] boolean *val);
   HRESULT Hash([out] long *val);
}
```

The UUID for ICORBAObject is:

`{204F6243-3AEC-11cf-BBFC-444553540000}`

Automation controllers gain access to operations on the CORBA object reference interface through the Dual Interface **DIORBObject::GetCORBAObject** method described next.

```
interface DICORBAObject: IDispatch
{
   HRESULT GetInterface([out, retval] IDispatch ** val);
   HRESULT GetImplementation([out, retval] IDispatch **
               val);
   HRESULT IsA([in] BSTR repositoryID, [out, retval]
      boolean);
   HRESULT IsNil([out, retval] boolean *val);
   HRESULT IsEquivalent([in] IDispatch* obj,[out,retval]
         boolean * val);
   HRESULT NonExistent([out,retval] boolean *val);
   HRESULT Hash([out, retval] long *val);
}
```

The UUID for DICORBAObject is:

```
{204F6244-3AEC-11cf-BBFC-444553540000}
```

15.7.6 IORBObject Interface

The IORBObject interface provides Automation and COM clients with access to the operations on the ORB pseudo-object.

The IORBObject is defined as follows:

```
typedef struct {
   unsigned long cbMaxSize;
   unsigned long cbLengthUsed;
   [ size_is(cbMaxSize), length_is(cbLengthUsed), unique ]
   LPSTR *pValue;
} CORBA_ORBObjectIdList;
interface IORBObject : IUnknown
   HRESULT ObjectToString([in] IUnknown* obj, [out] LPSTR
   *val);
   HRESULT StringToObject([in] LPTSTR ref, [out] IUnknown
   *val);
   HRESULT GetInitialReferences([out], CORBA_ORBObjectIdList
   *val);
   HRESULT ResolveInitialReference([in] LPTSTR name, [out]
   IUnknown ** val));
}
```

The UUID for IORBObject is:

```
{204F6245-3AEC-11cf-BBFC-444553540000}
```

A reference to this interface is obtained by calling
ICORBAFactory::GetObject(”CORBA.ORB.2”).

The methods of DIORBObject delegate their function to the similarly-named operations on the ORB pseudo-object associated with the IORBObject.

Automation clients access operations on the ORB via the following Dual Interface:

```
interface DIORBObject: IDispatch {
   HRESULT ObjectToString([in] IDispatch* obj, [out,retval]
   BSTR *val);
   HRESULT StringToObject([in] BSTR ref, [out,retval]
   IDispatch * val);
   HRESULT GetInitialReferences([out, retval]
   SAFEARRAY(IDispatch *) *val);
   HRESULT ResolveInitialReference([in] BSTR name, [out,
   retval] IDispatch ** val));
   HRESULT GetCORBAObject([in] IDispatch* obj, [out, retval]
   DICORBAObject * val);
}
```

The UUID for DIORBObject is:

```
{204F6246-3AEC-11cf-BBFC-444553540000}
```

A reference to this interface is obtained by calling
DICORBAFactory::GetObject(”CORBA.ORB.2”).

This interface is very similar to IORBObject, except for the additional method GetCORBAObject. This method returns an IDispatch pointer to the DICORBAObject interface associated with the parameter Object. This operation is primarily provided to allow Automation controllers (i.e., Automation clients) that cannot invoke QueryInterface on the View object to obtain the ICORBAObject interface.

15.7.7 Naming Conventions for View Components

Naming the COM View Interface Id

The default tag for the COM View’s Interface Id (IID) should be:

```
IID_I<module name>_<interface name>
```

For example, if the module name is “MyModule” and the interface name is “MyInterface” then the default IID tag should be:

```
IID_IMyModule_MyInterface
```

If the module containing the interface is itself nested within other modules, the default tag should be:

```
IID_I<module name>_<module name>_...<module name>_<interface
name>
```

where the module names read from outermost on the left to innermost on the right. Extending our example, if module “MyModule” were nested within module “OuterModule,” then the default IID tag shall be:

```
IID_IOuterModule_MyModule_MyInterface
```

Tag for the Automation Interface Id

No standard tag is required for Automation and Dual Interface IDs because client programs written in Automation controller environments such as Visual Basic are not expected to explicitly use the UUID value.

Naming the COM View Interface

The default name of the COM View’s Interface should be:

```
I<module name>_<interface name>
```

For example, if the module name is “MyModule” and the interface name is “MyInterface,” then the default name should be:

```
IMyModule_MyInterface
```

If the module containing the interface is itself nested within other modules, the default name should be:

```
I<module name>_<module name>_...<module name>_<interface
name>
```

where the module names read from outermost on the left to innermost on the right. Extending our example, if module “MyModule” were nested within module “OuterModule,” then the default name shall be:

```
IOuterModule_MyModule_MyInterface
```

Naming the Automation View Dispatch Interface

The default name of the Automation View’s Interface should be:

```
D<module name>_<interface name>
```

For example, if the module name is “MyModule” and the interface name is “MyInterface,” then the default name should be:

`DMyModule_MyInterface`

If the module containing the interface is itself nested within other modules, the default name should be:

`D<module name>_<module name>_...<module name>_<interface name>`

where the module names read from outermost on the left to innermost on the right. Extending our example, if module “MyModule” were nested within module “OuterModule,” then the default name shall be:

`DOuterModule_MyModule_MyInterface`

Naming the Automation View Dual Interface

The default name of the Automation Dual View’s Interface should be:

`DI<module name>_<interface name>`

For example, if the module name is “MyModule” and the interface name is “MyInterface,” then the default name should be:

`DIMyModule_MyInterface`

If the module containing the interface is itself nested within other modules, the default name should be:

`DI<module name>_<module name>_...<module name>_<interface name>`

where the module names read from outermost on the left to innermost on the right. Extending our example, if module “MyModule” were nested within module “OuterModule,” then the default name shall be:

`DIOuterModule_MyModule_MyInterface`

Naming the Program Id for the COM Class

If a separate COM class is registered for each View Interface, then the default Program Id for that class shall be:

`<module name> “.” <module name> “.” ...<module name> “.” <interface name>`

where the module names read from outermost on the left to innermost on the right. In our example, the default Program Id shall be:

```
"OuterModule.MyModule.MyInterface"
```

Naming the Class Id for the COM Class

If a separate COM co-class is registered for each Automation View Interface, then the default tag for the COM Class Id (CLSID) for that class should be:

```
CLSID_<module name>_<module name>_...<module name>_
<interface name>
```

where the module names read from outermost on the left to innermost on the right. In our example, the default CLSID tag should be:

```
CLSID_OuterModule_MyModule_MyInterface
```

15.8 Distribution

The version of COM (and OLE) that is addressed in this specification (OLE 2.0 in its currently released form) does not include any mechanism for distribution. CORBA specifications define a distribution architecture, including a standard protocol (IIOP) for request messaging. Consequently, the CORBA architecture, specifications, and protocols shall be used for distribution.

15.8.1 Bridge Locality

One of the goals of this specification is to allow any compliant interworking mechanism delivered on a COM client node to interoperate correctly with any CORBA-compliant components that use the same interface specifications. Compliant interworking solutions must appear, for all intents and purposes, to be CORBA object implementations and/or clients to other CORBA clients, objects, and services on an attached network.

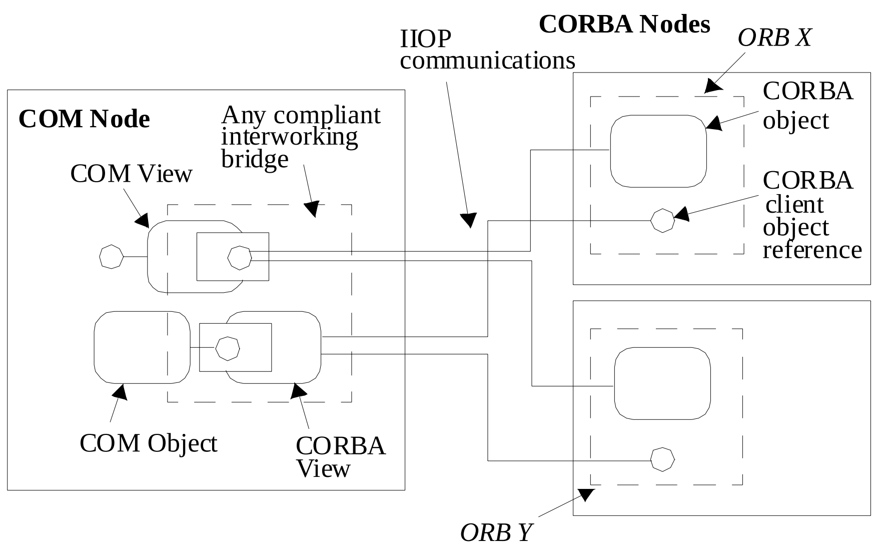

Figure 15-7 Bridge Locality

Table 15-7 on page 15-33 illustrates the required locality for interworking components. All of the transformations between CORBA interfaces and COM interfaces described in this specification will take place on the node executing the COM environment. Mapping agents (COM views, CORBA views, and bridging elements) will reside and execute on the COM client node. This requirement allows compliant interworking solutions to be localized to a COM client node, and to interoperate with any CORBA-compliant networking ORB that shares the same view of interfaces with the interworking solution.

15.8.2 Distribution Architecture

External communications between COM client machines, and between COM client machines and machines executing CORBA environments and services, will follow specifications contained in *CORBA*. Figure 15-7 illustrates the required distribution architecture. The following statements articulate the responsibilities of compliant solutions.

- All externalized CORBA object references will follow *CORBA* specifications for Interoperable Object References (IORs). Any IORs generated by components performing mapping functions must include a valid IIOP profile.
- The mechanisms for negotiating protocols and binding references to remote objects will follow the architectural model described in *CORBA*.
- A product component acting as a CORBA client may bind to an object by using any profile contained in the object’s IOR. The client must, however, be capable of binding with an IIOP profile.
- Any components that implement CORBA interfaces for remote use must support the IIOP.

15.9 Interworking Targets

This specification is targeted specifically at interworking between the following systems and versions:

- CORBA as described in *CORBA: Common Object Request Broker Architecture and Specification*.
- OLE as embodied in version 2.03 of the OLE run-time libraries.
- Microsoft Object Description Language (ODL) as supported by MKTYPELIB version 2.03.3023.
- Microsoft Interface Description Language (MIDL) as supported by the MIDL Compiler version 2.00.0102.

In determining which features of Automation to support, the expected usage model for Automation Views follows the Automation controller behavior established by Visual Basic 4.0.

15.10 Compliance to COM/CORBA Interworking

This section explains which software products are subject to compliance to the Interworking specification, and provides compliance points. For general information about compliance to CORBA specifications, refer to the Preface, Section 0.5, Definition of CORBA Compliance.

15.10.1 Products Subject to Compliance

COM/CORBA interworking covers a wide variety of software activities and a wide range of products. This specification is not intended to cover all possible products that facilitate or use COM and CORBA mechanisms together. This Interworking specification defines three distinct categories of software products, each of which are subject to a distinct form of compliance. The categories are:

- Interworking Solutions
- Mapping Solutions
- Mapped Components

Interworking Solutions

Products that facilitate the development of software that will bidirectionally transform COM and/or Automation invocations into isomorphic CORBA invocations (and vice versa) in a generic way are Interworking Solutions. An example of this kind of software would be a language processor that parses OMG IDL specifications and automatically generates code for libraries that map the OMG IDL interfaces into OLE Automation interfaces and which also parses OLE Automation ODL and automatically generates code for libraries that map the OLE Automation interfaces into CORBA interfaces. Another example would be a generic bridging component that, based on run-time interface descriptions, interpretively maps both COM and CORBA invocations onto CORBA and COM objects (respectively).

A product of this type is a **compliant** Interworking Solution if the resulting mapped interfaces are transformed as described in this specification, and if the mapped interfaces support all of the features and interface components required by this specification.

A compliant Interworking Solution must designate whether it is a compliant COM/CORBA Interworking Solution and/or a compliant Automation/CORBA Interworking Solution.

Mapping Solutions

Products that facilitate the development of software that will unidirectionally transform COM and/or Automation invocations into isomorphic CORBA invocations (and vice versa) in a generic way are described as *Mapping Solutions*. An example of this kind of software would be a language processor that parses OMG IDL specifications and automatically generates code for libraries that map the OMG IDL interfaces into OLE Automation interfaces. Another example would be a generic bridging component that interpretively maps OLE Automation invocations onto CORBA objects based on run-time interface descriptions.

A product of this type will be considered a **compliant** Mapping Solution if the resulting mapped interfaces are transformed as described in this specification, and if the mapped interfaces support all of the features and interface components required in this specification.

A compliant Mapping Solution must designate whether it is a compliant COM to CORBA Mapping Solution, a compliant Automation to CORBA Mapping Solution, a compliant CORBA to COM Mapping Solution, and/or a compliant CORBA to Automation Mapping Solution.

Mapped Components

Applications, components or libraries that expose a specific, fixed set of interfaces mapped from CORBA to COM or Automation (and/or vice versa) are described as Mapped Components. An example of this kind of product would be a set of business objects defined and implemented in CORBA that also expose isomorphic OLE Automation interfaces.

This type of product will be considered a **compliant** Mapped Component if the interfaces it exposes are mapped as described in this specification, and if the mapped interfaces support all of the features and interface components required in this specification.

15.10.2 Compliance Points

The intent of this submission is to allow the construction of implementations that fit in the design space described in Section 15.2, “Interworking Object Model,” on page 15-3, and yet guarantee interface uniformity among implementations with similar or overlapping design centers. This goal is achieved by the following compliance statements:

- When a product offers the mapping of CORBA interfaces onto isomorphic COM and/or Automation interfaces, the mapping of COM and/or Automation interfaces onto isomorphic CORBA interfaces, or when a product offers the ability to automatically generate components that perform such mappings, then the product must use the interface mappings defined in this specification. Note that products may offer custom, nonisomorphic interfaces that delegate some or all of their behavior to CORBA, COM, or Automation objects. These interfaces are not in the scope of this specification, and are neither compliant nor noncompliant.
- Interworking solutions that expose COM Views of CORBA objects are required to expose the CORBA-specific COM interfaces ICORBAObject and IORBObject, defined in “ICORBAObject Interface” on page 15-27 and “IORBObject Interface” on page 15-28, respectively.
- Interworking solutions that expose Automation Views of CORBA objects are required to expose the CORBA-specific Automation Dual interfaces DICORBAObject and DIORBObject, defined in “ICORBAObject Interface” on page 15-27 and “IORBObject Interface” on page 15-28, respectively.
- OMG IDL interfaces exposed as COM or Automation Views are not required to provide type library and registration information in the COM client environment where the interface is to be used. If such information is provided; however, then it must be provided in the prescribed manner.
- Each COM and Automation View must map onto one and only one CORBA object reference, and must also expose the IForeignObject interface, described in “IForeignObject Interface” on page 15-26. This constraint guarantees the ability to obtain an unambiguous CORBA object reference from any COM or Automation View via the IForeignObject interface.
- If COM or Automation Views expose the IMonikerProvider interface, they shall do so as specified in “IMonikerProvider Interface and Moniker Use” on page 15-23.
- All COM interfaces specified in this submission have associated COM Interface IDs. Compliant interworking solutions must use the IIDs specified herein, to allow interoperability between interworking solutions.
- All compliant products that support distributed interworking must support the CORBA Internet Inter-ORB Protocol (IIOP), and use the interoperability architecture described in CORBA in the manner prescribed in “Distribution” on page 15-32. Interworking solutions are free to use any additional proprietary or public protocols desired.
- Interworking solutions that expose COM Views of CORBA objects are required to provide the ICORBAFactory object as defined in “ICORBAFactory Interface” on page 15-24.

- Interworking solutions that expose Automation Views of CORBA objects are required to provide the DICORBAFactory object as defined in “ICORBAFactory Interface” on page 15-24.
- Interworking solutions that expose CORBA Views of COM or Automation objects are required to derive the CORBA View interfaces from **`CosLifeCycle::LifeCycleObject`** as described in CORBA View of COM/Automation Life Cycle, as described under “Binding and Life Cycle” on page 15-20.