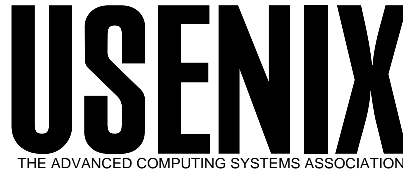

The following paper was originally published in the

5th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '99)

San Diego, California, USA, May 3–7, 1999

Intercepting and Instrumenting COM Applications

Galen C. Hunt
Microsoft Research

Michael L. Scott
University of Rochester

© 1999 by The USENIX Association
All Rights Reserved

Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes. This copyright notice must be included in the reproduced paper. USENIX acknowledges all trademarks herein.

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

Intercepting and Instrumenting COM Applications

Galen C. Hunt
Microsoft Research
One Microsoft Way
Redmond, WA 98052
galenh@microsoft.com

Michael L. Scott
Department of Computer Science
University of Rochester
Rochester, NY 14627
scott@cs.rochester.edu

Abstract

Binary standard object models, such as Microsoft's Component Object Model (COM) enable the development of not just reusable components, but also an incredible variety of useful component services through run-time interception of binary standard interfaces. Interception of binary components can be used for conformance testing, debugging, profiling, transaction management, serialization and locking, cross-standard middleware interoperability, automatic distributed partitioning, security enforcement, clustering, just-in-time activation, and transparent component aggregation.

We describe the implementation of an interception and instrumentation system tested on over 300 COM binary components, 700 unique COM interfaces, 2 million lines of code, and on 3 major commercial-grade applications including Microsoft PhotoDraw 2000. The described system serves as the foundation for the Coign Automatic Distributed Partitioning System (ADPS), the first ADPS to automatically partition and distribute binary applications.

While the techniques described in this paper were developed specifically for COM, they have relevance to other object models with binary standards, such as individual CORBA implementations.

1. Introduction

Widespread adoption of Microsoft's Component Object Model (COM) [16, 25] standard has produced an explosion in the availability of binary components, reusable pieces of software in binary form. It can be argued that this popularity is driven largely by COM's binary standard for component interoperability.

While binary compatibility is a great boon to the market for commercial components, it also enables a wide range of unique component services through interception. Because the interfaces between COM components are well defined by the binary standard, a component service can exploit the binary standard to intercept inter-component communication and interpose itself between components.

Interception of binary components can be used for conformance testing, debugging, distributed communication, profiling, transaction management, serialization and locking, cross-standard middleware interoperability, automatic distributed partitioning, security enforcement, clustering and replication, just-in-time activation, and transparent component aggregation.

In this paper, we describe an interception system proven on over 300 COM binary components, 700 unique COM interfaces, and 2 million lines of code [5]. We have extensively tested our COM interception system on three major commercial-grade applications: the MSDN Corporate Benefits Sample [12], Microsoft PhotoDraw 2000 [15], and the Octarine word-processor from the Microsoft Research COM Applications Group. The interception system serves as the foundation for the Coign Automatic Distributed Partitioning System (ADPS) [7] [8], the first ADPS to automatically partition and distribute binary applications.

In the next section, we describe the fundamental features of COM as they relate to the interception and instrumentation of COM applications. Sections 3 and 4 explain and evaluate our mechanisms for intercepting object instantiation requests and inter-object communication respectively. We describe related work in Section 5. In Section 6, we present our conclusions and propose future work.

2. COM Fundamentals

COM is a standard for creating and connecting components. A COM component is the binary template from which a COM object is instantiated. Due to COM's binary standard, programmers can easily build applications from components, even components for which they have no source code. COM's major features include multiple interfaces per object, mappings for common programming languages, standard-mandated binary compatibility, and location-transparent invocation.

2.1. Polymorphic Interfaces

All first-class communication in COM takes place through interfaces. An interface is a strongly typed reference to a collection of semantically related functions. An interface is identified by a 128-bit globally unique identifier (GUID). An explicit agreement between two components to communicate through a

named interface contains an implicit contract of the binary representation of the interface.

Microsoft Interface Definition Language (MIDL)

Figure 1 contains the definitions of two interfaces: `IUnknown` and `Istream` in the *Microsoft Interface Definition Language* (MIDL). Syntactically, MIDL is very similar to C++. To clarify the semantic features of interfaces, MIDL attributes (enclosed in square brackets []) can be attached to any interface, member function, or parameter. Attributes specify features such as the data-flow direction of function arguments, the size of dynamic arrays, and the scope of pointers. For example, the `[in, size_is(cb)]` attribute on the `pb` argument of the `Write` function in Figure 1 declares that `pb` is an input array with `cb` elements.

```
[uuid(00000000-0000-0000-C000-000000000046)]
interface IUnknown
{
    HRESULT QueryInterface(
       [in] REFIID riid,
       [out,iid_is(riid)] void **ppObj);
    ULONG AddRef();
    ULONG Release();
};

[uuid(b3c11b80-9e7e-11d1-b6a5-006097b010e3)]
interface IStream : IUnknown
{
    HRESULT Seek(
       [in] LONG nPos);
    HRESULT Read(
       [out,size_is(cb)] BYTE *pb,
       [in] LONG cb);
    HRESULT Write(
       [in,size_is(cb)] BYTE *pb,
       [in] LONG cb);
};
```

Figure 1. MIDL for Two Interfaces.

The MIDL definition of an interface describes its member functions and their parameters in sufficient detail to support location-transparent invocation.

`IUnknown`

The `IUnknown` interface, listed in Figure 1, is special. All COM objects must support `IUnknown`. Each COM interface must include the three member functions from `IUnknown`, namely: `QueryInterface`, `AddRef`, and `Release`. `AddRef` and `Release` are reference-counting functions for lifetime management. When an object's reference count goes to zero, the object is responsible for freeing itself from memory.

COM objects can support multiple interfaces. Clients dynamically bind to a new interface by calling `QueryInterface`. `QueryInterface` takes as input the GUID of the interface to which the client would like to bind and returns a pointer to the new interface. Through run-time invocation of `QueryInterface`, clients can determine the exact functionality supported by any object.

2.2. Common Language Mappings

The MIDL compiler maps interface definitions into formats usable by common programming languages. Figure 2 contains the C++ abstract classes generated by the MIDL compiler, for the interfaces in Figure 1. MIDL has straightforward mappings into other compiled languages such as C and Java. In addition, the MIDL compiler can store metadata in binary files called *type libraries*. Many development tools can import type libraries. Type libraries are well suited for scripting languages such as the Visual Basic Scripting Edition in Internet Explorer [11].

```
class IUnknown
{
  public:
    virtual HRESULT QueryInterface(
                     REFIID riid,
                     void **ppObj) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
};

class IStream : IUnknown
{
  public:
    virtual HRESULT Seek(
                     LONG nPos) = 0;
    virtual HRESULT Read(
                     BYTE *pb,
                     LONG cb) = 0;
    virtual HRESULT Write(
                     BYTE *pb,
                     LONG cb) = 0;
};
```

Figure 2. C++ Language Mapping.

The MIDL compiler maps a COM interface into an abstract C++ class.

2.3. Binary Compatibility

In addition to language mappings, COM specifies a platform-standard binary mapping for interfaces. The binary format for a COM interface is similar to the common format of a C++ *virtual function table* (VTBL, pronounced "V-Table"). All references to interfaces are stored as interface pointers (an indirect pointer to a virtual function table). Figure 3 shows the binary mapping of the `IStream` interface.

Each object is responsible for allocating and releasing the memory occupied by its interfaces. Quite often, objects place per-instance interface data immediately

following the interface virtual-function-table pointer. With the exception of the virtual function table and the pointer to the virtual function table, the object memory area is opaque to the client.

The standardized binary mapping enforces COM's language neutrality. Any language that can call a function through a pointer can use COM objects. Any language that can export a function pointer can create COM objects.

COM components are distributed either in application executables (.EXE files) or in *dynamic link libraries* (DLLs).

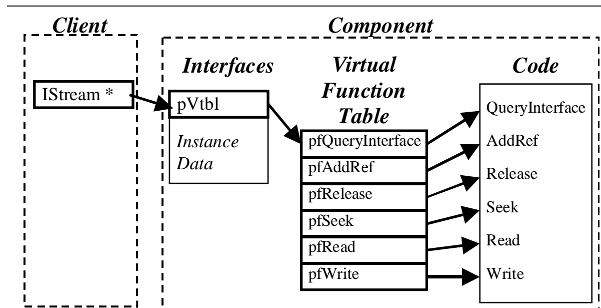

Figure 3. Binary Interface Mapping.

COM defines a standard binary mapping for interfaces. The format is similar to the common representation of a C++ pure abstract virtual function table.

2.4. Location Transparency

Binary compatibility is important because it facilitates true location transparency. A client can communicate with a COM object in the same process (*in-process)*, in a different process (*cross-process)*, or on an entirely different machine (*cross-machine)*. The location of the COM object is completely transparent to both client and component because in each case invocation takes place through an interface's virtual function table.

Interface Proxies and Stubs

Location transparency is achieved through proxies and stubs generated by the MIDL compiler. *Proxies* marshal function arguments into a single message that can be transported between address spaces or between machines. *Stubs* unmarshal messages into function calls. Interface proxies and stubs copy data structures with deep-copy semantics. In theory, proxies and stubs come in pairs—the first for marshaling and the second for unmarshaling. In practice, COM generally combines code for the proxy and stub for a specific interface into a single reusable binary. COM proxies and stubs are similar in purpose to CORBA [19, 23] stubs and skeletons. However, their implementations vary because COM proxies and stubs are only used when inter-object communication crosses process boundaries.

In-Process Communication

For best performance, components reside in the client's address space. An application invokes an in-process object directly through the interface virtual function table. In-process communication has the same cost as a C++ virtual function call because it uses neither interface proxies nor stubs. The primary drawback of in-process objects is that they share the same protection domain as the application. The application cannot protect itself from erroneous or malicious resource access by the object.

Cross-Process Communication

To provide the application with security, objects can be located in another operating-system process. The application communicates with cross-process objects through interface proxies and stubs. The application invokes the object through an indirect call on an interface virtual function table. In this case, however, the virtual function table belongs to the interface proxy. The proxy marshals function arguments into a buffer and transfers execution to the object's address space where the interface stub unmarshals the arguments and calls the object through the interface virtual function table in the target address space. Marshaling and unmarshaling are completely transparent to both application and component.

Cross-Machine Communication

Invocation of distributed objects is very similar to invocation of cross-process objects. Cross-machine communication uses the same interface proxies and stubs as cross-process communication. The primary difference is that once the function arguments have been marshaled, COM sends the serialized message to the destination machine using the DCOM protocol [3], a superset of the Open Group's Distributed Computing Environment Remote Procedure Call (DCE RPC) protocol [4].

3. Interception of Object Instantiations

COM objects are dynamic objects. Instantiated during an application's execution, objects communicate with the application and each other through dynamically bound interfaces. An object frees itself from memory after all references to it have been released by the application and other objects.

Applications instantiate COM objects by calling API functions exported from a user-mode COM DLL. Applications bind to the COM DLL either statically or dynamically.

Static binding to a DLL is very similar to the use of shared libraries in most UNIX systems. Static binding

is performed in two stages. At link time, the linker embeds in the application binary the name of the DLL, a list of all imported functions, and an indirect jump table with one entry per imported function. At load time, the loader maps all imported DLLs into the application's address space and patches the indirect jump table entries to point to the correct entry points in the DLL image.

Dynamic binding occurs entirely at run time. A DLL is loaded into the application's address space by calling the `LoadLibrary` Win32 function. After loading, the application looks for procedures within the DLL using the `GetProcAddress` function. In contrast to static binding, in which all calls use an indirect jump table, `GetProcAddress` returns a direct pointer to the entry point of the named function.

BindMoniker	OleCreateDefaultHandler
CoCreateInstance	OleCreateEx
CoCreateInstanceEx	OleCreateFontIndirect
CoGetClassObject	OleCreateFromData*
CoGetInstanceFromFile	OleCreateFromFile*
CoRegisterClassObject	OleCreateLink*
CreateAntiMoniker	OleCreateStaticFromData
CreateBindCtx	OleGetClipboard
CreateClassMoniker	OleLoad
CreateDataAdviseHolder	OleLoadFromStream
CreateFileMoniker	OleLoadPicture
CreateGenericComposite	OleLoadPictureFile
CreateItemMoniker	OleRegEnumFormatEtc
CreateOleAdviseHolder	OleRegEnumVerbs
CreatePointerMoniker	StgCreateDocfile
GetRunningObjectTable	StgCreateDocfileOn*
MkParseDisplayName	StgGetIFillLockBytesOn*
MonikerCommonPrefixWith	StgOpenAsyncDocfileOn*
MonikerRelativePathTo	StgOpenStorage
OleCreate	StgOpenStorageOn*

Figure 4. Object Instantiation Functions.

COM supports approximately 50 functions capable of creating instantiation a new object. However, most instantiations request use either `CoCreateInstance` or `CoCreateInstanceEx`.

The COM DLL exports approximately 50 functions capable of instantiating new objects; these are listed in Figure 4. With few exceptions, applications instantiate objects exclusively through the `CoCreateInstance` function or its successor, `CoCreateInstanceEx`. From the instrumentation perspective there is little difference among the COM API functions. For brevity, we use `CoCreate` as a placeholder for any function that instantiates new COM objects.

3.1. Alternatives for Instantiation Interception

To intercept all object instantiations, instrumentation should be called at the entry and exit of each object instantiation function.

Figure 5 enumerates the techniques available for intercepting functions; namely: source-code call replacement, binary call replacement, DLL redirection, DLL replacement, breakpoint trapping, and inline redirection.

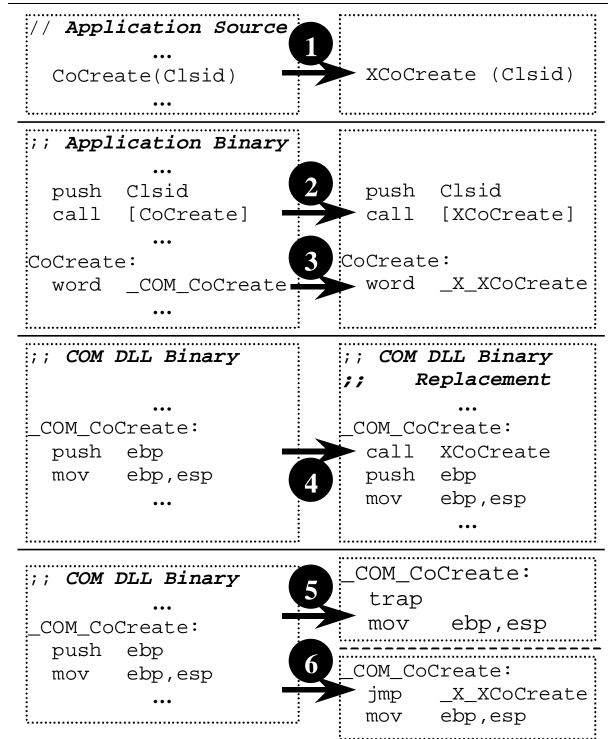

Figure 5. Intercepting Instantiation Calls.

Object instantiation calls can be intercepted by 1) call replacement in the application source code; 2) call replacement in the application binary; 3) DLL redirection; 4) DLL replacement; 5) trapping in the COM DLL; and 6) inline redirection in the COM DLL.

Call replacement in application source code.

Calls to the COM instantiation functions can be replaced with calls to the instrumentation by modifying application source code. The major drawback of this technique is that it requires access to application source code.

Call replacement in application binary code.

Calls to the COM instantiation functions can be replaced with calls to the instrumentation by modifying application binaries. While this technique does not require source code, replacement in the application binary does require the ability to identify all applicable call sites. To facilitate identification of all call sites, the application must be linked with substantial symbolic information.

DLL redirection.

The import entries for COM APIs in the application can be modified to point to another library. Redirection to another DLL can be achieved either by replacing the name of the COM DLL in the import table before load time or by replacing the function addresses in the indirect jump table after load. Unfortunately, redirecting to another DLL through either of the import tables fails to intercept dynamic calls using `LoadLibrary` and `GetProcAddress`.

DLL replacement.

The only way to guarantee interception of a specific DLL function is to insert the interception mechanism into the function code. The most obvious method is to replace the COM DLL with a new version containing instrumentation. DLL replacement requires source access to the COM DLL library. It also unnecessarily penalizes all applications using the COM DLL, whether they use the additional functionality or not.

Breakpoint trapping of the COM DLL.

Rather than replace the DLL, the interception mechanism can be inserted into the image of the COM DLL after it has been loaded into the application address space. At run time, the instrumentation system can insert a breakpoint trap at the start of each target instantiation function. When execution reaches the function entry point, a debugging exception is thrown by the trap and caught by the instrumentation system. The major drawback to breakpoint trapping is that debugging exceptions suspend all application threads. In addition, the debug exception must be caught in a second operating-system process. Interception via breakpoint trapping has a high performance penalty.

Inline redirection of the COM DLL.

The most favorable method for intercepting DLL functions is to inline the redirection call. At load time, the first few instructions of the target instantiation function are replaced with a jump instruction to a detour function in the instrumentation. Replacing the first few instructions is usually a trivial operation as these instructions are normally part of the function prolog generated by a compiler and not the targets of any branches. The replaced instructions are used to create a trampoline. When the modified target function is invoked, the jump instruction transfers execution to the detour function in the instrumentation. The detour function passes control to the remainder of the target function by invoking the trampoline.

3.2. Evaluation of Instantiation Interception

Our instrumentation system uses inline indirection to intercept object instantiation calls. At load time, our instrumentation replaces the first few instructions of the target function with a jump to the instrumentation detour function. Pages for code sections are mapped into a processes' address space using copy-on-write semantics. Calls to `VirtualProtect` and `FlushInstructionCache` enable modification of code pages at run time. Instructions removed from the target function are placed in a statically allocated trampoline routine. As shown in Figure 6, the trampoline allows the detour function to invoke the target function without interception.

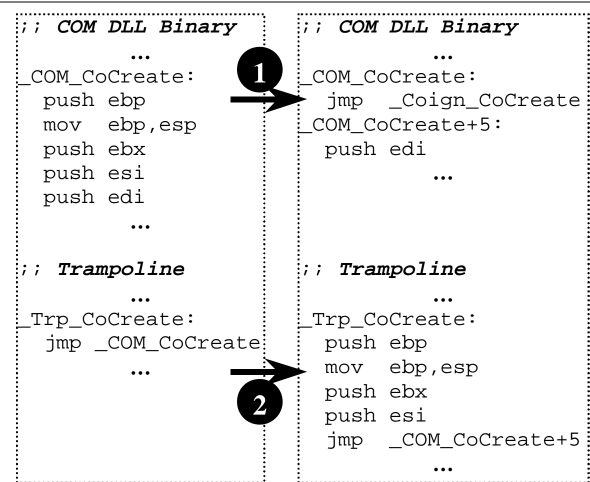

Figure 6. Inline Redirection.

The first few instructions of the target API function are moved to the trampoline and replaced with a jump to the interception system. The trampoline effectively invokes the API function without interception. On the Intel x86 architecture, a jump instruction occupies five bytes.

Function vs. Interception Technique	Empty Function	CoCreateInstance
Direct Call	0.11us	14.84us
DLL Redirection	0.14us	15.19us
DLL Replacement	0.14us	15.19us
Breakpoint Trap	229.56us	265.85us
Inline Redirection	0.15us	15.19us

Table 1. Interception Times.

Listed are the times for intercepting either an empty function or `CoCreateInstance` on a 200MHz Pentium PC.

Although inline indirection is complicated by the variable-length instruction set of the Intel x86 architecture, its low run-time cost and versatility more than offset the development penalty. Inline redirection of the `CoCreateInstance` function has less than a 3% overhead, which is more than an order of magnitude smaller than the penalty for breakpoint trapping. Table 1 lists the average invocation time of the target function within a loop consisting of 10,000 iterations. The invocation times include the cost of redirection, but not any additional instrumentation. Unlike DLL redirection, inline redirection correctly intercepts both statically and dynamically bound invocations. Finally, inline redirection is much more flexible than DLL redirection or application code modification. Inline redirection of any API function can be selectively enabled for each process individually at load time based on the needs of the instrumentation.

To apply inline redirection, our instrumentation system must be loaded into the application's address space before the application executes. The current system is packaged as a DLL and post-linked to the application binary with a binary rewriter. Once loaded into the application address space, instrumentation is inlined into system DLL images. Mechanisms for inserting the interception system into an application's address space are described fully in a paper on our Detours package [6].

4. Intercepting Inter-Object Calls

The bulk the interception system's functionality is devoted to identifying interfaces, understanding their relationships to each other, and quantifying the communication through them. This section describes how our system intercepts interface calls.

Invoking an interface member function is similar to invoking a C++ member function. The first argument to any interface member function is the "`this`" pointer, the pointer to the interface. Figure 7 lists the C++ and C syntax to invoke an interface member function.

4.1. Alternatives for Invocation Interception

There are four techniques, described below, available to intercept member function invocations:

Replace the interface pointer.

Rather than return the object's interface pointer, the interception system can return a pointer to an interface of its own making. When the client attempts to invoke an interface member function, it will invoke the instrumentation, not the object. After taking appropriate steps, the instrumentation "forwards" the request to the object by directly invoking the object interface. In one sense, replacing the interface pointer is functionally similar to using remote interface proxies and stubs. For remote marshaling, COM replaces a remote interface pointer with a local interface pointer to an interface proxy.

Replace the interface virtual function table pointer.

The runtime can replace the virtual function table pointer in the interface with a pointer to an instrumentation-supplied virtual function table. The instrumentation can forward the invocation to the object by keeping a private copy of the original virtual function table pointer.

Replace function pointers in the interface virtual function table.

Rather than intercept the entire interface as a whole, the interception system can replace each function pointer in the virtual function table individually.

Intercept object code.

Finally, the instrumentation system can intercept member-function calls at the actual entry point of the function using inline redirection.

```
IStream *pIStream;

// C++ Syntax
pIStream->Seek(nPos);

// C Syntax
pIStream->pVtbl->pfSeek(pIStream, nPos);
```

Figure 7. Invoking an Interface Function.

Clients invoke interface member functions through the interface pointer. The first parameter to the function (hidden in C++) is the "`this`" pointer to the interface.

4.2. COM Programming Idioms

The choice of an appropriate technique for intercepting member functions is constrained by COM's binary standard for object interoperability and common COM programming idioms. Our interception system attempts to deduce the identity of the each called object, the static type of the called interface, the identity of the called member function, and the static types of all function parameters. In addition, our interception degrades gracefully. Even if not all of the needed information can be determined, the interception system continues to function correctly.

By design, the COM binary standard restricts the implementation of interfaces and objects to the degree necessary to insure interoperability. COM places four specific restrictions on interface design to insure object interoperability. First, a client accesses an object through its interface pointers. Second, the first item pointed to by an interface pointer must be a pointer to a

virtual function table. Third, the first three entries of the virtual function table must point to the `QueryInterface`, `AddRef` and `Release` functions for the interface. Finally, if a client intends to use an interface, it must insure that the interface's reference count has been incremented.

As long as an object programmer obeys the four rules of the COM binary standard, he or she is completely free to make any other implementation choices. For example, the component programmer is free to choose any appropriate memory layout for object and per-instance interface data. This lack of implementation constraint is not an accident. The original designers of COM were convinced that no one implementation (even of something as universal as the `QueryInterface` function) would be suitable for all users. Instead, they attempted to create a specification that enabled binary interoperability while preserving all other degrees of freedom.

Specification freedom breeds implementation diversity. This diversity is manifest in the number of common programming idioms employed by COM component developers. These idioms are described here in sufficient detail to highlight the constraints they place on the implementation of a COM interception and instrumentation system. Each of these idioms has broken at least one other COM interception system or preliminary versions of our interception system.

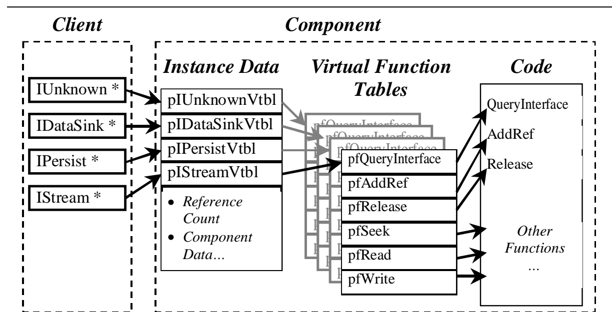

Figure 8. Simple Object Layout.

The object instance is allocated as a single memory block. The block contains one VTBL pointer for each supported interface, an instance reference count, and other object-specific data. All interfaces share common implementations of `QueryInterface`, `AddRef`, and `Release`.

Simple Multiple-Interface Objects

Most objects support at most roughly a dozen interfaces with no duplicates. It is common practice to lay out these simple objects in a memory block containing one VTBL pointer per interface, a reference count, and internal object variables; see Figure 8. Within the object's member functions, a constant value is added to the "`this`" pointer to find the start of the memory block and to access object variables. All of the object interfaces use a common pair of `AddRef` and `Release` functions to maintain the object reference count.

Multiple-Instance and Tear-off Interfaces

Sometimes, an object must support multiple copies of a single interface. *Multiple-instance* interfaces are often used for iteration. A new instance of the interface is allocated for each client. Multiple-instance interfaces are typically implemented using a *tear-off* interface. A tear-off interface is allocated as a separate memory block. The tear-off interface contains the interface's VTBL pointer, an interface-specific reference count, a pointer to the object's primary memory block, and any instance-specific data. In addition to multiple-instance interfaces, tear-off interfaces are often used to implement rarely accessed interfaces when object memory size must be minimized, (i.e. when the cost of the extra four bytes for a VTBL pointer per object instance is too expensive).

Universal Delegators

Objects commonly use a technique called *delegation* to export interfaces from another object to a client. Delegation is often used when one object aggregates services from several other objects into a single entity. The aggregating object exports its own interfaces, which delegate their implementation to the aggregated objects. The delegating interface calls the aggregated interface. This implementation is interface specific, code intensive, and requires an extra procedure call during invocation. The implementation is code intensive because delegating code must be written for each interface type. The extra procedure call becomes particularly important if the member function has a large number of arguments or multiple delegators are nested through layers of aggregation.

An obvious optimization and generalization of delegation is the *universal delegator*. A universal delegator is a type-independent, re-usable delegator. The data structure for a universal delegator consists of a VTBL pointer, a reference count, a pointer to the aggregated interface, and a pointer to the aggregating object. Upon invocation, a member function in the universal delegator replaces the "`this`" pointer on the argument stack with the pointer to the delegated interface and jumps directly to the entry point of the appropriate member function in the aggregated interface. The universal delegator is "universal" because its member functions need know nothing about the type of interface to which they are delegating; they reuse the invoking call frame. Implemented in a manner similar to tear-off interfaces, universal delegators are instantiated on demand, one per delegated interface with a common VTBL shared among all instances.

Explicit VTBL Pointer Comparison.

Rather than using explicit constant offsets, some COM components implemented in C locate the start of an object's main memory block by comparing VTBL interface pointers. For example, the `IStream::Seek` member function of the object in Figure 8 starts with its "this" pointer pointing to `pIStreamVtbl`. The object locates the start of its memory structure by decrementing the "`this`" pointer until it points to a VTBL pointer equal to the known location of the VTBL for `IUnknown`. This calculation will produce erroneous results if an interception system has replaced the VTBL pointer.

Explicit Function Pointer Comparison.

In a manner similar to VTBL pointer comparison, some components perform calculations assuming that function pointers in the VTBL will have known values. These calculations break if the interception system has replaced a VTBL function pointer.

4.3. Interface Wrapping

Our instrumentation system intercepts invocation of interface member functions by replacing the interface pointer given to the object's client with an interface pointer to a specialized universal delegator, the *interface wrapper*. The implementation of interface wrappers was chosen after evaluating the functionality of possible alternatives and testing their performance against a suite of object-based applications.

For brevity, we often refer to the process of creating an individual interface wrapper and replacing the interface pointer with a pointer to an interface wrapper as *wrapping* the interface. We also refer to interfaces as being *wrapped* or *unwrapped*. A wrapped interface is one to which clients receive a pointer to the interface wrapper. An unwrapped interface is one either without a wrapper or with the interface wrapper removed to yield the original object interface.

Interface wrapping provides an easy way to identify an interface and a ready location to store information about the interface: in the per-instance interface wrapper. Unlike interface wrapping, inline redirection must store per-instance data in an external dictionary. Access to the instance-data dictionary is made difficult because member functions are often re-used by multiple interfaces of dissimilar type. This is definitely the case for universal delegation, but common even for less exotic coding techniques. As a rule, almost all objects reuse the same implementation of `QueryInterface`, `AddRef`, and `Release` for multiple interfaces.

Interface wrapping is robust, does not break application code, and is extremely efficient. Finally, as we shall see in the next section, interface wrapping is central to correctly identifying the object that owns an interface.

4.4. The Interface Ownership Problem

In addition to intercepting interface calls, the interception system attempts to identify which object owns an interface. A major breakthrough in the development of our interception system was the discovery of heuristics to find an interface's owning object.

The interface ownership problem is complicated because to COM, to the application, and to other objects, an object is visible only as a loosely coupled set of interfaces. The object can be identified only through one of its interfaces; it has no explicit object identity.

COM supports the concept of an object identity through the `IUnknown` interface. As mentioned in Chapter 2, every interface must inherent from and implement the three member functions of `IUnknown`, namely: `QueryInterface`, `AddRef`, and `Release`. Through the `QueryInterface` function, a client can query for any interface supported by the object. Every object must support the `IUnknown` interface. An object's `IUnknown` interface pointer *is* the object's *COM identity*. The COM specification states that a client calling `QueryInterface(IID_IUnknown)` on any interface must always receive back the same `IUnknown` interface pointer (the same COM identity).

Unfortunately, an object need not provide the same COM identity (the same `IUnknown` interface pointer) to different clients. An object that exports one COM identity to one client and another COM identity to a second client is said to have a *split identity*. Split identities are especially common in applications in which objects are composed together through a technique known as aggregation. In aggregation, multiple objects operate as a single unit by exporting a common `QueryInterface` function to all clients. Due to split identities, COM objects have no system-wide, unique identifier.

The Obvious Solution

A client can query an interface for its owning `IUnknown` interface (its COM identity). In the most obvious implementation, the interception system could maintain a list of known COM identities for each object. The runtime could identify the owning object by querying an interface for its COM identity and comparing it to a dictionary of known identities.

In practice, calling `QueryInterface` to identify the owning object fails because `QueryInterface` is not free of side effects. `QueryInterface` increments the reference count of the returned interface. Calling `Release` on the returned interface would decrement its reference count. However, the `Release` function also has side effects. `Release` instructs the

object to check if its reference count has gone to zero and to free itself from memory in the affirmative. There are a few identification scenarios under which the object's reference count does in fact go to zero. In the worse case scenario, attempting to identify an interface's owner would produce the unwanted side effect of instructing the object to remove itself from memory!

Sources of Interface Pointers

To find a correct solution to the interface ownership problem, one must understand how a client receives an interface pointer. It is also important to understand what information is available about the interface. A client can receive an object interface pointer in one of four ways: from one of the COM API object instantiation functions; by calling `QueryInterface` on an interface to which it already holds a pointer; as an output parameter from one of the member functions of an interface to which it already holds a pointer; or as an input parameter on one of its own member functions. Recall that our system intercepts all COM API functions for object instantiation. At the time of instantiation, the interception system wraps the interface and returns to the caller a pointer to the interface wrapper.

An Analogy for the Interface Ownership Problem

The following analogy is helpful for understanding the interface ownership problem. A person finds herself in a large multi-dimensional building. The building is divided into many rooms with doors leading from one room to another. The person is assigned the task of identifying all of the rooms in the building and determining which doors lead to which rooms. Unfortunately, all of the walls in the building are invisible. Additionally, from time to time new doors are added to the building and old doors are removed from the building.

Mapping the analogy to the interface ownership problem; the building is the application, the rooms are the objects, and the doors are the interfaces.

We describe the solution first in terms of the invisible room analogy, then as it applies to the interface ownership problem. In the analogy, the solution is to assign each room a different color and to paint the doors of that room as they are discovered. The person starts her search in one room. She assigns the room a color—say red. Feeling her way around the room, she paints one side of any door she can find without leaving the room. The door must belong to the room because she didn't pass through a door to get to it. After painting all of the doors, she passes through one of the doors into a new room. She assigns the new room a color—say blue. She repeats the door-painting algorithm for all doors in the blue room. She then passes through one of the doors and begins the process again. The person repeats the process, passing from one room to another.

If at some point the person finds that she has passed into a room where the door is already colored, then she knows the identity of the room (by the color on the door). She looks for any new doors in the room, paints them the appropriate color, and finally leaves through one of the doors to continue her search.

The Solution to the Interface Ownership Problem

From the analogy, the solution to the interface ownership problem is quite simple. Each object is assigned a unique identifier. Each thread holds in a temporary variable the identity of the object in which it is currently executing. Any newly found interfaces are instrumented with an interface wrapper. The current object identity is recorded in the interface wrapper as the owning object. Finding the doors in a room is analogous to examining interface pointers passed as parameters to member functions. When execution exits an object, any unwrapped interface pointers passed as parameters are wrapped and given the identity of their originating object. By induction, if an interface pointer is not already wrapped, then it must belong to the current object.

The most important invariant for solving the interface ownership problem is that at any time the interception system must know exactly which object is executing. Stored in a thread-local variable, the current object identifier is updated as execution crosses through interface wrappers. The new object identifier is pushed onto a local stack on entry to an interface. On exit from an interface wrapper (after executing the object's code), the object identifier is popped from the top of the stack. At any time, the interception system can examine the top values of the identifier stack to determine the identity of the current object and any calling objects.

There is one minor caveat in implementing the solution to the interface ownership problem. While clients should only have access to interfaces through interface wrappers, an object should never see an interface wrapper instead of one of its own interfaces because the object uses its interfaces to access instance-specific data. An object could receive an interface wrapper to one of its own interfaces if a client passes an interface pointer back to the owning object as an input parameter on another call. The solution is simply to unwrap an interface pointer whenever the pointer is passed as a parameter to its owning object.

4.5. Acquiring Static Interface Metadata

Interface wrapping requires static metadata about interfaces. The interface wrapper must be able to identify all interface pointers passed as parameters to an interface member function. There are a number of sources for acquiring static interface metadata. Possible sources include the MIDL description of an interface, COM type libraries, and interface proxies and stubs.

Acquiring static interface metadata from the MIDL description of an interface requires static analysis tools to parse and extract the appropriate metadata from the MIDL source code. In essence, it needs the MIDL compiler. Ideally, interface static metadata should be available to the interface wrapping code in a compact binary form.

Another alternative is to acquire static interface metadata from the COM type libraries. COM type libraries allow access to COM objects from interpreters for scripting languages, such as JavaScript [18] or Visual Basic [13]. While compact and readily accessible, type libraries describe only a subset of possible COM interfaces. Interfaces described in type libraries cannot have multiple output parameters. In addition, the metadata in type libraries does not contain sufficient information to determine the size of all possible dynamic array parameters.

Static interface metadata is also contained in the interface proxies and stubs. MIDL-generated proxies and stubs contain marshaling metadata encoded in strings of marshaling operators (called MOP strings). Static interface metadata can be acquired easily by interpreting the MOP strings. Unfortunately, the MOP strings are not publicly documented. Through an extensive process of trial and error involving more than 600 interfaces, at the University of Rochester, we were able to determine the meanings of all MOP codes emitted by the MIDL compiler.

Our interception system contains a MOP interpreter and a MOP precompiler. A heavyweight, more accurate interception subsystem uses our homegrown MOP interpreter. A lightweight interception subsystem uses the MOP precompiler to simplify the MOP strings (removing full marshaling information) before application execution.

The MOP precompiler uses dead-code elimination and constant folding to produce an optimized metadata representation. The simplified metadata describes all interface pointers passed as interface parameters, but does not contain information to calculate parameter sizes or fully walk pointer-rich arguments. Processed by a secondary interpreter, the simplified metadata allows the lightweight runtime to wrap interfaces in a fraction of the time required with full MOP strings.

While other COM instrumentation systems do use the MOP strings to acquire static interface metadata, ours is the first system to exploit a precompiler to optimize parameter access

The interception system acquires MOP strings directly from interface proxies and stubs. However, in some cases, components are distributed with MIDL source code, but without interface proxies and stubs. In those cases, the programmer can easily create interface proxies and stubs from the IDL sources with the MIDL compiler. OLE ships with about 250 interfaces without MOP strings. We were able to create interface proxies and stubs with the appropriate MOP string in under one hour using MIDL files from the OLE distribution.

4.6. Coping With Undocumented Interfaces

A final difficulty in interface wrapping is coping with *undocumented* interfaces, those interfaces without static metadata. While all documented COM interfaces should have static metadata, we have found cases where components from the same vendor will use an undocumented interface to communicate with each other.

When a function call on a documented interface is intercepted, the interface wrapper processes the incoming function parameters, creates a new stack frame, and calls the object interface. Upon return from the object's interface, the interface wrapper processes the outgoing function parameters and returns execution to the client. Information about the number of parameters passed to the member function is used to create the new stack frame for calling the object interface. For documented interfaces, the size of the new stack frame can easily be determined from the marshaling byte codes.

When intercepting an undocumented interface, the interface wrapper has no static information describing the size of stack frame used to call the member function. The interface wrapper cannot create a stack frame to call the object. It must reuse the existing stack frame. In addition, the interface wrapper must intercept execution return from the object in order to preserve the interface wrapping invariants used to identify objects and to determine interface ownership.

For function calls on undocumented interfaces, the interface wrapper replaces the return address in the stack frame with the address of a trampoline function. The original return address and a copy of the stack pointer are stored in thread-local temporary variables. The interface wrapper transfers execution to the object directly using a jump rather than a call instruction.

When the object finishes execution, it issues a return instruction. Rather than return control to the caller—as would have happened if the interface wrapper had not replaced the return address in the stack frame—execution passes directly to the trampoline. As a fortuitous benefit of COM's callee-popped calling convention, the trampoline can calculate the function's stack frame size by comparing the current stack pointer with the copy stored before invoking the object code. The trampoline saves the frame size for future calls, and then returns control to the client directly through a jump instruction to the temporarily stored return address.

The return trampoline is used only for the first invocation of a specific member function. Subsequent calls to the same interface member function are forwarded directly through the interface wrapper.

By using the return trampoline, the interception system continues to function correctly even when confronted with undocumented interfaces. To our knowledge, our is the only COM instrumentation system to tolerate undocumented interfaces.

4.7. Evaluation of Interface Wrapping

Detailed in Table 2, wrapping the interface by replacing the interface pointer adds a 36% overhead to trivial function like `IUnknown::AddRef` and just a 3% overhead to a function like `IStream::Read`. Processing the function arguments with interpreted MOP strings adds on average about 20% additional execution overhead while processing with precompiled MOP strings adds under 3% additional overhead. Replacing the interface pointer is preferred over the alternative interception mechanisms because it does not break under common COM programming idioms.

Function vs. Interception Technique	IUnknown::AddRef	IStream::Read
Direct Call	0.19us	15.73us
Replace Interface Pointer	0.26us	16.24us
Replace VTBL	0.26us	16.24us
Replace Function Pointer	0.26us	16.24us
Intercept Object Code	0.30us	16.29us

Table 2. Interface Interception Times.

Listed are the times for intercepting the `IUnknown::AddRef` and `IStream::Read` (with 256 bytes of payload data) on a 200MHz Pentium PC.

5. Related Work

Brown [1, 2] describes an interception system for COM using Universal Delegators (UDs). To use Brown's UD, the application programmer is entirely responsible for wrapping COM interfaces. The programmer must manually wrap each outgoing or incoming parameter with a special call to the UD code. While providing robust support for applications such as object aggregation, Brown's UD is not suitable for binary-only interception and instrumentation.

HookOle [10] is a general interception system for instrumenting COM applications. Like our system, HookOle extracts interface metadata from MIDL MOP strings. However, rather than replacing interface pointers, HookOLE replaces function pointers (in the VTBL) and assumes that the same function will not be used to implement multiple, dissimilarly typed interfaces. HookOLE breaks whenever an object uses universal delegation. HookOle provides no support for undocumented interfaces. The ITest Spy Utility [14] uses HookOle to provide a test harness for OLE DB components.

Microsoft Transaction Server (MTS) [21] intercepts inter-component communication to enforce transaction boundaries and semantics. MTS wraps COM interfaces in a manner similar to our interception system. However, MTS supports only a subset of possible COM interfaces and does not provide support for undocumented interfaces.

COM+ [9] provides a generalized mechanism called, *interceptors*, for intercepting communication between COM+ objects. A significant redesign of COM, COM+ has complete control over the memory layout of all objects. This control significantly reduces the complexity of interception, but only works for newly designed COM+ components.

COMERA [24] is an extensible remoting architecture for distributed COM communication. COMERA relies on existing DCOM [3] proxies and stubs to intercept cross-process communication. Neither COMERA nor DCOM support in-process interception.

Eternal [17] intercepts CORBA IIOP-related messages via the Unix `/proc` mechanism. Intercepted messages are broadcast to objects replicated for fault tolerance. The `/proc` mechanism is limited to cross-process communication and extremely expensive (requiring at least two crossings of process boundaries).

Finally, a number of CORBA [23] vendors support interception and filtering mechanisms. In general, instrumenting COM applications is more difficult than equivalent CORBA applications. COM standardizes interface format, but not object format. Each ORB specifies parts of the CORBA object format related to interception. So for example, the interface ownership problem has no equivalent in CORBA, but the problem of instrumenting binary CORBA application independent of ORB vendor remains unsolved.

6. Conclusions and Future Work

We have described a general-purpose interception system for instrumenting COM components and applications. Important features of our interception system include inline redirection of all COM object-instantiation functions, interception of COM interfaces through interface wrappers, accurate tracking of inter-

face ownership, and robust support for undocumented interfaces.

Our interception system has been tested on over 300 COM binary components, 700 unique COM interfaces, and 2 million lines of code. Using our interception system, the Coign ADPS has automatically partitioned and distributed three major applications including Microsoft PhotoDraw 2000.

While our interception system is COM specific, the techniques described are relevant to CORBA ORBs. For example, inline redirection and interface wrappers could be used to intercept Portable Object Adapter (POA) [20] functions and object invocations [22].

Bibliography

[1] Brown, Keith. Building a Lightweight COM Interception Framework, Part I: The Universal Delegator. *Microsoft Systems Journal*, vol. 14, pp. 17-29, January 1999.

[2] Brown, Keith. Building a Lightweight COM Interception Framework, Part II: The Guts of the UD. *Microsoft Systems Journal*, vol. 14, pp. 49-59, February 1999.

[3] Brown, Nat and Charlie Kindel. *Distributed Component Object Model Protocol -- DCOM/1.0*. Microsoft Corporation, Redmond, WA, 1996.

[4] Hartman, D. Unclogging Distributed Computing. *IEEE Spectrum*, 29(5), pp. 36-39, May 1992.

[5] Hunt, Galen. Automatic Distributed Partitioning of Component-Based Applications. Ph.D. Dissertation, Department of Computer Science. University of Rochester, 1998.

[6] Hunt, Galen. Detours: Binary Interception of Win32 Functions. *Proceedings of the 3rd USENIX Windows NT Symposium*, pp. *to appear*. Seattle, WA, July 1999.

[7] Hunt, Galen and Michael Scott. A Guided Tour of the Coign Automatic Distributed Partitioning System. *Proceedings of the Second International Enterprise Distributed Object Computing Workshop (EDOC '98)*, pp. 252-262. La Jolla, CA, November 1998. IEEE.

[8] Hunt, Galen C. and Michael L. Scott. The Coign Automatic Distributed Partitioning System. *Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI '99)*, pp. 187-200. New Orleans, LA, February 1999. USENIX.

[9] Kirtland, Mary. Object-Oriented Software Development Made Simple with COM+ Runtime Services. *Microsoft Systems Journal*, vol. 12, pp. 49-59, November 1997.

[10] Microsoft Corporation. *HookOLE Architecture*. Alpha Release, Redmond, WA, October 1996.

[11] Microsoft Corporation. *Internet Explorer*. Version 2.0, Redmond, WA, October 1997.

[12] Microsoft Corporation. Overview of the Corporate Benefits System. *Microsoft Developer Network*, January 1997.

[13] Microsoft Corporation. *Visual Basic Scripting Edition*. Version 3.1, Redmond, WA, October 1997.

[14] Microsoft Corporation. *ITest Spy Utility*. OLE DB SDK, Version 1.5. Microsoft Corporation,, Redmond, WA, January 1998.

[15] Microsoft Corporation. *PhotoDraw 2000*. Version 1.0, Redmond, WA, November 1998.

[16] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*, Redmond, WA, 1995.

[17] Narasimhan, P., L. E. Moser, and P. M. Melliar-Smith. Exploiting the Internet Inter-ORB Protocol Interface to Provide CORBA with Fault Tolerance. *Proceedings of the Thrid USENIX Conference on Object-Oriented Technologies*. Portland, OR, June 1997.

[18] Netscape Communications Corporation. *Netscape JavaScript Guide*, Mountain View, CA, 1997.

[19] Object Management Group. *The Common Object Request Broker: Architecture and Specification, Revision 2.0*. vol. Revision 2.0, Framingham, MA, 1995.

[20] Object Management Group. Specification of the Portable Object Adapter (POA). OMG Document orbos/97-05-15 ed. June 1997.

[21] Reed, Dave, Tracey Trewin, and Mai-lan Tomsen. Microsoft Transaction Server Helps You Write Scalable, Distributed Internet Apps. *Microsoft Systems Journal*, vol. 12, pp. 51-60, August 1997.

[22] Schmidt, Douglas C. and Steve Vinoski. Object Adapters: Concepts and Terminology. *C++ Report*, 9(11), November 1997.

[23] Vinoski, Steve. CORBA: Integrating Diverse Applications within Distributed Heterogeneous Environments. *IEEE Communications*, 14(2), February 1997.

[24] Wang, Yi-Min and Woei-Jyh Lee. COMERA: COM Extensible Remoting Architecture. *Proceedings of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pp. 79-88. Santa Fe, NM, April 1998. USENIX.

[25] Williams, Sara and Charlie Kindel. The Component Object Model: A Technical Overview. *Dr. Dobb's Journal*, December 1994.