INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Université d'Ottawa · University of Ottawa

Metamorphic Objects

For

Dynamic Reconfiguration of

CORBA-based Applications

By

Derek Shawn Elsaesser

A thesis submitted to the
School of Graduate Studies and Research
In partial fulfillment of the requirements for the degree of

Masters of Applied Science

Ottawa-Carleton Institute for Electrical Engineering
Department of Electrical Engineering
Faculty of Engineering
University of Ottawa
June 2000

Abstract

This thesis presents a solution to the problem of dynamic reconfiguration of distributed applications. This solution is based on the concept of a *metamorphic object*, which can dynamically change its interface, class definition, and implementation without invalidating existing client object references. The concept of object metamorphism is introduced and discussed. Forwarding is proposed as a mechanism for realizing metamorphic objects in a distributed computing environment and the *Forwarder* design pattern is presented. A framework for developing distributed applications using metamorphic objects is presented as a specification for a *Dynamic Versioning Service* for the Common Object Request Broker Architecture. A test application is developed to evaluate the versioning of metamorphic objects and to measure the impact on performance. It is found that the ability to transform an individual object to a new class definition and implementation, without interruption in service, results in only a slight increase in operation invocation time.

Acknowledgments

I would like to thank my supervisor, Dr. Dan Ionescu, for his guidance, encouragement, optimism and support throughout my research. His ability to set lofty goals and challenge one to achieve them is responsible for the magnitude of this thesis.

I also wish to thank the Defence Research Establishment Ottawa for supporting my Master's program by giving me the time and encouragement to complete it. Thanks to Dr. Richard Brown of Xwave Solutions Inc. for his help in reviewing and editing this thesis.

I have a special thank-you for my beloved wife Nancy Sagriff for her love and support throughout my research. She worked hard and tolerated much so that I could complete this work. I could not have done it without her, nor would I wish to.

Table of Contents

List of Figures

List of Tables

Glossary of Terms

ADT	Abstract Data Type
BOA	Basic Object Adapter
CORBA	OMG standard for distributed computing applications.
DII	Dynamic Invocation Interface. Runtime operation requests used by clients.
DSI	Dynamic Skeleton Interface for server implementation
DVS	A Dynamic Versioning Service for CORBA, used as a framework for developing distributed applications using metamorphic objects.
Forwarder	A servant that redirects operation invocations using the Forwarder Pattern.
GIOP	General Inter-ORB Protocol
IDE	Integrated Development Environment
IDL	Interface Definition Language defined by the OMG for CORBA.
IFR	Interface Repository that registers IDL specifications.
IIOP	Internet Inter-ORB Protocol
Implementer	A servant that provides the application functionality for an IDL interface.
IOR	Interoperable Object Reference for objects in separate CORBA processes.
IR	Implementation Repository that registers CORBA server executables.
ISO	International Standards Organization
Metamorphic Object	An object which can dynamically change its interface, class definition, and implementation.
MO	An instance of the MetamorhicObject base-class or a derived class.
PO	Persistent Object
POA	Portable Object Adapter
POS	Persistent Object Service as specified by the OMG.
OMA	Object Management Architecture
OMG	Object Management Group.
OODBMS	Object-Oriented Database Management System
OQL	Object Query Language

ORB	Object Request Broker, the CORBA process that provides communications between client and server processes.
OTS	Object Transaction Service
RDBMS	Relational Database Management System
Servant	A programming language implementation class that provides the functionality for an IDL interface.
SII	Static Invocation Interface. Compiled proxies used by client processes.
SQL	Sequential Query Language
SSI	Static Skeleton Interface for server implementation.
TCP/IP	Transmission Control Protocol/Internet Protocol
TMN	Telecommunication Management Network
UML	Unified Modeling Language.
Versioning	The transformation of a metamorphic object from one definition and implementation to another.
VMS	Version Manager Service, part of the Dynamic Versioning Service.

1 Introduction

1.1 Motivation and Research Objectives

As we enter the 21st century, Information Technology is pervasive throughout our society. This technology continues to advance at a rapid rate, resulting in fast growth of system complexity and an ever-increasing need for applications to evolve over time. A result of this evolution is the trend towards distributed computing in a heterogeneous environment, such as the Internet or World Wide Web. In an effort to manage the complexity of developing these systems, a consortium of major software vendors and developers, the Object Management Group (OMG), has produced a standard for distributed computing called the Common Object Request Broker Architecture (CORBA) [19]. The basic concept behind CORBA is that server and client processes use a software component called an Object Request Broker (ORB). An ORB enables simple message passing between objects, even if these objects reside on different machines. The network complexity is dealt with by the ORB [7]. CORBA also encourages the development of open applications by requiring distributed objects or components to be defined at a functional (virtual) level by their interface in a common language, the OMG's Interface Definition Language (IDL). This enables the implementation of distributed systems to use different programming languages and different operating systems and hardware platforms. CORBA is predicted to be the foundation for many large scale, or "Enterprise", distributed applications for the World Wide Web, electronic commerce, business, telecommunication management networks, and many others [6, 7, 35, 44, 46].

An aspect of these large distributed systems not directly addressed by the CORBA specification is, how are mission critical applications to be modified without an interruption in service? As technology continues to advance, these applications will constantly be required to evolve their functionality to address new user requirements and to upgrade hardware and software platforms, including the ORB itself, to exploit new

technologies. CORBA systems are typically comprised of a large number of client applications, residing on remote platforms, accessing a large number of objects that exist in multiple server applications, again residing on separate platforms, over a network. This results in a large number of object references between client applications and server objects. If the server objects are to be modified, to add new functionality or to change the underlying software implementation or hardware platform, all connections between the clients and the servers providing these objects must be broken. Depending on the nature of the application, it may not be possible to restore all of the client-server object references to their previous state. This can result in a significant loss of service to system users.

The CORBA specification enables the interface for a new object class to be defined dynamically. However, it does not address the problems of how to migrate dynamically existing objects to instances of these new classes and maintain valid object references between clients and server objects, without an interruption in service caused by shutting down and restarting servers. A Dynamic Versioning Service is required for dynamically evolving objects in distributed systems. The Dynamic Versioning Service is proposed in this thesis to perform the following functions:

- Versioning of class definitions and maintaining consistency of the class hierarchy across all servers in the network;
- Determine the translation scheme from one version of a class to another;
- Transform the state of an instance of one class to an instance of another version of the class;
- Migrate the persistent storage of the object state to the persistent store of the new version;
- Versioning selected object instances on selected servers to the new class implementation;
- Maintain object reference consistency for all affected clients to redirect all messages to the new instantiation of the object; and

- Provide a means for clients to re-establish a direct connection to the new version of the object.

Note that the Dynamic Versioning Service is not responsible for any modifications to the client application required to use any new functionality offered by the new version of the object class or handle exceptions resulting from the removal of attributes or operations in the new class version.

An example of a distributed system requiring such a Dynamic Versioning Service is the "BaseLayer" CORBA-based network management system developed at the Machine Intelligence Research Laboratory, University of Ottawa [12, 13, 14]. One of the goals of the BaseLayer application is to develop a methodology for dynamically modifying the interface and implementation of objects in a distributed network management environment, without interrupting network services or introducing inconsistencies in the network. As objects are instantiated on CORBA servers and managed by remote clients through CORBA object references, these references must remain valid during and after the versioning process. As new classes or sub-classes are added to the system, each factory server must instantiate the correct class and version in a given hierarchy when creating a new object for a client. To maintain the inheritance structure and versions of classes across a network, the changes to a class definition must be propagated to all the hosts in the network. The system must ensure that all client references remain valid and that the referenced object will behave in a sound manner.

This thesis presents a Dynamic Versioning Service to support the evolution of objects in a distributed environment. A theorem is proposed for versioning active instances of objects from one class to another. A framework is presented for dynamically transforming CORBA objects between class versions and forwarding client references to the new version of the object. An implementation of this framework is constructed to demonstrate dynamic versioning. Results are presented on the impact to performance and network integrity caused by the dynamic versioning of Metamorphic Objects. Finally,

recommendations are made for a Dynamic Versioning Service for future implementations of CORBA.

1.2 Organization of the Thesis and Contributions

This thesis is organized to provide an incremental understanding of the problem of object evolution in a distributed environment, including the concept of a metamorphic object. A mechanism is proposed for dynamic versioning of metamorphic objects, and implementation results are presented.

Chapter 2 describes the need for and the problems of object evolution in a distributed environment. Recent work in the dynamic reconfiguration of distributed applications is reviewed. The requirements for dynamic object evolution in mission-critical applications are defined.

Chapter 3 presents an overview of the Common Object Request Broker Architecture. Because CORBA is both the application environment and the foundation of the concepts proposed in this thesis, it requires a detailed review. Many of the concepts that support metamorphic objects and the Dynamic Versioning Service are based on the concepts, principles, and functions defined by version 2.0 of the CORBA standard. Examples from existing literature on various applications of CORBA that relate to the topic of this thesis are discussed. The aspects of CORBA that are applied in subsequent chapters are presented in detail while other concepts are mentioned for completeness.

Chapter 4 describes a new concept in object-oriented programming, a *Metamorphic Object* that can dynamically change its definition and implementation. Forwarding is proposed as a mechanism to realize Metamorphic Objects in a distributed environment. The concept of dynamically versioning Metamorphic Objects is described and the criteria for mapping a Metamorphic Object between class definitions is presented.

In Chapter 5, a framework is presented for a Dynamic Versioning Service for CORBA. The *Forwarder Pattern* for metamorphic objects is introduced as part of the design of this framework. Other components of the Dynamic Versioning Service, such as the persistent store for metamorphic objects, are included in the design. A specification for a complete Dynamic Versioning Service is presented using the Interface Definition Language and a detailed design of the core components are presented using the Unified Modeling Language (UML).

An implementation of the Dynamic Versioning Service that verifies the operation of Metamorphic Objects is presented in Chapter 6. Results are presented on the impact to performance and network integrity caused by dynamic versioning. The impact of Forwarding on operation invocation times is measured along with the time spent accessing the persistent store and making remote invocations.

Finally, in Chapter 7 conclusions are presented on the costs and benefits of Metamorphic Objects. The impact of the Dynamic Versioning Service prototype on large scale distributed systems is discussed. Recommendations are made for a Dynamic Versioning Service for future distributed applications using CORBA to support Metamorphic Objects.

This thesis contains several research contributions, which can be summarized as follows:

1. An innovative concept in object-oriented programming, a Metamorphic Object, is presented for the first time.

2. The technical aspects of the CORBA standard that are relevant to supporting Metamorphic Objects are examined and concepts for applying them to implement the Dynamic Versioning Service are presented.

3. A formal description of Metamorphic Objects in a distributed environment is presented in chapter 4. The theory for transforming object instances between different

interface and class definitions is presented. The semantics and precept for invoking operations on a new version of a metamorphic object using the original object reference and interface is also presented.

4. A framework for a Dynamic Versioning Service is described in chapter 5. The *Forwarder Pattern* for realizing Metamorphic Objects is presented. The requirements of a persistent store for the Dynamic Versioning Service are also presented.

5. A fully functional prototype of a Dynamic Versioning Service for CORBA is constructed. A test application is produced and tested. The impact of dynamic versioning of Metamorphic Objects on network consistency and performance is demonstrated.

6. Recommendations are made for future research in Metamorphic Objects.

2 Dynamic Reconfiguration of Distributed Applications

This chapter describes the need for and the problems of object evolution in a distributed environment. Related work in the field of object evolution in distributed systems is discussed. The requirements for object evolution in mission-critical applications are defined.

2.1 The Problem of Object Evolution in a Distributed Environment

A distributed object-oriented application typically consists of multiple client processes referencing objects on remote server processes. Like stand-alone software packages, distributed applications tend to evolve over their lifetime to address changes in user requirements, technological changes, or changes in the execution environment. This evolution can include changes related to: (1) the application architecture, (2) the software implementation, or (3) the physical location of software components in a distributed environment. The problem of implementing changes in a distributed computing environment has been termed "distributed application reconfiguration" [21]. Because these changes are applied to an application consisting of multiple dependent processes, reconfiguration needs to be conducted *dynamically*, without shutting down parts of the network or blocking uninvolved components. An environment or technique is required that allows dynamic reconfiguration of distributed applications with minimum disruption in service.

Research in the dynamic reconfiguration of distributed applications has led to the classification of four categories of change [21]:

- **Structure**: This change affects the application structure by adding or removing services or components. These changes result in a new architecture for the application or in a new version or configuration.

- **Geometry**: The application structure remains unchanged but its mapping onto a distributed architecture is modified. Components are migrated to accommodate node failures, new nodes included in the system, etc. Geometric reconfiguration is useful for load balancing, security, adaptation to changes in communications resources, etc.
- **Implementation**: The application's overall structure remains the same but a user requires a new implementation of an application component.
- **Interface**: The set of operations provided by a component is modified, for example, to offer more services.

The evolution of the application architecture involves the definition of a new object model containing new class definitions and associations. The implementation of the new object model may involve modification to the existing code-base or a new implementation in a different programming language. With changes in geometry, the physical relocation of software components, or even individual objects, may involve executing the component on a new host machine or new operating system. Making any changes to a distributed application to support this evolution can have a dramatic impact on the entire system. Remote client processes that maintain references to objects on a server process that is terminated to enable new code are left with invalid, dangling, object references. This can cause significant perturbations throughout the system as client processes encounter unexpected object faults. Even though distributed applications are designed and implemented with some degree of fault tolerance for object faults, significant changes to a distributed application, especially the object model, may result in severe inconsistency and instability in the application.

To illustrate the problem encountered by invalidating object references, an example of a simple distributed system using CORBA is shown in Figure 2-1. This figure shows two client processes holding references to objects on two different servers, "MO_A" and "MO_B". It also shows the registration of these object references with the CORBA Naming Service, which allows any client to obtain a reference to the server objects without direct interaction with the server process. If a server was shut down, even

momentarily, to modify an object interface or implementation, any number of object references in both clients could become invalid. Even if an object reference held by clients became valid again after being rebound in the Naming Service, as with the approach taken by [21], the service disruption may be unacceptable and the network application may be in an inconsistent state.

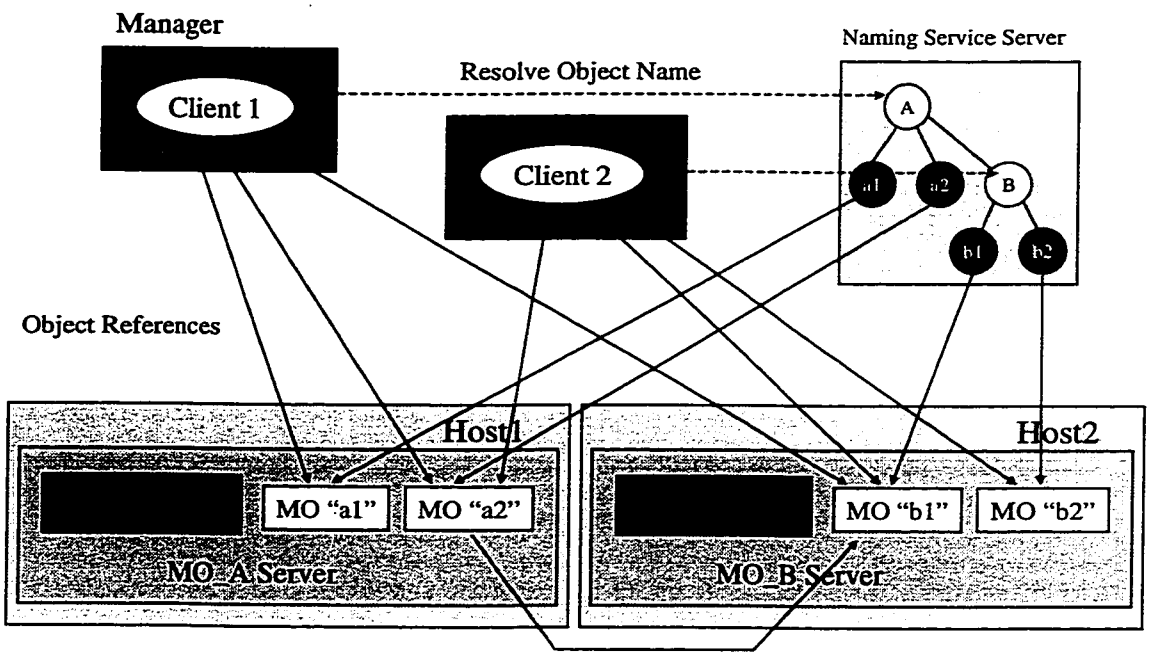

Figure 2-1 Object References in a Distributed Application.

Modifications to a distributed application typically requires that the entire application, including all server and client processes, be shutdown and restarted with the new application architecture, software implementation, or physical deployment. As distributed applications increase in size and complexity, this approach becomes increasing costly in terms of time, application availability, and software management. For mission critical applications, including distributed real-time systems such as for Telecommunication Management Networks (TMN) [11, 12], unavailability of even a portion of the distributed application is unacceptable. A means for evolving such applications, without any interruption in service or perturbation in the system, is essential. Thus, software components, ranging from all processes on a specific host to individual instances of a

specific class in a server process, must be **incrementally evolved** (in terms of architecture, implementation, and deployment) without invalidating existing object references in the client processes.

Many industries, such as e-commerce or telecommunications, could benefit from the ability to incrementally evolve distributed applications without a loss in service. For example, one of the most difficult problems involved in adding new services and capacity to a Telecommunication Management Network is the interruption in customer service. If a TMN could be modified incrementally without loss of service, more services could be added to future telecommunication systems with less disruption in service than current systems. Such a technology would provide a significant advantage to firms developing these systems.

2.2 Related Work

Distributed application reconfiguration has been the area of much research over the past decade involving many different, non-CORBA, programming environments including Polylith [25], Durra [22], Argus [23], and Conic [24]. These environments used one of two approaches to implementing a reconfiguration mechanism: modification of the application source code (Pollith, Durra), or implementation of an application independent reconfiguration mechanism (Argus, Conic). The first approach required the application developer to predict where in the code reconfiguration may occur; the second approach required blocking execution of large portions of the application, resulting in significant disruptions. Some aspects of these approaches or implementation limited the evolution of these projects. With the emergence of CORBA as widely accepted standard for distributed computing, a mechanism for dynamically reconfiguring distributed applications using the facilities provided in CORBA is required. It must not invalidate the CORBA object model or require custom, non-standard implementations of the architecture.

2.2.1 Rebinding of references using an Extended Naming Service

Recent work in dynamic reconfiguration of CORBA-based applications has been conducted for France Telecom/CNET [21]. This work describes the problems of object replacement, addition, removal or migration, and binding modification and addition. Object addition and removal require no special actions; they are simple creation and destruction operations. Object replacement is implemented as an addition followed by a removal. This is complicated if the internal state of the object must be transferred; however this is not addressed. The work in [21] focuses on the problem of migrating an object from one implementation and location to another and modification of client bindings to the new implementation.

Object migration, as defined in [21], involves a number of actions: suspend execution of the CORBA object, save its current state, re-create the object at a remote site, restore its state, and finally resume execution and change all object bindings to reference the new execution context. The approach taken attempts to solve the problem of object migration by automatically rebinding object references in client processes to the new version of the target object [21]. The rebinding of object references is accomplished at the architecture level by introducing an IDL post-processor to modify the client stub and server skeleton code (described in Chapter 3). The code that is embedded in the client proxy objects enables them to handle the object fault exceptions without modification to the application source code. This embedded code, called the 'configurable stub', processes all requests to the target object and catches the exception raised by an invalid object reference. It then blocks the calling object and its process until the new object reference is available from an Extended Naming Service. The new reference to the target object is then bound in the client stub code and the client process is unblocked allowing requests to be invoked on the new target object.

The object migration algorithm used by [21] is comprised of four steps.

- Step 1: The target CORBA object switches from the execution state to the reconfiguration state. The object is blocked; it can not receive or initiate new

requests. Any messages sent to the object while in this state are rejected and an exception is returned. Any client object receiving this exception consults the Extended Naming Service to locate the new implementation of the target object. The Extended Naming Service blocks the client object until the end of the target object's migration.

- Step 2: The current state of the target object is saved. Serialization is used to save the object's state; no transformations are performed.
- Step 3: The object is moved to a new site and re-created from its stored state, the result being an identical copy of the original object. There is no change in the object's interface or functionality. The object now has a new IOR (see section 3.5) and informs the Extended Naming Service of this reference. This implies that some connections are no longer valid and client objects that were not blocked by the Extended Naming Service now have erroneous bindings.
- Step 4: The target changes its state from reconfiguration to execution state. The Extended Naming Service updates the IOR for all blocked client objects. Any invalid references by clients that were not blocked are updated incrementally when the client encounters an object fault.

This approach successfully enabled objects to be migrated to new processes without modifications to client source code. However, it did encounter "weak execution disruption" during reconfiguration [21]. It was also found that dependent requests, whose completion depends on the completion of another request, could result in the blocking of a chain of objects during reconfiguration.

Although the work in [21] proposes a mechanism for migrating objects to new implementations, it does not address all of the aspects of dynamic reconfiguration of distributed systems. In particular, it does not address the problems of dynamically changing an object's interface or functionality. The implementation of this migration mechanism requires modification to the architecture of the CORBA proxy and skeleton objects, making them non-standard. This may be unacceptable for large commercial applications. Finally, the modification of client object references was found to cause

disruptions and inconsistencies throughout the network. A better approach is to avoid the problem of invalidating object references in the first place, as proposed in Chapter 4.

2.2.2 Cross-version mapping using a mediator

Other recent work in software evolution for distributed services proposes a *mediator* mechanism to enable clients to invoke requests on a new version of a service [20]. The mediator considers functionality compatibility between versions, rather than operation signature compatibility, when mediating the request. Thus, instead of forcing change on the client side, this model allows flexible interoperability between different versions of client and server software. The mediator uses mapping information to translate a request to and from and a new object interface. This mapping is transparent to the client.

A service is defined by [20] as an object in a distributed system that is described by its interface signature and is available for use by clients which know the calling protocol of the object, i.e. what operations to invoke and what parameters to supply. An example of a service object is a network print service. A new *version* of a service is defined when an object evolves in a way that its interface is changed. This may require clients to evolve according to the protocol that has changed if they want to continue using the service. Evolution at the client side means change in all client programs that are scattered in the system. It is not easy to achieve to evolve clients, as it requires time to locate and change all clients using the service and their normal operation would be interrupted. The mediator mechanism attempts to overcome these problems through evolution transparency without client conversion by making the evolved service version appear as if it was the original version. The old-version client program will still operate with the new service version and cross-version interoperability is achieved.

This mediator model can be applied to any distributed object architecture that employs a type (interface) repository to store object interface definitions. This includes RM-ODP [48], CORBA [49], DCE [50], and COM/DCOM [51]. Most of these distributed object

architectures support evolution transparency in a restricted way by allowing only incremental changes to be made on a service, usually by allowing a subtype (interface subclass) to be defined. CORBA recognizes version relationships between services of the same name but no versioning semantics is defined. There are no specifications for substituting one version of an interface for another and there are no known CORBA vendors that support multiple versions of the same interface name in the same interface repository [52]. This requires that new versions of an interface be implemented as a subtype of an existing interface. COM/DCOM does support multiple interface versions but does not allow them to be substituted for one another [51]. The mediator mechanism provides evolution transparency for clients of evolved services using the version substitutability approach. It employs an interface naming convention for service types in the interface repositories used by different distributed computing architectures.

The mediator mechanism uses a *mapping operator* that upgrades an old-version operation request to a request for the new service. This concept is based on works in managing schema evolution in object-oriented databases [53, 54] and DRASTIC [55], which is a persistent Java platform that supports substitutability between service versions. DRASTIC defines a contract that simply describes how to map a client request of one version of a service to another. The mapping operator approach uses a mediator object that is situated between the old-version client and the new-version service, as shown in Figure 2-2. To be able to intercept operation requests from the old-version client without alteration on the underlying invocation protocol, the mapping operator is disguised as an instance of the old service. To use the new-version service, the client must bind to the mapping operator object. The client can then invoke an operation on the new-version service using the protocol of the old-version service by sending the request to the mapping operator. The mapping operator does not execute the request but upgrades the request and diverts it to an instance of the new version service.

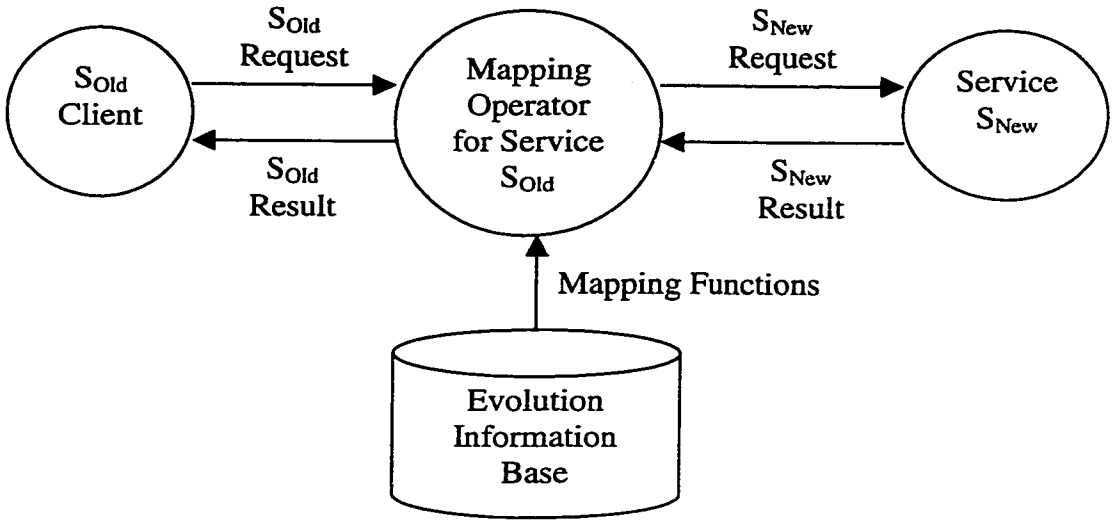

Figure 2-2 Transformation of messages using the mediator mechanism.

The upgrade of the service request is two-way: the mapping operator transforms both request and result for the existing client. It uses *mapping functions* which are stored in a repository. These mapping functions are defined by the developer based on the syntactic and semantic compatibilities between the old and new interface definitions. The mediator mechanism is implemented on the ANSAware platform [56] for the RM-ODP architecture and includes a tool called an evolution manager that maintains all necessary evolution information for the production of mapping functions and instantiation of mapping operators.

The functions of a mapping operator and the criteria for mapping functions are discussed in [20]. It describes the classification of interface evolution in the ANSAware system and proposes a list of changes to a CORBA interface to support the mediator mechanism. It also discusses the propagation of interface evolution in a distributed environment and extensions to the type repository in the RM-ODP architecture and CORBA to support object evolution. The mapping operator is identified as adding a substantial overhead to operation invocations because each old-version invocation must be mapped to a new-version operation and a new request must be constructed and executed on the new-

version service. The result must also be mapped to the signature expected by the old-version client. Since a mapping operator must be supplied for every instance of an old-version service that is being upgraded, it is best suited for applications involving a large number of client programs using a relatively small number of service objects where invocation time is not critical. Also, since the mediator approach treats different versions of a service as distinct entities, it does not deal with the transfer of a service object's state from one version to another.

2.2.3 Signature matching using group service abstraction

Other work in the dynamic reuse of services in distributed systems has been conducted that investigates the mapping of interface specifications between versions [57]. The aim is to enable the dynamic reuse and accommodation of services to cope with the diversity of object implementations and specifications. A *service group* abstraction is proposed that relates different services with different interface definitions by role. An extension to IDL for CORBA called the *Group Description Language*, is presented for specifying the behaviour and role of objects in a service group. Client applications can then be written to interact with a service group via a higher-level interface. Such an interface contains service signatures defined to fit a particular context, and mappings are defined for particular services. Formal definitions are provided for service signature matching and role conformance for the service mapping. The limitation of compatible interfaces being restricted to subtypes (as required in CORBA and RM-ODP) is avoided. A type manager is proposed which permits the addition and deletion of various compatibility relationships between types in a generic type repository. This is similar to the interface repository requirements specified in [20]. However, the Java prototype described implements only base classes to support service groups and does not address the mechanisms required to dynamically redirect an operation request to a different version of a service without invalidating client object references.

2.2.4 Deficiencies of Previous Approaches

Object evolution in distributed applications requires that objects be capable of dynamically changing their structure, geometry, implementation, and interface without invalidating existing client references to these objects or causing an interruption in service. The research conducted has been unable to identify an existing mechanism for object evolution that addresses all of these requirements.

The approach taken with the Extended Naming Service in section 2.2.1 enables an object to change its geometry and implementation but does not address the requirement to change the object's interface and functionality. Although this approach provides a mechanism for re-establishing client references to an object that has been versioned, it still causes invalid object references that can result in a loss of service between distributed objects. The Naming Service in CORBA could provide a mechanism for clients to obtain references to a new version of an object under program control. Each client could obtain a new object reference from a symbolic name binding when it is suitable for the client to do so, not as a result of an object fault causing an exception. This would avoid the interruption in service experienced by [21].

The mediator approach in section 2.2.2 provides a solution to changing the structure, geometry, implementation, and interface of a service object. It addresses some of the requirements for providing an interface repository capable of managing multiple versions of an interface with the same name but different signatures. It describes the need for a configuration service to manage the mapping between different versions of compatible interfaces. However the mapping operator is identified as adding a substantial overhead to operation invocations because requests and results must be translated dynamically. The requirement for a separate mapping operator for every instance of an old version of a service object makes it suited for applications involving a large number of client programs using a relatively small number of service objects where invocation time is not critical. Also, the mediator does not address the transfer of an object's state from one version to another; it treats a new version of a service object as a new entity.

Much research has been conducted in defining the mappings between different interfaces and class definitions [20, 21, 22, 23, 24, 25, 27]. This includes the versioning of Object-Oriented database schema [32, 53, 54], design patterns that use sub-classing and polymorphism to invoke operations of different objects [34, 35, 36], and abstract data types [26, 28, 31]. Recent work in the dynamic evolution of distributed systems, specifically [20] and [21], have also attempted to address the problems of dynamically changing the interface or implementation of an object without an interruption in service. Each of these approaches has merit but fail to address all of the requirements for dynamic object evolution for mission-critical real-time distributed applications, such as telecommunication management networks. The concept of a Metamorphic Object is presented in Chapter 4 to address these requirements.

3 CORBA For Distributed Applications

This chapter presents an overview of the Common Object Request Broker Architecture. Because CORBA is both the application environment and the foundation of the concepts proposed in this thesis, it warrants a detailed review. Many of the concepts that support the dynamic reconfiguration of distributed applications are based on the theories, principles, and functions defined by version 2.0 of the CORBA standard. Examples from existing literature on various applications of CORBA that relate to the topic of this thesis are discussed. The aspects of CORBA that are applied in subsequent chapters are presented in details while other concepts are presented for completeness. The technical aspects of the CORBA standard that are relevant to supporting metamorphic objects are examined and concepts for applying them to implement a Dynamic Versioning Service are presented.

3.1 The Common Object Request Broker Architecture

A distributed object system consists of multiple client programs referencing objects on remote servers. CORBA is a non-proprietary standard for developing distributed object systems. The CORBA standard was developed by the OMG comprised of representatives from most major software companies. The concepts proposed in this thesis are based on revision 2.0 of the CORBA standard dated July 1995 [19]. However, references to features of the new version 3.0 of CORBA, released in 1999, are made to identify how the concepts of this thesis can be applied to new CORBA implementations. Version 3.0 of CORBA was not finalized at the time this research was started and a stable commercial implementation was not available during development of the DVS. Investigation of the version 3.0 of CORBA for metamorphic objects is recommended for future research.

3.1.1 The Object Management Architecture

The Object Management Architecture (OMA) defines the functions and relationships between the main components that make up an implementation of the CORBA standard. The OMA, shown in Figure 3-1 [7, 19], is comprised of the core CORBA infrastructure which provides the basic ORB functionality, standards for **CORBAservices** that extend ORB support for applications, and the user applications. The OMA also defines a number of CORBA frameworks, called **CORBAfacilities**, for specific domains such as e-commerce or telecommunications. The BaseLayer application [13, 14] is an example of a CORBAfacility implementation for Telecommunication Management Networks (TMN).

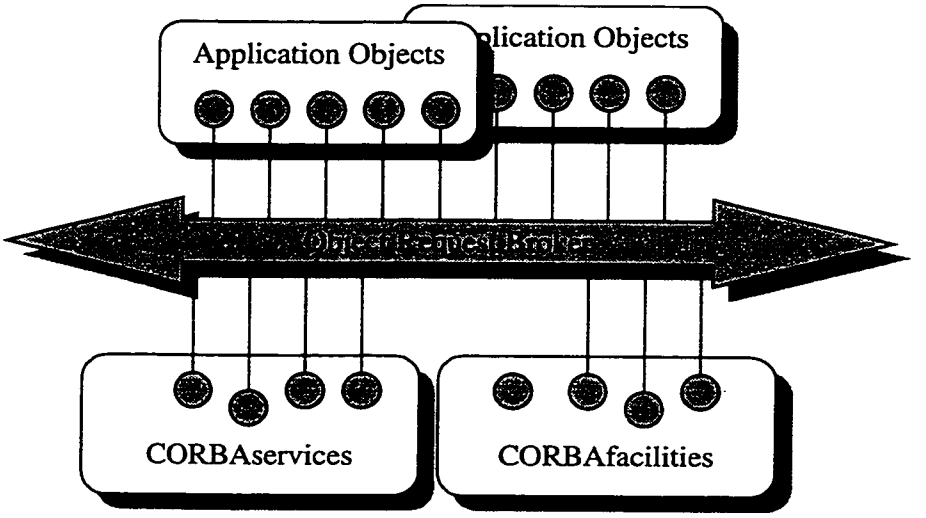

Figure 3-1 The Object Management Architecture (OMA).

This section briefly describes the specific features of CORBA and CORBAservices that are of interest to this thesis. As the complete OMA and CORBA specifications are quite extensive and freely available from the OMG, they will not be fully described here.

3.2 Concepts and Terminology

CORBA provides platform-independent programming interfaces and models for portable distributed object-oriented computing applications. Its independence from programming languages, computing platforms, and network protocols makes it highly suitable for the development of new applications and their integration into existing distributed systems.

CORBA uses a terminology derived from other related technologies with a few new terms. The most important terms required to understanding CORBA systems are explained in the following list [1, 6, 7].

- An *Object Request Broker (ORB)* is a software component that resides with or near every client application and object. The ORB receives method invocation requests from the client and delivers them to the target object. If the client and target object exist in separate memory spaces, there is an ORB associated with the client process and a separate ORB associated with the server process for the target object. As part of the Common ORB Architecture, all ORBs, regardless of vendor, hardware platform, or software language implementation, can process standard CORBA requests between clients and server objects.
- A *CORBA object* is a "virtual" entity capable of being located by an ORB and having client requests invoked on it. It is virtual in the sense that it does not really exist unless it is made concrete by an implementation written in a programming language. The realization of a CORBA object by programming language constructs is analogous to the way virtual memory does not exist in an operating system but is simulated using physical memory [1]. The fact that CORBA distinguishes between an object's virtual interface and it its concrete implementation is a useful construct that is exploited by the Dynamic Versioning Service.
- A *target object*, within the context of a CORBA request invocation, is the CORBA object that is the target of that request. The CORBA object model is a single-dispatching model in which the target object of the request is determined solely by the object reference used to invoke the request. As will be shown later, the maintenance

of the object reference during versioning of the target object is a key component of this thesis.

- A *client* is an entity that invokes a request on a CORBA object. A client may exist in an address space that is completely separate from the CORBA object, or the client and the CORBA object may exist within the same application. An application that acts as the client in the context of a request may act as the server for another request.
- A *proxy* is an object that acts as a representative or stand-in for a remote object. Client applications may use proxies in their memory space in order to pass client requests to a CORBA object on a remote server. This enables the client application to use the CORBA object in the same manner as local objects, making the network communications transparent to the developer or user. Proxies are used as part of the client-side Static Invocation Interface, discussed later.
- A *server* is an application in which one or more CORBA objects exist. Clients invoke request upon server objects. A server application may also act as a client to another server.
- A *request* is an invocation of an operation on a CORBA object by a client. A Request flows from a client to the target object in the server, and the target object returns a result if the request requires one.
- An *object reference* is a handle used to identify, locate, and address a CORBA object. To clients, object references are opaque entities. The CORBA object model makes the location of a CORBA object transparent to the caller. Clients use object references to direct requests to objects, but they cannot create object references from their constituent parts, nor can they access or modify the contents of an object reference. An object reference refers only to a single CORBA object, and must be unique within the scope of its domain. The CORBA object model requires that a particular object reference must denote the *same* object throughout the object's lifetime. Once an object is destroyed, all its references must become permanently non-functional.
- A *servant* is a programming language entity that implements one or more CORBA objects. Servants are said to *incarnate* CORBA objects because they provide concrete implementations for those objects. Servants exist only within the context of a server application. In C++ or Java, servants are object instances of a particular class.

There can be multiple classes of servant objects associated with a class of CORBA objects, but only one servant instantiation may be associated with a CORBA object at a time. Client applications need not know how servants are implemented.

- *Creation* is the initial instantiation of a virtual CORBA object and its object reference. From a client perspective, CORBA objects are normally created by invoking normal CORBA operations on a *factory object* [8] within a server. Creation of a CORBA object may or may not include the *activation* of the object.
- *Activation* is the act of starting an existing CORBA object to allow it to service requests. Activation does not imply CORBA object creation since a CORBA object cannot be activated if it does not already exist. Activation may cause the creation of the servant.
- *Deactivation* is the act of shutting down an active CORBA object by removing its association with its servant. Deactivation does not imply CORBA object destruction, as the CORBA object continues to exist as a virtual entity and can be reactivated at a later time. Deactivation may result in the destruction of the servant.
- *Destruction* is the act of removing the CORBA object from existence. All object references to a CORBA object that has been destroyed are invalid.
- *Incarnation* is the act of associating a concrete servant object with a CORBA object so that it may service requests. This requires the instantiation of a servant object on the server and its association with the CORBA object. In other words, incarnation "gives bodily form or substance to" [39] a virtual CORBA object.
- *Etherealization* is the opposite of incarnation and refers to the act of destroying the instance of the servant of a CORBA object. That is, etherealization takes away the "body" of the CORBA object on the server. The CORBA object still exists as a virtual entity and may again be reincarnated when it activated by a client.

It is important to distinguish between the lifecycle of a virtual CORBA object and a concrete servant object. The terms activation, deactivation, creation and destruction are used to refer to the changes in the state of the CORBA virtual object. The terms incarnation and etherealization refer to the association and disassociation of a servant

object with a CORBA object, respectively [6]. The lifecycle of a CORBA object is shown in Figure 3-2.

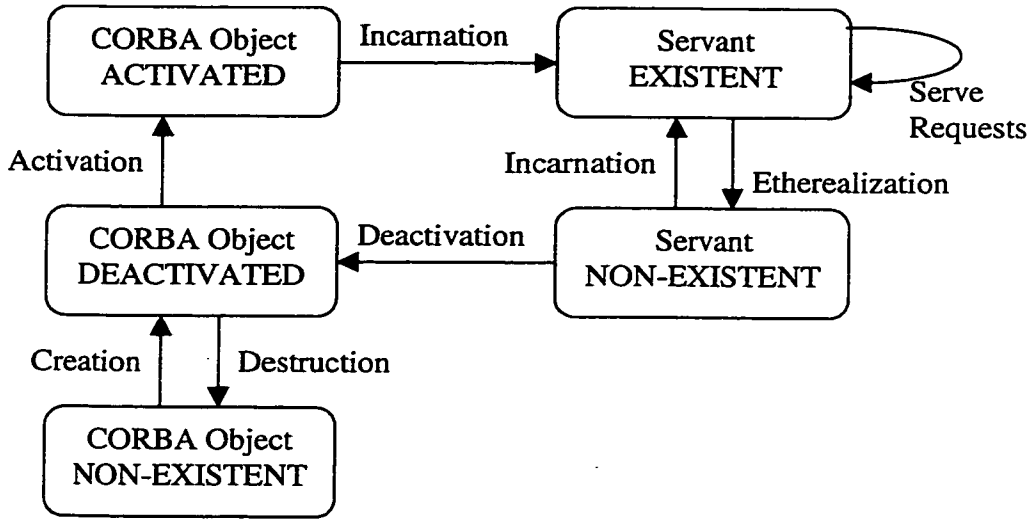

Figure 3-2 Life Cycle of CORBA Objects and Servants

3.3 The Object Request Broker

The core CORBA infrastructure is comprised of the Object Request Broker (ORB), its underlying inter-ORB communications protocol, the client-side interface, and the server side interface. These are shown in Figure 3-3 [7].

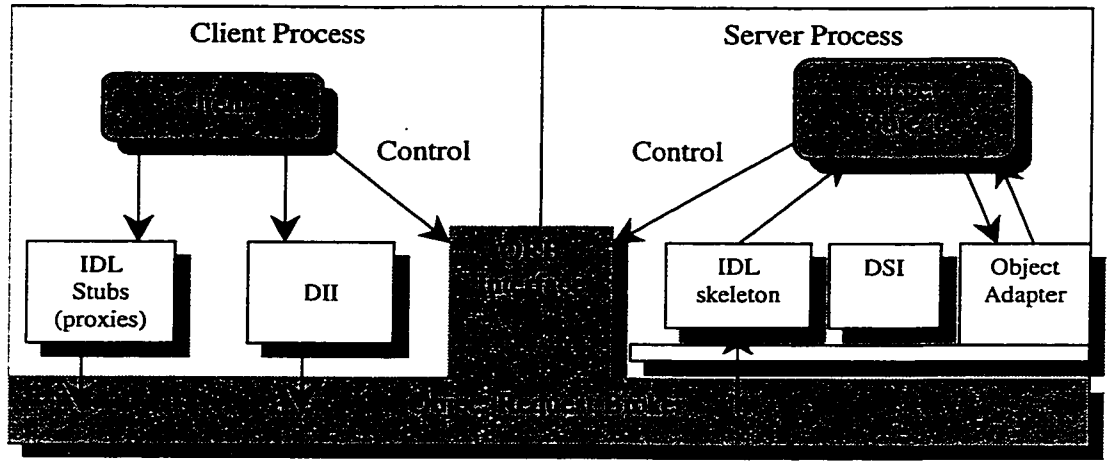

Figure 3-3 Client and Server sides of the ORB.

The ORB is required to allow a client to make a request on an object across a network, between operating systems, between different programming languages, and between ORBs from different vendors. This is accomplished by using a standard protocol for message passing between ORBs called the General Inter-ORB Protocol (GIOP). As of 1996, each ORB implementation is required to support the Internet Inter-ORB Protocol (IIOP) for passing GIOP messages over Transmission Control Protocol/Internet Protocol (TCP/IP) networks [19, 45]. There are other environment-specific inter-ORB protocols, such as for real-time systems; however, this thesis will limit the discussion to ORBs using IIOP.

The components of the ORB are presented in the following sections. As this thesis presents a Dynamic Versioning Service for server-side implementations, the functions of the ORB are important to this discussion.

3.3.1 The Object Adapter

In CORBA, object adapters serve as the glue between servants and the ORB. The intent of an object adapter, as defined by [8], is to adapt the interface of one object to a different interface expected by a caller. The object adapter enables a client to invoke requests on a

servant object without knowing the object's true interface, only its CORBA interface [7]. In C++, servants are instances of C++ objects. They are derived from skeleton classes produced by compiling IDL interface definitions. To implement operations, the servant class must override the virtual functions of the skeleton base class. The C++ servants are registered with the object adapter to allow it to dispatch requests to the servants when clients invoke requests on the CORBA objects incarnated by those servants.

The object adapter fulfills three key requirements of the ORB [7].

1. It creates object references, which allow clients to address objects.
2. It ensures that each target object in incarnated by a servant.
3. It takes requests received by a server-side ORB and further directs them to the servant incarnating each of the target CORBA objects.

In version 2.0 of the CORBA specification, the standard object adapter required by all ORBs is the Basic Object Adapter (BOA). There can be other forms of an Object Adapter, such as a Database Adapter that can be used to provide persistence to CORBA objects using an Object-Oriented Database Management System (OODBMS) [7, 41]. In the new CORBA 3.0 specification, the BOA will be replaced with the Portable Object Adapter (POA). The POA specification provides a full suite of features and services intended to allow developers to write scalable, high-performance server applications. This thesis is based on the CORBA 2.0 specification and does not address the functions of the POA, although many of the concepts proposed may be applied to the POA.

3.3.2 Server-side Static Skeleton Interface

In the development of server applications, skeleton classes for servant objects are produced by compiling IDL interface definitions. The servant class implements the operations for the IDL interface by overriding the virtual functions of the skeleton base class. This approach is known as the Static Skeleton Interface (SSI) and is by far, the most common approach for developing server applications. The SSI provides static

mapping between the IDL interface and implementation class. The Object Adapter uses this mapping for invoking the incoming request for a CORBA object on its underlying servant object. When using the SSI, the Object Adapter implements the adapter pattern as described in [8] to map an operation request from the IDL interface to the servant implementation class.

The adapter pattern has two possible structures, one using inheritance and one using delegation, to pass an invocation on an interface object to a servant object [8]. Both these patterns can be used in the design of the Static Skeleton Interface for server-side application classes. In developing C++ servants for CORBA 2.0 using Orbix version 2.3 [41, 42], these patterns are known as the BOAImpl and the TIE approaches, respectively. These patterns only apply to developing servants for server applications; they do not apply to client applications.

In the BOAImpl approach, shown in Figure 3-4, the inheritance form of the adapter pattern is used. The IDL compiler produces the CORBA proxy class that is used by the client application, and the SSI class for the server application. This SSI class is called the "BOAImpl" class in Orbix as the class name is suffixed with BOAImpl. The C++ servant class must inherit from the BOAImpl class to provide an implementation for the CORBA object.

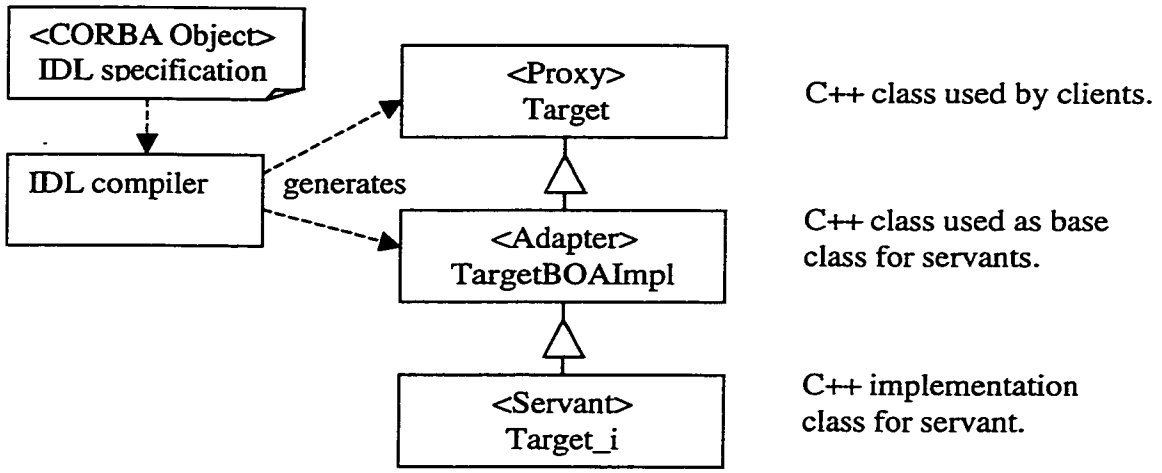

Figure 3-4 BOAImpl Approach, the inheritance form of the adapter pattern.

The second form of the adapter pattern uses delegation to pass invocations from the interface to the implementation object. In Orbix, this is achieved by the use of an adapter called a "TIE" object. The TIE class is generated by a macro from the servant class. The TIE object takes the servant implementation object as part of its constructor. It then delegates all incoming requests for the CORBA object to the servant. The TIE approach to implementing server objects is shown in Figure 3-5.

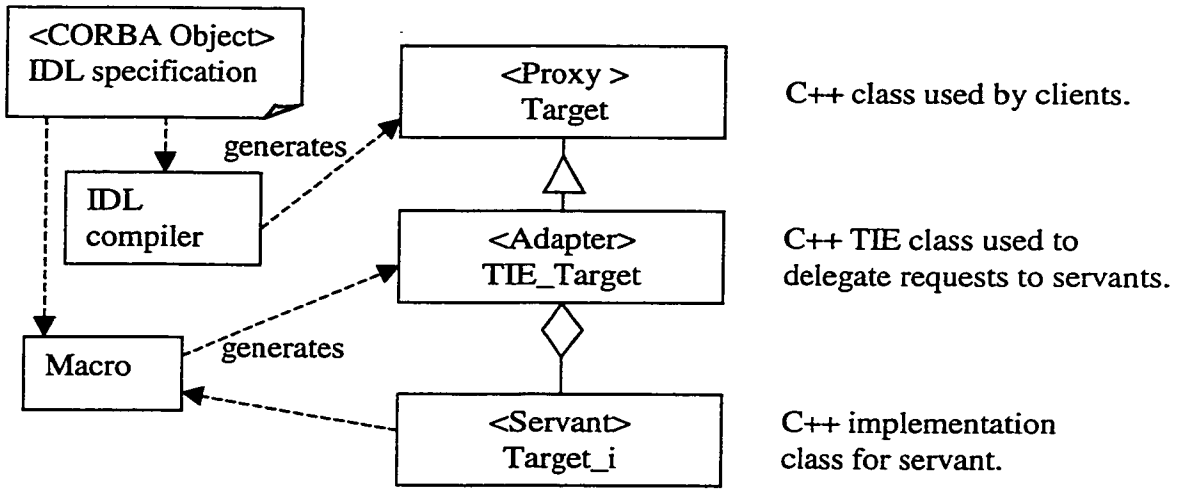

Figure 3-5 TIE Approach, the delegation form of the adapter pattern.

The advantage of the TIE approach over the BOAImpl approach is that the servant class does not have to inherit from any BOAImpl classes. This allows the reuse of existing classes or the use of classes that must inherit from a hierarchy that is incompatible with the BOA hierarchy, such as the use of a database adapter to an OODBMS. The drawback of the TIE approach is that servant incarnation is more complicated because a TIE object, which comprises the servant object, must also be created. Both the TIE and BOAImpl approaches can be used in the same implementation but applications are usually developed using one approach exclusively.

3.3.3 Server-side Dynamic Skeleton Interface

The Dynamic Skeleton Interface (DSI) can be used instead of the static skeleton interface for special applications [7]. It allows a server to receive an operation or attribute invocation on any object, even one with an IDL interface unknown at compile time. Instead of linking the skeleton code with a servant object, the server can provide a function that is called when an incoming invocation occurs. This function can then determine the identity of the object being invoked and the operation name and arguments in the request. The server can then dynamically decide how to process the request, such as translating the request into another interface or communication protocol. In fact, the DSI was explicitly designed to help programmers to write gateways between CORBA and non-CORBA environments such as SNMP or CIMP [7, 11]. The implementation of the DSI can be very complex, even for a simple server.

3.3.4 Client-side Static Invocation Interface

Client applications using CORBA can be written in any programming language for which a compatible ORB is available. The design and implementation of a client application is related to the server application only through the use of a common IDL specification. The use of stub code generated from an IDL specification is known as the Static Invocation Interface (SII) for client applications. The SII provides the client application with a set of

CORBA classes that it can process. Within the client application, these CORBA objects are accessed in the same manner as native objects. The client typically uses the IDL generated stub code to instantiate a local proxy for a CORBA object. The proxy exists within the client address space and is referenced by the client application as a native object. The proxy acts as a representative for a remote CORBA object that is incarnated by a servant on a remote server. The proxy maintains a CORBA object reference to the target object on the remote server. Only one proxy object exists in the client address space for a single remote target object, regardless of the number of local references to the target object. An example of a client proxy is shown in Figure 3-6.

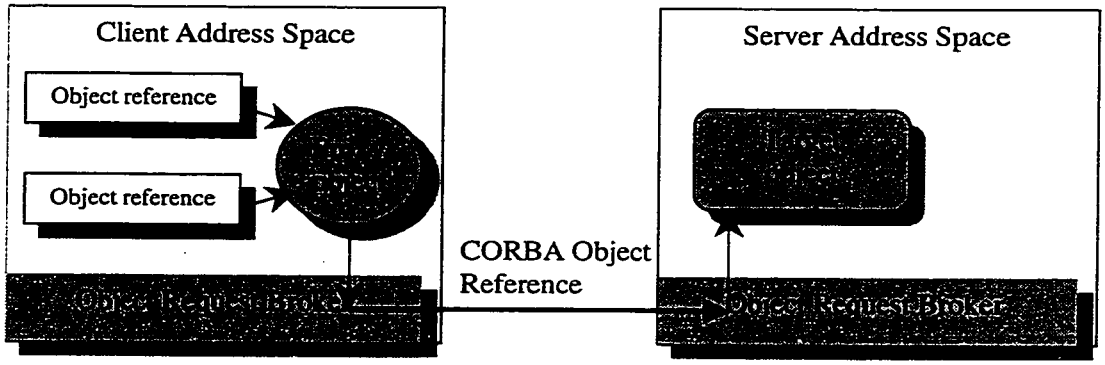

Figure 3-6 Client proxy for a target object on a remote server

The client-side ORB is responsible for processing all invocations on the client object by implicitly creating an IIOP request message, marshaling the request, transmitting the request over the network to the remote server, receiving and unmarshaling any return values, and passing the result back to the client application. The transmission of operation or attribute requests on a CORBA object over a network is transparent to the client.

3.3.5 Client-side Dynamic Invocation Interface

The use of predefined CORBA object classes may be limiting to some client applications. To address this, the OMG has defined the Dynamic Invocation Interface (DII) for client-side CORBA applications. The DII enables client applications to send operation and

attribute requests to a remote CORBA object without predefined knowledge of the object's interface. Such calls are termed "dynamic" because the IDL interfaces used by a program do not have to be "statically" determined at the time when the program is designed and implemented. Instead, the client application creates a Request object, provided by the ORB interface, and populates the Request with the target object reference, the name of the operation or attribute for the target object, and any parameters to be passed. The interface description of a previously unknown CORBA object can be obtained from the Server's host Interface Repository, which maintains the IDL specifications for all server objects.

Invocation of the DII request causes the ORB to explicitly create an IIOP request message and send it to the target object on the remote server. The composition of the IIOP request message is the same regardless of the client using the SII or DII to generate the request. This makes the use of SII or DII in the design and implementation of the client application independent of the server application.

3.4 The Interface Definition Language (IDL)

The Interface Definition Language (IDL) is used to describe the interfaces of objects in CORBA. The language is part of the OMG specification [19] for CORBA and is also an International Standards Organization (ISO) standard [45]. IDL allows interfaces to be defined independently of the languages used to implement and use these interfaces. IDL is not a programming language and can not be used to implement the functionality of an interface. The IDL specification must be mapped to a programming language such as C++, Smalltalk, Java, COBOL, or Ada, to provide an implementation for an interface. This mapping is usually provided as part of the CORBA development environment for a specific language. This section presents the basic features of IDL required to support the concepts proposed in this thesis. A detailed description of the IDL standard can be found in [6, 7, 19, 44].

The OMG IDL is CORBA's fundamental abstraction mechanism for separating object interfaces from their implementations. Because IDL describes interfaces but not implementations, it is a purely declarative language. There is no way to write executable statements in IDL, and there is no way to specify the object state (execution and state are implementation concerns). The core concepts supported by IDL are interfaces, operations, exceptions, and inheritance. These are presented in the following sections.

3.4.1 Interfaces

Every CORBA object has exactly one interface, but there can be an unlimited number of objects of the same interface in a distributed system. In this respect, IDL interfaces correspond to class *definitions* and CORBA objects correspond to class *instances*. Interface instances can be implemented in a single address space, in multiple processes on the same machine, or in multiple processes on different machines. A CORBA object is an interface instance that is remotely addressable. As such, an IDL interface defines the smallest granularity of distribution supported by CORBA. This determines the extent to which an application can be distributed over different physical address spaces; functionality can only be distributed if there is an interface defined to access that functionality.

The syntax of IDL interfaces is deliberately simple because it must support mapping to multiple implementation languages, and because it does not have to support execution. IDL interfaces are strictly declarative and each forms a namespace for the following types of constructs enclosed within the interface definition [6]:

- Constant definitions
- Type definitions
- Exception definitions
- Attribute definitions
- Operation definitions

Note that interface definitions cannot be nested within another interface. However, multiple interfaces can be defined within a shared namespace called a *module*. Modules combine related interface definitions into a logical group and prevent pollution of the global namespace. The use of modules to scope the interface definition is essential to maintaining the global namespace in large heterogeneous distributed systems. Modules can contain other modules so it is possible to create nested hierarchies using the namespaces of modules. Interfaces in separate modules can be addressed by using the module name as part of the name up to the global level; this is referred to as the *fully scoped name* of an interface.

IDL interfaces only define the interface to an object and do not imply anything about the implementation. This is characterized by a few subtleties of the interface definition for a CORBA object [7]:

- All definitions made in an IDL interface are public. There is no concept of private members in IDL. If some aspect of the object is not to be made public, then it is simply not included in the IDL definition.
- IDL interfaces do not have member variables. IDL does not support the concept of a variable because member variables, or attributes of an object, store state information. The state of an object is considered an internal of the object, which makes the state an implementation concern, not part of the interface.
- Interface attributes do not necessarily map to object member variables as they could be implemented in a number of ways: a static data value, an element of an array, a field in a file, etc. As such, an interface attribute cannot be assumed to represent the state of the CORBA object, although it could be used to access state information in the object's implementation.
- The parameter-passing mode and name of each argument in an operation must be specified. Return types must be specified for all operations.
- Interface names become type (class) names once they are declared.
- Interface instances can be passed as parameters to operations or return values between objects in separate address spaces.

- There are no constructors or destructors for an interface. Clients cannot create CORBA objects of any interface on any arbitrary server in the system. Object creation is a responsibility of the server. The use of the factory pattern to export an interface for creating objects is common in server design.
- A CORBA object must have a unique identity for a given interface on a given server. This identifier is referred to as the Object's *marker*. The marker is normally set at object creation and may or may not be made public through the object's interface. A CORBA object is considered to exist as long as a server can incarnate its servant for an object reference indicating the object's marker, interface and server. This has special significance for the use of Loaders [7] to act as factories, as will be discussed later.

3.4.2 IDL Types

IDL provides a set of basic types that form the primitive components of an interface. These basic types, shown in Table 3-1 [6], correspond to common primitive types found in implementation languages such as C++ or Java, and are described in detail in [19]. The most significant basic type is the universal container type any, which can be used to transmit IDL types that are unknown at compile time. A value of type any can contain a value of any other IDL type, such as long, a user-defined type, object references, or another value of type any. Values of type any are type-safe because they contain a field describing the type of the value and enforce run-time type checking on the extraction of the any value.

Table 3-1 IDL Basic Types.

Type	Range	Size
short	-2^{15} to $2^{15}-1$	≥ 16 bits
long	-2^{31} to $2^{31}-1$	≥ 32 bits
unsigned short	0 to $2^{15}-1$	≥ 16 bits
unsigned long	0 to $2^{32}-1$	≥ 32 bits

float	IEEE single-precision	≥ 32 bits
double	IEEE double-precision	≥ 64 bits
char	ISO Latin-1	≥ 8 bits
string	ISO latin-1, except ACSII NUL	Variable-length
boolean	TRUE or FALSE	Unspecified
octet	0-255	≥ 8 bits
any	Run-time identifiable arbitrary type	Variable-length

IDL also supports the following user-defined types:

- Type definition statements (typedef) used to create new user-defined named types for existing types;
- Enumerated types (enum) with ordinal values increasing from left to right, but otherwise undefined in value;
- Structures (struct) containing one or more named members of arbitrary type;
- Unions (union) with a discriminator to indicate which member is active;
- Arrays (typedef Type name [ ] [ ]) of multiple dimensions with declared bounds;
- Sequences (typedef sequence<Type, bound> name) are variable-length list of any element type that can be bounded or unbounded;
- Constant and literal (const Type) declarations of integer, floating-point, and boolean constant named values.

3.4.3 Interface Attributes

Attribute definitions can occur only as part of an interface definition. An IDL attribute is defined as a symbolic name with its type declared as an IDL basic type, user-defined type, or an object as previously declared in an IDL Interface. An attribute defines a pair of operations that the client can call to send and receive a value. These attributes do not have to be implemented as member variables of the servant class that incarnates the CORBA object defined by the interface. Attributes can be viewed as operations to access values or objects through the object reference to the CORBA object in which they are

declared. Attributes can be declared as readonly, which specifies that the client can get the current value of the attribute, but not set it. Servant objects usually provide operations to get and set (if not readonly) attribute values. IDL attributes vary from IDL operations, defined below, in that they cannot raise user-defined exceptions, though they can raise system exceptions.

An attribute of an interface can be of that interface type (a self-referential type) provided that a forward declaration is made for the interface name within the module. Forward declarations are required wherever an interface is used as an attribute or operation argument type before the actual interface is defined within a module.

3.4.4 Interface Operations

Operation definitions can occur only as part of an interface definition. Each operation name within an interface declaration must be unique. IDL does not allow overloading of operations. Each operation declaration must contain:

- a return result type;
- an operation name; and
- zero or more parameter declarations, each comprised of a directional attribute, a type declaration, and an argument name.

The directional attribute determines the parameter-passing mode as one of in, out, or inout, where:

- in indicates that the parameter is sent from the client to the server.
- out indicates that the parameter is sent from the server to the client.
- inout indicates that a parameter is initialized by the client, sent to the server, possibly modified and sent back to the client.

These directional attributes are required to reduce network traffic by only passing parameters in the direction required, and to determine the responsibility for memory management in the client and server.

Operation calls from a client to server are normally synchronous in CORBA. The client process will block until the operation is executed on the server object, even if no return value is expected. For applications where this is not desired, IDL permits an operation to be declared as oneway. A oneway operation must adhere to the following rules to ensure that no return values are possible:

- It must have a return type void.
- It must not have any out or inout parameters.
- It must not have a raises expression for user exceptions.

The CORBA specification for how an ORB must implement the oneway operation does not guarantee that it will arrive at the server or that it will not block the client [6]. As blocking is an implementation issue, it may be preferable to implement non-blocking operations using the Dynamic Invocation Interface. This would reduce the complexity and uncertainty of the IDL interface. There are also other CORBA services available that can provide non-blocking behavior for clients.

An IDL operation may raise an exception to indicate that an error has occurred. Exceptions are defined as part of an interface or module and may include one or more member variables of specified types. These are treated as arguments returned to the operation caller that attempts to indicate the reason for the error. As well as user-defined exceptions, CORBA defines a set of standard exceptions that may indicate system errors, such as inter-ORB communication failure or that a CORBA object is non-existent.

3.4.5 Interface Inheritance

Another important feature of IDL is the support for Inheritance. Scoped resolution for inheritance works the same as for C++, i.e. ":" identifiers are resolved by successively searching base interfaces towards the root. The root of the inheritance tree for all IDL objects is implicitly defined as Object. As inheritance gives rise to polymorphism, a derived interface can be treated as if it were a base interface. And because all IDL interfaces directly or indirectly inherit Object, all interfaces are type-compatible with type Object. This allows IDL operations to be specified as passing Object type parameters and return values. It is then up to the client to narrow the object reference to the desired subclass type. This is a common technique in CORBA and is used by many of the CORBA services such as the Name Service that will be discussed later.

The following is an example declaration of a module containing two interfaces with one interface being a subclass of the other. The example shown in Figure 3-7 illustrates a number of the concepts of IDL interfaces described in this section.

```
module MyModule {

        interface MyBaseClass {

                exception MyError {                                    // exception
                        boolean error_occurred;
                };

                const long a = 1;                                      // constant
                typedef long NewValue;                                 // user-defined type

                attribute long value;                                  // attributes
                readonly attribute string name;

                void    add(in NewValue v) raises(MyError);            // operation
        };

        interface MySubClass : MyBaseClass {

                const long a = 2;                              // redefined constant
                typedef unsigned long NewValue;                // redfined typedef

                // Can not redefine superclass attributes or operations

                long    subtract(in NewValue v) raises(MyError);
                void    setName(in string n);
        };

        interface MyOtherClass {

                void    changeObject (inout MyModule::MyBaseClass msc);
                object  getObject(in string name);
        };
};
```

Figure 3-7 Example of an IDL Interface definition.

The CORBA IDL also supports multiple-inheritance as shown in Figure 3-8 [6]. This is useful for interface aggregation. The usual type compatibility rules apply when passing CORBA objects as a base interface instead of its derived interface. The limitation on multiple inheritance is that operations and attributes must not be inherited more than once from separate base interfaces to avoid conflicting type definitions. Multiple-inheritance is a convenient technique for adding an interface with a new service to an interface that is part of a separate hierarchy. Because IDL only provides interface inheritance, it does not

imply that the use of multiple inheritance in the interface requires multiple inheritance in the under-lying implementation.

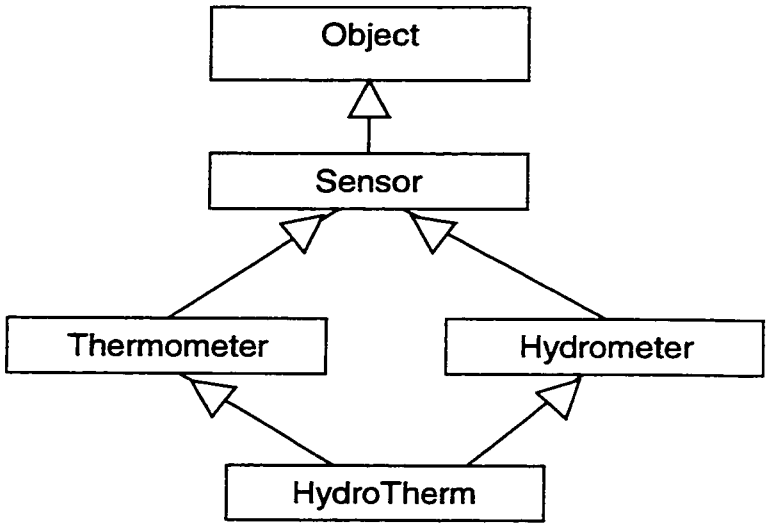

Figure 3-8 Multiple inheritance of the same base interface

The last important aspect of IDL that is relevant to this thesis is the distinction between the interface inheritance hierarchy and the implementation hierarchy. IDL inheritance only applies to interfaces, the implementation options are completely unconstrained. The structure of the IDL hierarchy need not be reflected in the implementation. The inheritance hierarchy of the implementation classes can be completely independent of the interface hierarchy. This separation between interface and implementation is one of the key aspects of CORBA that is exploited in this thesis.

3.5 CORBA Object References

The behavior of object references in CORBA is important to this thesis. As this thesis proposes a concept for dynamically versioning a CORBA object from one class definition to another, the problems of maintaining consistent and valid object pointers between clients and target objects must be addressed. Important aspects of CORBA object

references that impact the versioning process are presented here but the significance will be discussed later.

There are three types of object references involved in a typical CORBA operation. These are the local reference to the proxy object in the client, the local reference to the servant object in the server, and the CORBA object reference that is passed between the client and server. The first two are standard object references implemented by the programming language and point to real objects in memory. The CORBA object reference is a virtual reference, as it does not point to a location in memory on a particular machine. Instead, a CORBA object reference is realized as a data structure that is used by the ORB to locate a CORBA object on a remote server using IIOP. The client and server applications do not use the CORBA object reference explicitly as it is represented as a local object reference in the programming language of the client or server. From the client's perspective, a CORBA object reference is comprised of the local reference to a proxy object, the reference used by the IIOP message to the remote server, and the local reference on the server to the servant.

Clients can obtain CORBA object references in a number of ways. References are typically obtained as parameters or returned values from operations. Factory objects on servers typically create a CORBA object and return a reference to it to the client. A CORBA object reference can also be made available through the Naming Service which maps symbolic names to existing CORBA object references. The client uses these references in the same way as any other local object reference.

Clients can also convert CORBA object references to and from strings using the IDL operations object_to_string() and string_to_object(), respectively. The 'stringified' object reference produced and processed by these operations is called the Interoperable Object Reference (IOR). The IOR is a variable length string of hexadecimal digits that encode all aspects of the object reference required to uniquely identify the object. The format of the IOR is standardized by CORBA and can be decoded by any ORB. The CORBA object model, which specifies that object references must be opaque, requires that the

internals of IOR strings are not to be modified by either the server or client applications. They can be stored or transmitted outside of CORBA and will remain valid as long as the object to which they refer still exists. Two stringified object references for the same object may be different because the ORB can encode additional information into the IOR string. For instance it might encode the port number of the server and still uniquely identify the object. However, this implies that a stringified object reference cannot be used by a client application to compare CORBA objects.

Some ORBs, such as Orbix® by IONA, provide an overloaded `string_to_object()` operation that takes strings as arguments for the marker, interface name, server name, host name, and Interface repository name of the target object [41]. This overloaded `string_to_object()` operation implicitly tries to bind to the object described on the specified server and, if it can successfully locate the CORBA object, returns a valid object reference.

Another important aspect of CORBA object references is the reference count maintained by clients and servers. Both the proxy and servant objects maintain a reference count that indicates the number of *local* references that refer to it. The reference counts support local memory management only: management of space for objects in servers, and management of space for proxies in clients. This allows each application to determine when a local object (servant or proxy) is no longer referenced and its memory can be freed.

There is no means to determine how many clients hold CORBA object references to a target object on a remote server. If a server deletes the servant for a CORBA object, there could still be clients holding CORBA object references for the target object. Any subsequent operation calls on the target object using the CORBA object reference will require the server to reincarnate the servant. Only if the virtual CORBA object is destroyed by the server, signifying that it shall no longer exist, should an exception be returned to a client.

3.6 Other Components of the ORB

This section presents two components that provide the run-time functions of an ORB. These are the Implementation Repository and the Interface Repository. The functions that these components provide to the ORB are essential to the way in which an ORB supports requests from a client to a server. Understanding of the behavior and structure of these two components is important to understanding the problem of dynamically versioning objects in CORBA and the constraints imposed on the DVS.

3.6.1 The Implementation Repository

The Implementation Repository is the component of an ORB that is typically responsible for maintaining registration information about servers and their activation mode. It is also involved in establishing the initial connection between clients and servers. The CORBA 2.0 specification does not standardize the implementation repository because they involve many operating system specific operations, such as process creation and termination, and they impact the vendor specific features and performance of an ORB. Only the features common to the Implementation Repository of most general-purpose ORBs, including the Orbix 2.3 ORB used to demonstrate the concepts proposed, will be discussed.

An Implementation Repository has the following responsibilities.

- It maintains a registry of known servers.
- It records which server is currently running on which host and at which port number.
- It starts servers on demand if they are registered for automatic start-up.

The Implementation Repository also maintains a mapping from a server's logical name to the file name of the executable code that implements that server. This registry may also include other activation information such as the host and specific port number for the server to use, security information, and the activation mode for launching the server. This

mapping is usually set when the server's executable file is registered with the Implementation Repository.

An Implementation Repository can support servers that are launched on different hosts. Each server maintains configuration information indicating the host and port number on which its Implementation Repository resides. The set of servers and hosts supported by the same interface repository is known as the *location domain.*

The importance of the Implementation Repository is in its role of establishing a connection between a client and a server. When a server creates a CORBA object, it also creates the Interoperable Object Reference (IOR) for the object. The IOR is a complex addressing structure that includes the object marker, its IDL interface name, its server's logical name, and the host and port number of the server's Implementation Repository. All servers in the same location domain will include the same host and port number for the Implementation Repository in the IOR. Because a CORBA object reference can be passed to other clients or servers or publicly registered in the Naming Service, there must be a mechanism for initially resolving a client object reference to a target object on a specific server. The Implementation Repository provides this mechanism.

When a client first uses an object reference that it obtained from a source other than the current instantiation of the target object server, it must locate the server providing the CORBA object. If the object reference was returned directly from an active server, the client will use direct binding to the server and target object. Otherwise, the client will initially bind to the Implementation Repository host and port number in the IOR. Because the Implementation Repository and target server share the same ORB, the Implementation Repository decodes the client request and attempts to do the following:

- If a server with the specified logical name (as encoded in the IOR) is known and is currently running, the Implementation Repository creates a new IOR using the server's host and port number and passes it back to the client. The client ORB then uses the new IOR to reference the CORBA object and sends the request to

the target server. This change in object references is performed by the ORB and is outside of the application programmer's control.

- If a server with the specified logical name is not currently running but is registered for automatic activation in the Implementation Repository, the Implementation Repository will start the server process and then return a new IOR with the server's host and port number to the client.
- If a server with the specified logical name is not registered in the Implementation Repository, an OBJECT_NOT_EXIST system exception is returned to the client.

It is possible for the Implementation Repository to receive an IOR for a server that is not registered with it. The server may have been removed from the Implementation Repository or a process other than the target server may have created the object reference. The latter is possible because many ORBs provide operations that allow a client to initially create an IOR [7, 41].

For example, Orbix generates a static member function `_bind()` for each interface. The Orbix `_bind()` function is used to find a particular object (by specifying its marker, server's logical name, and host) and then, if successful, to create the proxy for it in the client's address space. The `_bind()` function creates an IOR and a request that is processed by the Implementation Repository to provide the actual object reference for the desired object, if it exists on the server. In this case, the client ORB must be configured with the location of the Implementation Repository; that is, it must be in the same *location domain.*

Another way that a client could obtain an object reference is with the overloaded ORB operation `string_to_object()`. This operation takes string arguments for the marker, IDL interface, server (logical) name, server host name, Implementation Repository host name, and interface repository name. This operation requests the CORBA object's current IOR using the specified Implementation Repository but does not require that the client application be in the same location domain as the server. It does require that the server use an ORB with a standard port number for its Implementation Repository. The

use of the `string_to_object()` operation to obtain a CORBA object reference is shown in Figure 3-9.

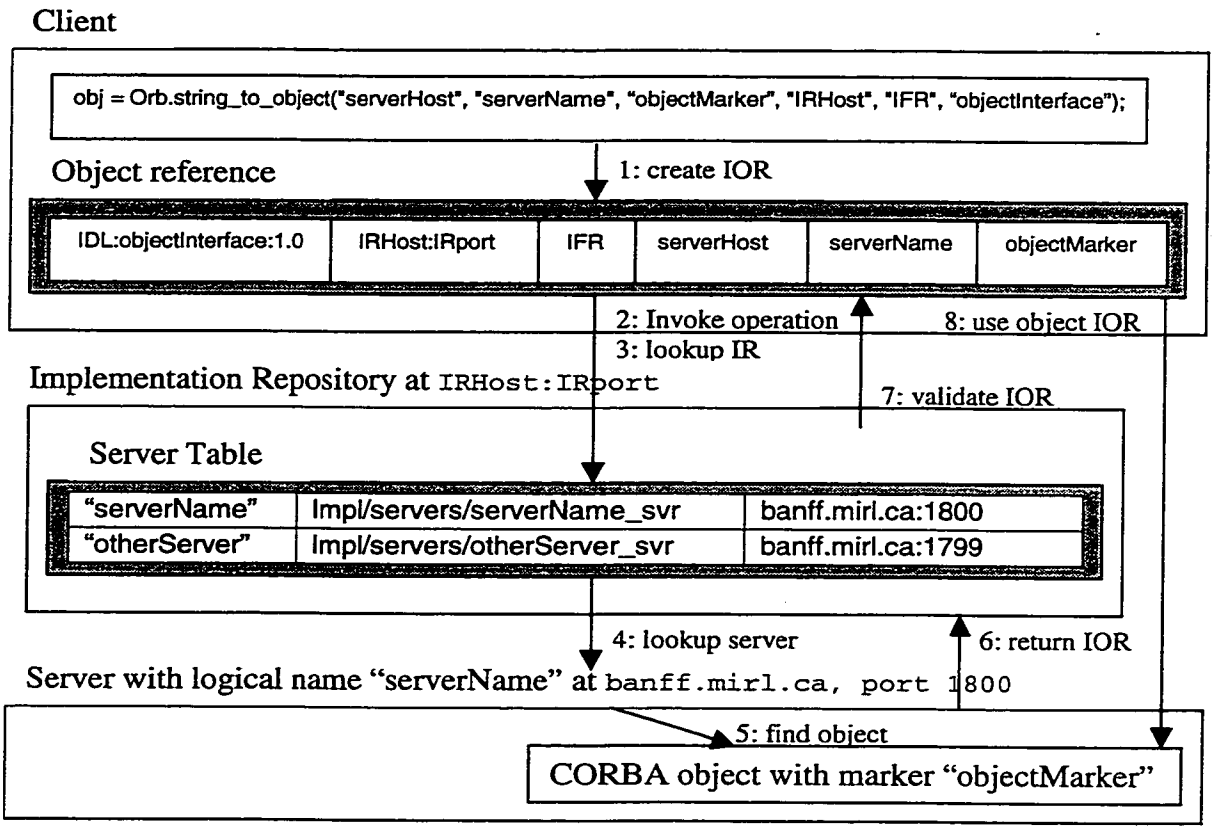

Figure 3-9 Binding of an object reference using `string_to_object()`

Both the use of the `_bind()` or `string_to_object()` operations will cause the Implementation Repository to launch the server, if not already active and registered for automatic activation, and request a reference to an object with the specified marker and IDL interface. It is the responsibility of the server to incarnate a servant for the CORBA object and return either a valid object reference or an OBJECT_NOT_EXIST system exception to the client. The CORBA object model dictates that these operations should only be successful if the desired CORBA object already exists in either its ACTIVATED or DEACTIVATED state. Theses operations are not intended to be used by clients for explicit CORBA object creation as that is a function of the server factory [6]. It is argued that versioning of an existing CORBA object is not creation, and as such, these

operations could be used in the creation of a new version of an existing CORBA object on a server.

The Implementation Repository can also launch a server if it is not active and is registered for automatic registration. Servers can also be launched when the ORB is started or manually started. The server can be set to run indefinitely, until it is manually shutdown, it exits due to an error, or it can be set to terminate on a timeout after a period of inactivity. In any case, a server can be launched using different activation modes that determine how the server process is implemented by the underlying operating system.

The following primary activation modes are supported [7]:

- Shared activation mode. This is the default activation mode and is used by most applications including the framework presented in this thesis. In this mode, all objects with the same server are managed by the same process on a given host.
- Unshared activation mode. In this mode, individual objects of a server are registered with the Implementation Repository and a single process handles all invocations for an individual object. The server process is activated by the first invocation of that object and one process is created per active registered object. The motivation for this mode is to support objects that cannot or should not run in the same process.
- Per-method-call activation mode. In this mode, individual operation or attribute names are registered with the Implementation Repository. Each operation invocation results in the creation of an individual process, which is terminated when the operation is complete.

The shared and unshared activation modes also support different sub-modes:

- Multiple-client. One server process is launched to serve requests from all client processes from an unlimited number of users. This is the default sub-mode used by most applications, including the framework presented in this thesis.

- Per-client. This sub-mode launches one process for a given server for all client processes from the same user. It launches a separate server process for different users. This requires a security facility to verify each client's user name.
- Per-client-process. This sub-mode launches a separate server process for different users. It also launches a separate process for a given server for each client process from the same user.

These different modes are used to support load balancing or security features of the application. Each mode has a different impact on processing requirements on the server host. The unshared activation modes also complicate the relationship between an object and its persistent storage, making the problem of maintaining object consistency more difficult. Although the Dynamic Versioning Service presented in this thesis could be implemented with different activation modes the shared activation mode with multiple-clients shall be used for simplicity.

3.6.2 The Interface Repository

The Interface Repository (IFR) is the component of CORBA that provides persistent storage of IDL specifications for modules, interfaces, and other IDL types. An IFR can be iterated through to browse its contents. Alternatively, given an object reference, information about the object's type (interface) can be determined at run-time by calling functions defined by the IFR. This facility is useful for many applications such as:

- Browsers that allow code designers to determine the types of objects that have been defined in the system and to list the details of these types.
- Client applications that use the Dynamic Invocation Interface to invoke operations on objects whose types were not known at compile time. Such client applications could dynamically determine the interface of an object and construct operation or attribute requests using the DII. An example of this

would be a client that uses a metamorphic object as proposed in this thesis. The client could use the IFR to determine the operations and attributes provided by a new version of an object and use the DII to invoke these operations.

- Integrated Development Environments (IDE) that provide a code management facility, software design and development tools, and debugging.
- A gateway that requires runtime type information about an object being invoked.

The contents of the IFR can be viewed as a set of CORBA objects where, for each IDL type definition, one object is stored in the repository [7]. In this way the IFR acts as a meta-object repository, where each object contained in it describes the structure and operations of CORBA objects present in the system. A metaclass is the class of a class, and each metaclass has one instance [8] used to describe the class. In languages such as Smalltalk, classes are themselves objects, which are an instance of their metaclass. In CORBA, the entry in the IFR can be viewed as the instance of the metaclass. Using IDL interfaces to the meta-objects in the IFR, an application can determine the full IDL definition for a given object at run-time.

The IFR is an important facility for the DVS. As the IFR can be viewed as a repository for meta-objects that contain the definitions of CORBA objects, the versioning of objects, that is the changing of the object's definition, is closely linked with the IFR. Maintaining the consistency of interface definitions among all Interface Repositories in a network is an important part of the process for dynamically versioning CORBA objects. Only the basic aspects of the IFR are described here but a detailed description of the IFR can be found in [6, 7, 19].

The objects in the IFR support one of the following IDL interfaces, depending on the IDL construct they describe [7]:

- Repository: the type of the repository itself, in which all of its other objects are nested.

- ModuleDef: the definition of a module. Each module has a name and may contain definitions of any type (except repository).
- InterfaceDef: the definition of an interface for a CORBA object. Each interface has a name and a possible inheritance declaration, and can contain definitions of type attribute, operation, exception, typedef, and constant.
- AttributeDef: each attribute has a name, a type, and a mode that determines whether or not it is read-only.
- OperationDef: each operation has a name, a return value, a set of parameters, and optionally, raises clauses for exceptions and contexts.
- ConstantDef: each constant has a name, a type, and a value.
- ExceptionDef: each exception has a name and a set of member definitions.
- StructDef: each struct has a name and holds the definition of each of its members.
- UnionDef: each union has a name and also holds a discriminant type and the definition of each of its members.
- EnumDef: each enum has a name and a list of member identifiers.
- AliasDef: each alias (typedef) has a name and type to which it maps.
- PrimitiveDef: objects of this type correspond to a primitive type such as short or long, and are predefined within the IFR.
- StringDef: objects of this type correspond to bounded strings. They do not have a name. If they have been defined using an IDL typedef statement, they will have an associated AliasDef object.
- SequenceDef: each sequence type records its bound and element type. Unbounded sequences have a bound of zero. All elements in a sequence must be of the same type. Objects of this type do not have a name. If they have been defined using an IDL typedef statement, they will have an associated AliasDef object.
- ArrayDef: each array type records its length and its element type. Each ArrayDef object represents one dimension; multiple ArrayDef objects are required to represent a multidimensional array type. The elements of an array must be of the same type. ArrayDef objects do not have a name. If they have been defined using an IDL typedef statement, they will have an associated AliasDef object.

In addition to these concrete types, the following abstract types (those without direct instances) are defined in the IFR: IRObject, IDLType, TypedefDef, Contained, and Container. The IFR types listed above all inherit from one or more of these abstract types. The details of the IFR hierarchy do not directly impact the Dynamic Versioning Service and are discussed only where necessary. A detailed description of the IFR abstract types can be found in [7, 19]. The structure of the Interface Repository and the interfaces to the IFR of importance are discussed below.

The concrete IFR types are nested within the IFR in a natural manner. The containment rules for these definitions correspond to their IDL definitions. For example, each InterfaceDef may contain objects representing components of a CORBA object's interface definition including constant, exception, attribute, and operation definitions. The containment rules for the IFR are shown in Figure 3-10 [7].

There are three ways to retrieve information about an object of any of the IFR types from the Interface Repository.

- Given a CORBA object reference, its corresponding InterfaceDef object can be found in the IFR. All CORBA objects inherit the operation CORBA::Object::_get_interface(), which can be invoked to obtain an object reference to its InterfaceDef object in the IFR. From this, all details of the object's interface definition can be found including its interface name, its inheritance hierarchy, and definitions for its operations and attributes.
- An object reference to an IFR Repository can be obtained and its full contents can then be navigated. Because the repository is itself a CORBA object, it exports an interface that can be invoked using an object reference. Most ORBs provide one or more methods for obtaining an object reference to the IFR.
- All IFR objects have a RepositoryId that is unique for a given object definition, regardless of which IFR contains it. For example, an InterfaceDef object for a specific CORBA object interface will have the same RepositoryId even if it is

contained in several repositories. Given this RepositoryId, a reference to the corresponding object in the IFR can be obtained and interrogated.

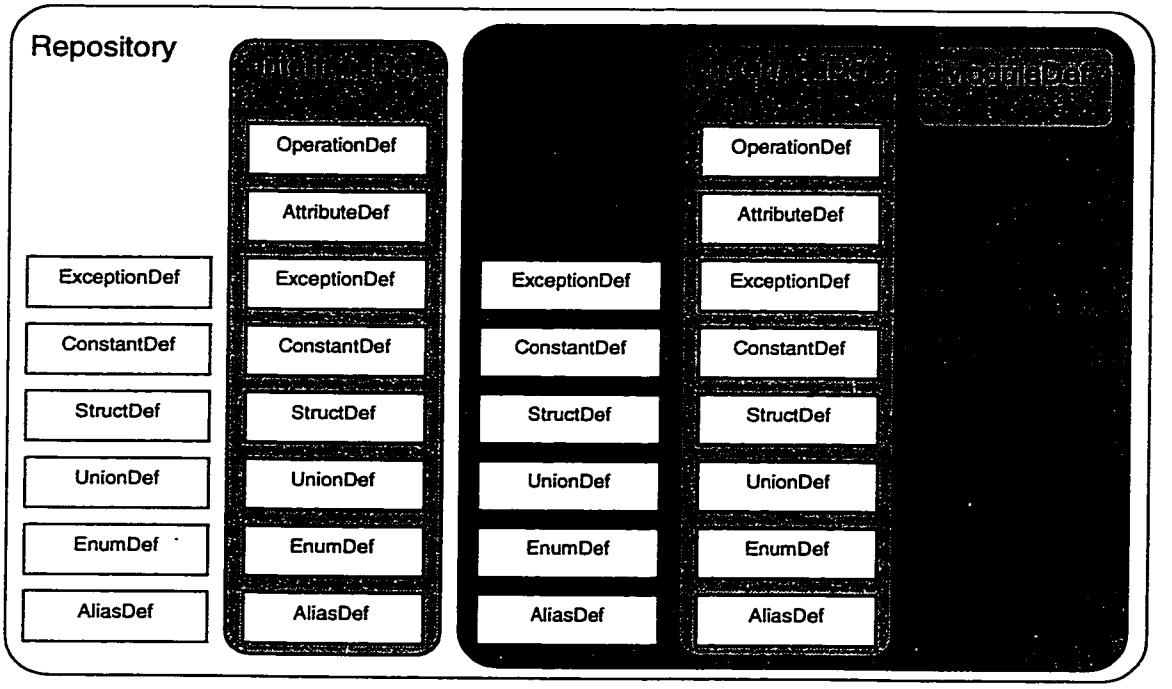

Figure 3-10 Containment rules for IFR definition objects.

The IFR provides IDL interfaces for each object type in the IFR. Understanding of these interfaces is important to many of the concepts in the Dynamic Versioning Service. However, as there are a large number of IFR types, they are not presented here. A comprehensive description of the IFR interfaces is available in [7, 41]. The IDL type definitions that are relevant to this discussion are shown in Figure 3-11.

```
// IDL. In module CORBA.
typedef string Identifier;
typedef string ScopedName;
typedef string RepositoryID;
typedef string VersionSpec;

enum DefinitionKind {
        dk_none, dk_Constant, dk_Exception, dk_Interface, dk_Module,
        dk_Operation, dk_Typedef, dk_Alias, dk_Struct, dk_Union, dk_Enum,
        dk_Primitive, dk_String, dk_Sequence, dk_Array, dk_Repository
};
```

Figure 3-11 IDL definitions for simple IFR types.

The main IFR types such as InterfaceDef and OperationDef use these simple types. The simple CORBA IFR types are follows:

- Identifier is a simple name that identifies modules, interfaces, constants, typedefs, exceptions, attributes, and operations.
- A ScopedName is an entity's name relative to scope based on the same containment rules used in the IFR. For example, the name of an interface is only a valid ScopedName relative to the module in which is contained. The scope of the name can be specified by using the name of the containing object first and concatenating subsequent names with the scope resolution operator "::" [9, 41]. For example, the ScopedName of an interface contained in a module can be specified by "ModuleName::InterfaceName". An absolute scoped name can be specified by beginning with "::", which would uniquely identify an entity in the repository.
- A RepositoryId is a string that uniquely identifies an object of any IFR type within a repository, or globally within a set of repositories if more than one IFR is being used.
- VersionSpec is used to indicate the version number of an IFR object; that is, to allow the IFR to distinguish two or more versions of a definition, each with the same name but with details that evolve over time. It appears that the OMG had considered versioning of interface definitions when it developed the CORBA standard. This version number even appears inside the object's IOR with a default

value of 1.0 (IDL:objectInterface:1.0). However, the IFR is not *required* to support such versioning; it is not required to store more than one definition with any given name. As this complicates the ORB, most vendors do not provide support for versioning. This would have been a useful facility for dynamic versioning of CORBA objects. Instead, a naming convention for IDL definitions must be adopted that indicates the version of the interface.

- Each IFR object has an attribute (called def_kind) of type DefinitionKind that records the kind of IFR object. These constants can be used when searching for objects in a repository.

3.7 CORBAservices

The CORBAservices are sets of optional utilities defined in IDL that form part of the Object Management Architecture specified by the OMG. They extend the core CORBA specification by providing commonly used services for CORBA objects and distributed applications. ORB vendors may provide an implementation for any of these services or a developer can use them as a framework in designing an application. Each CORBA service must adhere to the IDL specification provided by the OMG to support inter-operability between implementations. As such, CORBAservices are an example of the re-usability and inter-operability of open applications that can be provided by CORBA.

The following is a short description of some of the CORBAservices available in the CORBA 2.0 specification [6, 7, 41]. A detailed description is then presented for the CORBAservices pertinent to the Dynamic Versioning Service.

- **Naming Service**: allows a client in a distributed system with multiple objects residing on multiple servers or hosts to find an object given its name published in the Naming Service. A server can register any of its objects with the Naming Service, giving each a hierarchical name that is independent of the server or host on which it resides. The client resolves this name in the Naming Service to

receive a CORBA object reference to the target object, without prior knowledge of where the object resides.

- **Event Service**: allows a client or server to asynchronously send an event message to any number of receivers through an 'Event Channel'. These messages can be temporarily stored in the Event Service until delivered, decoupling the connection between the sender and receiver.
- **Security Service**: imposes a security policy that restricts access to individual objects or groups of objects. The security policy can be configured to allow only client applications or users with appropriate privileges to invoke operations on objects. Network communications can also be encrypted.
- **Trading Service**: allows a client to find an object that is registered with the Trading Service based on a set of constraints. These constraints may include the interface (or type) of object, the values of one or more of its attributes, or a relationship with other objects.
- **Object Transaction Service (OTS)**: provides a two-phase commit protocol for transactions that span multiple operations on an object. It controls the commitment and abortion of these transactions to ensure the integrity of the object's state.
- **Relationship Service**: allows relationships to be constructed and managed between objects without relying on direct object references that may change with the state and location of the CORBA object.
- **Query Service**: allows queries to be executed against collections of objects that are stored in a database. These queries use the Sequential Query Language (SQL) for relational databases and the Object Query Language (OQL) for object databases.
- **Persistent Object Service**: defines an abstract framework for designing how a CORBA object can be made persistent, allowing it to maintain its state across multiple activations. It defines how a database and an object should communicate to store and restore the object to and from the database.
- **Externalization Service**: allows an object's data to be converted to and from a stream of bytes so that is can be copied to another location.

- **Life Cycle Service**: defines standard interfaces that allow objects to be created, moved, and copied between servers using the Factory pattern [8]. These operations are often implemented using application-specific interfaces to support a more logical functionality for specific object types.

The CORBAservices of relevance to this thesis are the Naming Service, Event Service, Object Transaction Service, Persistent Object Service, and the Life Cycle Service. Only the Naming Service and the Persistent Object Service are discussed in detail here as they have a significant role in Dynamic Versioning Service. The effects of the other CORBAservices can, for the most part, be modeled as references to or from other CORBA objects that must remain consistent.

3.7.1 The CORBA Naming Service

The CORBA Naming Service defined by the OMG provides a mapping from a name to an object reference. The Naming Service provides important capabilities to clients and servers in a distributed environment [6]:

- The Naming Service provides a means for clients to obtain initial references to server applications. The Naming Service provides a directory-like structure of object names that can be traversed to locate the desired application name and obtain a reference to it.
- Servers can assign meaningful names to objects and group related objects in a named context. Client applications can then easily locate an object in the Naming Service and obtain a reference to it.
- By changing the value of an object reference advertised under a name, a server can get a client to use a different implementation of an interface without changing the client application. The client uses the same name to locate an object but is given an object reference to a different object. Of course, the Naming Service does not change existing object references already held by clients; it only provides an object reference when explicitly requested by a client.

The name-to-reference association is called a *name binding*. The same object reference may be bound to several names but each name may identify exactly one object reference. A *naming context* is an object within the Naming Service that implements a table that maps names to object references. The object references held by a name binding can refer to any CORBA object on any server or to a naming context object within the Naming Service. This allows naming contexts and bindings to be connected into a hierarchy, known as a naming graph, as shown in Figure 3-12.

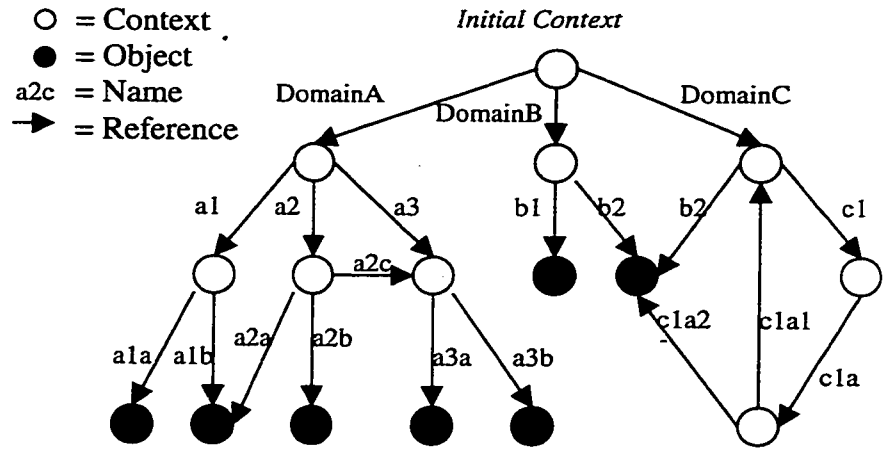

Figure 3-12 A naming graph in the Naming Service

The following rules govern the behavior of a naming graph:

- A context can appear as an interior node or as a leaf node.
- An application object always appears as a leaf node.
- Directed arcs represent object references and are labeled with their name in their context.
- Name bindings are unique within a given context; each name can appear only once within the same context.
- The same name binding can appear multiple times provided that each binding is in a different parent context.
- A single object or context can have multiple names.

- Given a starting context, a target node can be reached by traversing a path along the graph from the starting context to the target node. The sequence of bindings used in the traversal forms a pathname that uniquely identifies the target object.
- There can be more than one path to the same object or context.
- A naming graph has one or more distinguished contexts known as *initial naming* contexts that provide clients with a public point of entrance to the naming graph for a given Naming Service. This is typically the root of the naming graph.
- A naming graph can consist of several disconnected subgraphs, each with its root configured as an initial naming context.
- It is possible for the graph to have loops.

The semantics of the names that make up a name binding are more complicated than file and directory names used in a file system. The OMG Naming Service specifies that a Name is comprised of one or more structures defined as a NameComponent, as shown in the IDL definition in Figure 3-13.

```
// IDL.
Module CosNaming {
        typedef string Istring;
        struct NameComponent {
                Istring id;
                Istring kind;
        };
        typedef sequence<NameComponent> Name;
        // ...
};
```

Figure 3-13 IDL definition for a Name in the Naming Service

The id element of a NameComponent is the string bound in the naming graph to the respective object reference. The kind element is optional and can be used to indicate the role or class of object associated with the name. If the kind element is specified when an object is bound in the naming graph, it must be part of the name used by clients to resolve the object reference. Each NameComponent resolves a single object reference from its

parent context. If a full path name from a starting context to a target object or context is required, a full name can be specified by a sequence of NameComponents. Thus, the equivalence of names in the Naming Service is defined as [6]:

- Two name *components* are equivalent only if they have identical id and kind fields.
- Two names are equivalent only if all their components are equivalent.

The basic operations defined in IDL for associating names with object references in the Naming Service are bind(), rebind(), unbind(), resolve(). The first three operations set, change, and remove the binding between a name and an object reference. The resolve() operation is used to obtain the object reference bound to a name. The bindings between a context and an object reference support the operations bind_context(), rebind_context(), and list(). The list() operation can be used to iterate through a naming context. A Naming Service name knows if its binding is to an object or a context. A complete description of the Naming Service is available in [6, 7].

A typical sequence of operations performed on the Naming Service is:

- A server resolves the initial context for a Naming Service;
- The server locates or creates a name context to hold names for one or more objects;
- The server binds an object reference to the selected context by specifying an id and, optionally, a kind;
- The Naming Service maintains, persistently, a record of the name-reference bindings under the selected context;
- A client resolves the initial context for a Naming Service;
- The client resolves an object reference by specifying the complete name consisting of a sequence of name components, each containing an id and kind, which match the full path of the bound name; or
- The client traverses the naming graph and lists the names in each context until it finds the name component of interest and then resolves the object reference from the name;

- The client invokes operations on the object without explicit knowledge of the server on which the object is incarnated.

Maintaining consistency in the Naming Service is an important part in the versioning process. As client applications are assumed to use the Naming Service to obtain initial references to CORBA objects, it is possible to have the clients obtain references to the new version of an object by changing its binding in the Naming Service. However, the fact that an object reference can be bound under multiple names anywhere in the graph makes this approach very difficult unless each object maintains a record of names under which it is bound. If the kind were used to indicate the version of the interface the object supports, it would be possible to locate and change all objects of the same type. However, clients would no longer be able to match an existing name by its id and kind as they would no longer be equivalent. This problem is further complicated if a Federated Naming Service is employed [6], which provides a single logical Naming Service to clients but consists of multiple physical servers. A solution to maintaining consistency in the Naming Service is required to support the Dynamic Versioning Service.

3.7.2 Persistent Object Service

Many distributed applications require that the state of individual objects to be consistent between operation invocations. If a servant object is etherealized and re-incarnated as part of a CORBA object's lifecycle, described in section 3.2, its state must be made persistent. Since the Dynamic Versioning Service is intended for the general case of distributed applications, maintaining persistence of a CORBA object is a key factor in dynamically altering the interface and implementation definitions for that object. The DVS must provide a means of mapping a CORBA object's persistent state from one object definition to another during the versioning process. As such, a persistence mechanism for Metamorphic Objects is required. A number of techniques for providing a persistence mechanism were investigated, including the CORBA Persistent Object Service, but were discarded because they failed to meet some criteria of the versioning

process. This section discusses the Persistent Object Service (POS) and why it was found not suitable. This section also discusses the concept of a database adapter and its limitations in supporting the versioning process. The section that follows describes the Loader approach that was implemented for the DVS.

CORBA objects are considered to have a *dyanmic state* and a *persistent state*. The dynamic state is considered to be the state of transient variables within a servant's implementation code. These are internal variables that do not determine the state of the object. The persistent state is comprised of the variables and attributes of a CORBA object that must be maintained while the object is deactivated, that is, while no servant exists to maintain attribute values, so that its state can be later restored. For example, if, from a client's perspective, a CORBA object on a remote server has a given state at time t1. If this state is not changed, the client should observer the same state of the object at some later time t2, even if the server providing the object was shut-down and restarted.

The principle concepts of persistence in CORBA used by the Persistent Object Service are defined by [5]:

- A *persistent object* (PO) is an object, the lifetime of which can exceed the lifetime of the application that uses it.
- The *persistent state* of a PO is the n-tuple of values corresponding to the n attributes of the object considered relevant to the object's state.
- The *dependencies* of a persistent object are the set of all objects targeted by a reference from the PO.
- The *transitive closure of dependencies* of a PO is the set of all objects reachable from the PO via object references transitively over all nested dependencies; this is an analogy of the "deep copy" concept.

The CORBA Persistent Object Service (POS) is an abstract framework for designing how a CORBA object can be made persistent, allowing it to maintain its state across multiple activations [7]. It defines interfaces that can be used to encapsulate the mechanisms for making objects persistent. The implementations for these mechanisms are not specified

by the POS but can include Relational Database Management Systems (RDBMS), Object-Oriented Database Management Systems (OODBMS), and flat files. The OMG deliberately leaves the functionality core of the Persistence Service unspecified [5]. The intention of the POS is to provide a general architecture to store the state of CORBA objects without relying on any implementation technologies while also minimizing duplication of functions with other CORBA services [17, 18, 19]. The implementation of the POS as specified by the OMG has been attempted many times, with limited success [5, 6, 7]. However, a number of concepts were developed that are applicable to the Dynamic Versioning Service and are presented here.

The OMG specification for the architecture of the Persistent Object Service (POS) is shown in Figure 3-14 [5]. It requires objects that are to be made persistent to implement a standard Persistent Object (PO) interface that provides a Persistent Identifier (PID) attribute. The PID is used by a Persistent Object Manager (POM) to identify the persistent state of the object in the Persistent Data Service (PDS). The PDS provides the interface between the PO and the underlying data store. Any persistence operation on the PO is passed to the POM, which then uses the object's PID to access its persistent state using the PDS. The transfer of data between the PO and the PDS is provided by a *protocol.* This protocol could be one of the OMG specified protocols: Direct_Access (PDS_DA) protocol, ODMG-93 protocol, Dynamic Data Object (DDO) protocol, or some other application dependent protocol [7]. Because the POS is only a general architecture for persistence, many parts of the architecture are not standardized by the OMG [5].

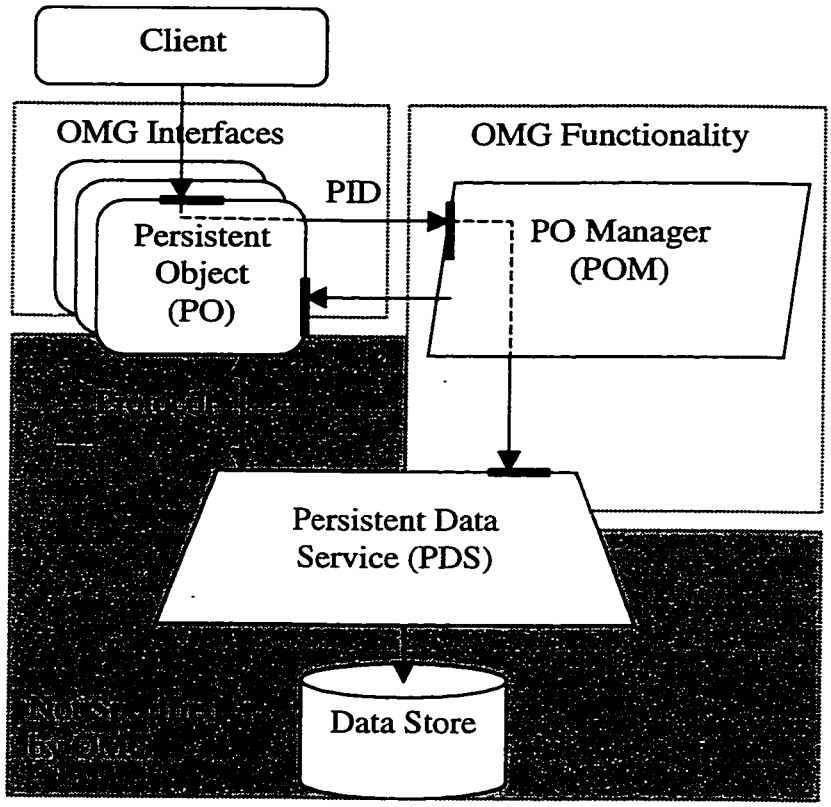

Figure 3-14 Architecture of the OMG Persistent Object Service.

The OMG specification defines only the interfaces for the PO, POM, and PDS, all of which must support five common operations: *connect()*, *disconnect()*, *store()*, *restore()*, and *delete()* [5]. These interfaces can be exported by the PO to allow clients to explicitly manage the object's persistence. Conversely, these operations can be hidden from the client and used by the server process to transparently manage the object's persistent state. The approach used affects how the object's persistent attributes are accessed. One way is to equip persistent objects with two IDL interface operations, *save_state()* and *load_state()*, that define how to store and load the object's persistent attributes compactly. This approach requires public access to the attributes that define the object's persistent state. Another approach is to directly access the attributes of the servant object that implements the CORBA object. This requires that the servant class provide a *friend* construct [8, 9] that allows access to its attributes by another server object, such as the POM. The direct access approach allows servant attributes not exposed in the IDL interface to be included as part of the object's persistent state. The decision to make all of

the object's persistent attributes public or not determines which approach to take in implementing the POS.

The aim of the POS is to use standard IDL interfaces that allow it to be used in conjunction with other CORBAservices. Implementations of the POS that successfully provide this level of reuse have been unsuccessful due to the lack of OMG standardization of the underlying components [5]. Other approaches to providing persistence have been more successful but resulted in non-standardized implementations. Because the Dynamic Versioning Service requires operations to version the definitions of objects, it cannot be constrained to the standard interfaces provided by the POS. Also, the implementation of the underlying persistence mechanism must support dynamic class definitions and transformation of an object's persistent state between these class definitions. This makes the OMG standard for the POS too constrictive for the Dynamic Versioning Service.

Another common approach to providing persistence for CORBA objects is the use of a *Database Adapter* [7]. A Database Adapter provides a virtual memory approach for incarnating CORBA servants. The servant that implements a CORBA object is also a database object that is automatically made persistent by an Object Oriented Database Management System (OODBMS). This approach uses a three-tiered architecture for providing CORBA objects to clients. In this approach, shown in Figure 3-15, the CORBA server acts as a server to clients and is itself a client to the OODBMS database server. The OODBMS is responsible for swapping servant objects in and out of the CORBA server memory space and maintaining persistent storage of all of the servant's attributes and dependencies. The implementation code for the servant class is used to generate the database schema for the class. This simplifies the development of the persistence mechanism for CORBA objects but has some limitations in regards to the Dynamic Versioning Service.

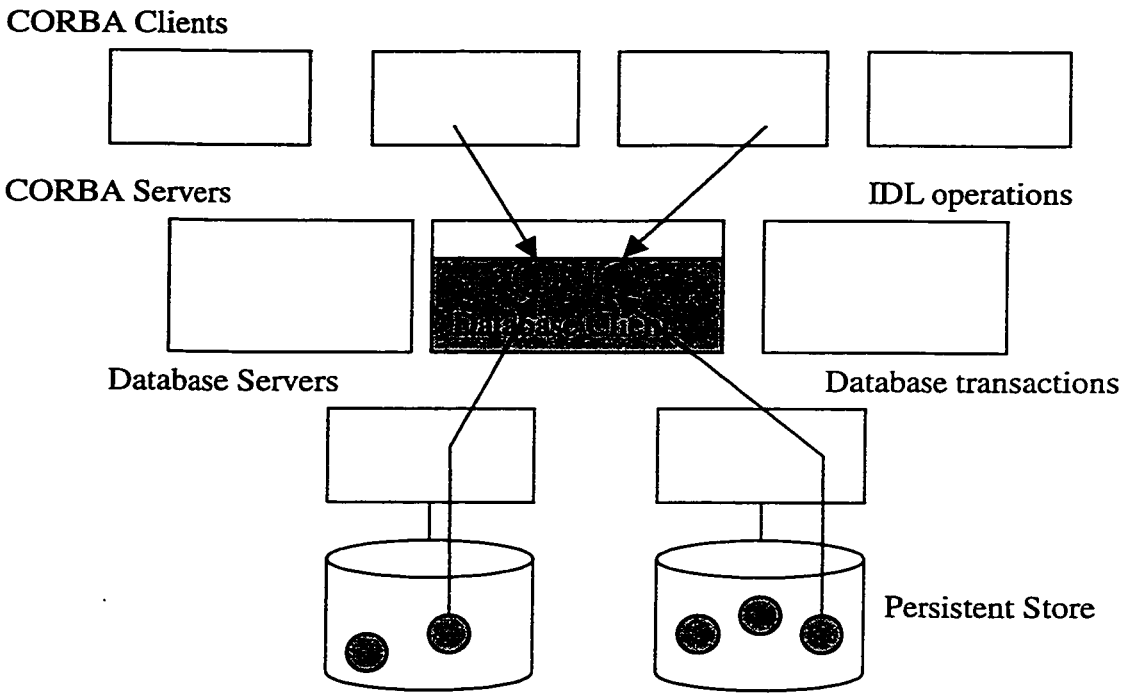

Figure 3-15 Three-tiered architecture for Database Adapters

Use of the Database Adapter in the Dynamic Versioning Service is limited by how the OODBMS manages the incarnation of the servant object in the CORBA server's memory space. Once the schema for a servant class has been set in the OODBMS, it will automatically create and maintain an instance of that class on the CORBA server whenever an operation on the specific object is invoked. The versioning process requires complete control over the type of servant object that is loaded to serve a CORBA object. Unless the server application can have control over what type of object the OODBMS loads into its client's memory for a given object reference, the Database Adapter approach cannot be used.

The OMG Persistent Object Service and the Database Adapter approaches were found inadequate for providing persistence for Metamorphic Objects. The POS did not provide suitable interfaces and implementation specifications. The Database Adapter did not provide the necessary application control over servant incarnation. However, these approaches do provide part of the solution to the persistence problem. These concepts

must be enhanced and combined with a suitable interface specification that supports versioning and a control mechanism that enables the server application to dynamically determine what type of servant object is incarnated to service a CORBA object. An approach for the latter is discussed in the next section.

3.7.3 Object Loaders

The CORBA 2.0 standard does not specify how an ORB must create servant objects. Most ORB vendors provide their own mechanism for instantiating objects to service requests by extending the basic ORB functions [7]. In Orbix 2.3, this mechanism is called a *Loader*. Orbix loaders allow objects to be loaded into a server on demand when requests arrive for them. Loaders also provide support for dynamic configuration of servants. Orbix loaders were one of the primary influences that led to the OMG specification for *Servant Managers* as part of the new Portable Object Adapter in CORBA 3.0 [4]. This section discusses how loaders can provide persistence for DMOs and dynamic configuration of servants for the Dynamic Versioning Service.

A loader is an instance of a derived class of `CORBA::LoaderClass` in Orbix [7, 41]. It is created by the server process during initialization and registered with the ORB. The loader then acts as an object fault exception handler for the ORB. When an operation invocation arrives at the server process, the ORB searches for the target object in the process's object table. An object fault occurs if a servant object does not exist to service the request. The Orbix ORB will call all registered loaders to attempt to instantiate the correct servant for the request. The loader will typically access some persistent store to determine if it can load a servant for the desired object. If successful, the servant is incarnated and the request is invoked on it. The servant then processes the request and responds to the caller. If no loader can load the target object, a `CORBA::OBJECT_NOT_EXIST` exception is raised and returned to the client. The load operation, shown in Figure 3-16, is completely transparent to the client.

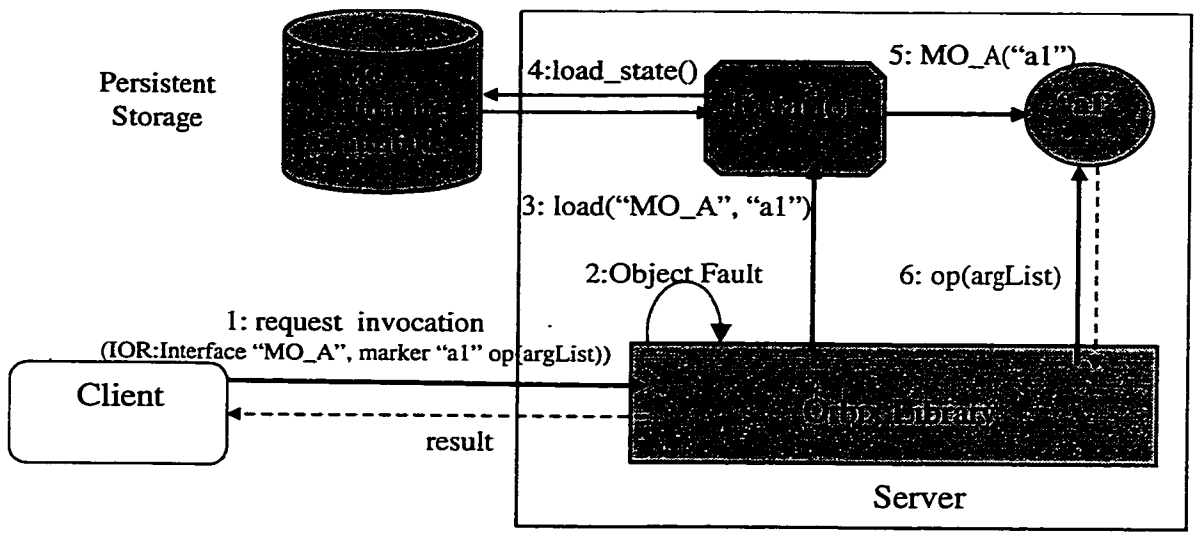

Figure 3-16 Operation of the Loader in Orbix

The loader is passed the name of the IDL Interface and marker for the target object. The marker is a string unique to each object of a given Interface on a specific server. The Interface name and marker are encoded as part of the object reference in the request message. The ORB extracts this information to locate the target object in the server process. The loader must be able to construct a servant object for the target interface and marker using state data from persistent storage. The loader may directly access the persistent store or it may use another object such as a client to an OODBMS or RDBMS, or a facility similar to the POS. In any case, the loader is only provided the Interface name and object marker to use as key to the persistent store.

The loader must be able to instantiate a servant object of the appropriate implementation class. The loader can instantiate any class of object that implements the specified interface for the ORB, with or without initializing its state, and the incoming operation will be invoked on it. This is an important aspect of the *Forwarder Pattern*, which is presented later. In this way, the loader provides a factory service for the object. It is possible for the loader to create a default object if the desired object is not found in persistent storage. It is also possible for the loader to create a CORBA object for the first time simply because an IOR reference containing an interface and marker entered the

server's address space. Such a reference could be constructed by a client using the `string_to_object()` operation. This would allow a loader to create a new object for the reference using any data it can retrieve using the interface name and marker encoded in the object reference. This is a useful construct for the versioning process that must instantiate a new object for a new interface based only on its persistent state.

The Loader provides all the basic operations to provide persistence for an object. All application loaders must inherit from the `CORBA::LoaderClass` base class. The operations provided by the `LoaderClass` are described below:

- `load()`: This function is invoked by the ORB on the loader when an object fault occurs. The loader is given the interface name and marker of the missing object so that it can identify which object to load. The ORB also calls this function when an object reference enters the server address space, possibly as an operation parameter, for an object for which no servant exists. The load operation can also be explicitly invoked on the loader by any server object that holds a reference to the loader.
- `save()`: When a server process terminates, each object can be saved by its loader. During shutdown, the ORB invokes the save operation on the loader for each individual object managed by that loader. The ORB also calls the save function when an object is destroyed. The save operation can also be explicitly invoked by calling `CORBA::Object::_save()` on any object managed by a loader. A servant object can call `_save()` to have its loader save its state to persistent storage.
- `record()` and `rename()`: These functions are used to control the assignment of the object's marker. Since the marker is used by the loader to uniquely identify an object, it is important that the loader use a logical scheme for assigning and changing an object's marker.

When a servant object is constructed, it is assigned a loader and marker. A factory object may hold a reference to the loader and assign it to servants for objects it creates. The

loader assigns a reference to itself when it creates the servant during a load operation. The servant can then reference its loader, to call `save()`, as long as it exists. When the servant is deleted, which may or may not imply the deletion of the CORBA object, `save()` is called on the loader with a flag to indicate the servant has been deleted. The loader can then decide to save, delete, or alter the persistent state of the CORBA object. If the CORBA object was not explicitly destroyed, the loader could reincarnate the servant the next time a reference to the object is received.

The power and flexibility of the Orbix Loader make it ideal for the control mechanism that must dynamically determine what type of servant object is incarnated to service a CORBA object. When coupled with a suitable persistence storage mechanism, the Orbix Loader provides the servant management functions required by the *Forwarder Pattern* to realize metamorphic objects using the Dynamic Versioning Service.

4 Metamorphic Objects

This chapter describes the concept of a Metamorphic Object to address the problems of dynamic reconfiguration of distributed applications. Forwarding is proposed as mechanism to realize Metamorphic Objects in a distributed environment. The concept of dynamically versioning Metamorphic Objects is described and the criteria for mapping between versions of a Metamorphic Object is presented.

4.1 Concept of Metamorphism for Distributed Objects

Object-Oriented programming is based on four basic concepts: **inheritance**, **polymorphism**, **encapsulation,** and **clientship**. These concepts also apply to distributed object-oriented systems. Distribution of objects requires that architectures such as CORBA provide mechanisms to support these concepts across separate memory spaces. This section describes these basic concepts and how they are combined with the distribution mechanisms in CORBA to formulate the concept of **metamorphism**.

Object-Oriented programs are made up of objects that contain both data and the procedures that operate on that data. The procedures are typically called **methods** or **operations**. Because objects are encapsulated, the only way to access an object's internal data is through an operation. An object performs an operation when it receives a **request** (or **message**) from a **client** [8]; this is the concept of clientship. Every operation is specified by its name, the objects it takes as parameters, and the operation's return value. This is known as the operation's **signature**. The set of all signatures defined by an object's operations is called its **interface**. The term **type** is often used to denote a particular interface. Objects that support the same interface are said to be of the same type, even though their implementations may be different. The term **Interface** is used here to denote a **Type** in order to maintain consistency with CORBA terminology. The use of an interface to hide an object's implementation is known as **encapsulation**.

Encapsulation is fundamental in distributed object-oriented systems because distributed objects are known only through their interface.

> Definition 4-1 Encapsulation: The result of hiding a representation and implementation in an object. Operations, as defined in the object's interface, are the only way to access and modify an object's representation [8, p360].

When a request is sent to an object, the particular operation that's performed depends on both the request and the receiving object. Different objects that support identical requests may have different implementations of the operations that fulfill these requests. The run-time association of a request to an object's operation is known as **dynamic binding**. Dynamic binding allows objects that have identical interfaces to be substituted for each other at run-time. This substitutability is known as **polymorphism**.

> Definition 4-2 Polymorphism: The ability to substitute objects of matching interface for one another at run-time [8, p361].

The degree to which different interfaces can have matching signatures may vary. The ability to use polymorphism applies to the specific operation signature being invoked. As long as the same operation has a matching signature in the two interfaces, it can be invoked on both objects, whether or not any or all other operation signatures match.

An object's implementation is defined by its **class** definition. A class is a specification for an object's internal data and representation and defines the operations the object can perform. In a statically typed implementation language such as C++, the class must specify the types of all attributes and operation parameters and return values. An object is an instance of its class. Instantiating an object involves the allocation of storage for the object's internal data and associating the operations with these data within the process in which the object exists. In a distributed environment, a mechanism must be provided to receive incoming messages and direct the operation request to the correct instance of a class in a server's memory space. This is the main function of an ORB in CORBA.

An object is an instance of a class and has a unique identity and **state**. An object's state is determined by the values of its data members at any point in time. These data members may be basic types (i.e. float, double, string) or they may be a reference to another object denoting either aggregation or association with that object. An object's response to an operation invocation is determined by its state.

> Definition 4-3 State: The state of an object is characterized by the value of one or more of the attributes of the class including the existence of links to other objects [10, p120].

New classes can be defined in terms of existing classes using **class inheritance**. When a subclass inherits from a parent class, it includes all the attributes and operations of the parent. Class inheritance is not the same as **interface inheritance**. Class inheritance defines an object's implementation in terms of another object's implementation, allowing for reuse of code. Interface inheritance, or subtyping, describes when an object can be used in place of another. An object can have many types, and objects of different classes can have the same type. In many programming environments there is no distinction between an object's class definition and its interface or type definition. In CORBA, however, there is a clear distinction between interface and implementation. A CORBA interface described in IDL can be implemented using many different programming languages, including some non-object-oriented languages. A servant object can have many IDL interfaces and servant objects of different classes can have the same IDL interface. This concept is applied in the Forwarder pattern to be discussed.

> Definition 4-4 Inheritance: A relationship that defines one entity in terms of another [8, p360]. Implementation inheritance may not be equivalent to interface, or type, inheritance.

The concept of Metamorphism is built upon these basic principles of object-oriented programming and the CORBA mechanism for sending requests to objects whose type

was unknown at creation time, the **Dynamic Invocation Interface** (DII). Through the use of polymorphism, the DII enables an operation request intended for an object of one type to be invoked on another object of a compatible type, as defined in section 4.3. The problem is how to redirect the operation invocation to the instance of the new type. First, a definition of a distinct object in a distributed environment is required.

> Definition 4-5 Object: A specific instance of a class and interface, having a given state, that is uniquely identified and accessed with a unique object reference that remains valid throughout the lifetime of the object.

As defined previously, the lifetime of an object is from its **Creation** to its **Destruction**. Throughout its lifetime, an object may be incarnated by many servants but remains the same unique object as long as its object reference remains valid. As the CORBA object model hides the location of the object from the client, the object can be incarnated anywhere in the domain. Thus, as long as the object reference enables a client to invoke operations on the object, that object is the same unique entity, regardless of what implementation is used to incarnate it or where it is incarnated.

The concept of metamorphosis, as defined by Webster's New World Dictionary, applies to any entity that undergoes "a change in form, structure, substance, or function" [39]. Therefore the concept of metamorphosis can be applied to objects. A **Metamorphic Object** is an entity that undergoes the process of **Metamorphism**, introduced here.

> Definition 4-6 Metamorphism: A change in form, structure, substance, or function of an object that is uniquely identified and accessed with a unique object reference that remains valid throughout the lifetime of the object.

This definition implies that an object, as identified by a unique object reference, can change its interface definition, class definition, implementation, even its location, and still remain the same entity. There is clear distinction between polymorphism and metamorphism as applied to an operation on an object. Polymorphism is used to invoke

the same operation on a *different* object, referenced with a different object pointer, when the new object supports the matching operation signature as part of its interface. Metamorphism is used to invoke an operation on the *same* object, referenced with the same object reference, after the object has changed its interface and implementation but still supports the matching operation signature as part of its new interface. The difference between polymorphism and metamorphism is illustrated in Figures 4-1 and 4-2, respectively.

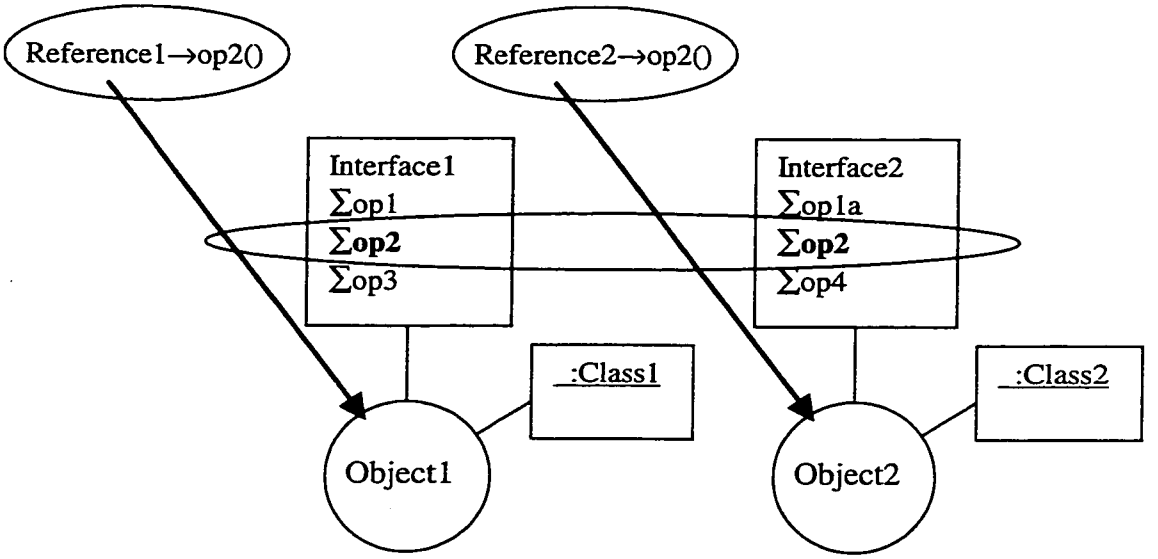

Figure 4-1 Invocation of an operation signature on two Polymorphic objects.

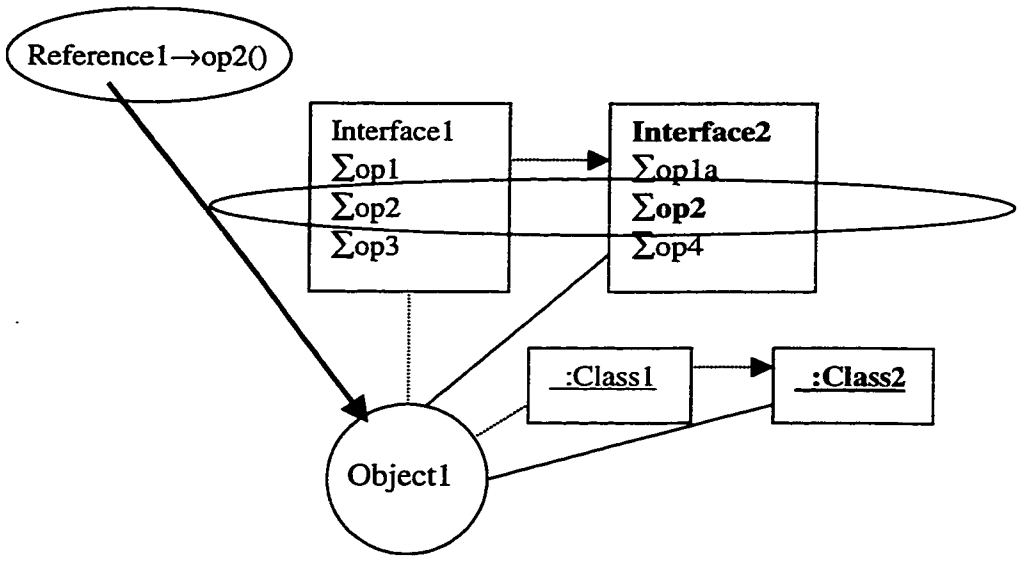

Figure 4-2 Invocation of an operation signature on a single Metamorphic Object.

Figure 4-2 shows the change in interface and class of an existing object, with the operation being invoked on the new *version* of the object. The object changes the interface and class it uses; it does not modify the definition of its initial interface and class. Modification of a class affects all instances of that class in the server process. The concept of metamorphism applies to individual objects, not to classes of objects. This requires that specific instances of a class be transformed into a new interface and class without affecting any other instances of that class, regardless of the server where they exist. The requirement to transform specific instances of metamorphic objects requires that the new version be incarnated with a new implementation on a different server process. Therefore, a **version** of a metamorphic object can be defined.

Definition 4-7 Version: The specific existence of a Metamorphic Object as determined by the set:

$$\text{Version}_i = [\ \text{Interface}_i \cup \text{Class}_i \cup \text{Implementation}_i\ ]$$

where:

Interface_i = a specific IDL Interface definition.

Class_i = a specific class definition in some programming language.

Implementation_i = a specific server process that incarnates the object.

This implies that in order to version a Metamorphic Object to change its interface or class, the object must be incarnated on a new server process that supports the new interface and class. This relocation is equivalent to the object migration described by [21]. But instead of changing the object reference, metamorphism requires that all messages sent to the previous version of the object, MO_{i-1}, using the initial object reference be redirected to the new version, MO_i, on its new server process. A mechanism to realize metamorphic objects using CORBA is proposed in the following section.

4.2 A Forwarding Mechanism for Realizing Metamorphic Objects

In CORBA, an object for a given reference is defined by its IDL Interface, as maintained by an Interface Repository, and its class, as determined by its implementation as part of a server process. It is assumed that the implementation code as part of a server process can not be modified without terminating the process. This is not always true as it possible to extend a class implementation in CORBA while a server process is running [12], and some programming environments such as Smalltalk make it possible to dynamically change a class definition using its metaclass [43, 46]. However these concepts are not part of CORBA and the realization of a metamorphic object must adhere to the CORBA Object Model as defined by the OMG [44].

In order to realize a metamorphic object, mechanisms are required to (1) ensure that operations using an existing object reference are invoked on the most recent version of the target object, and (2) the operation invocation can be processed by the target object to the degree that its new functionality permits. Ensuring that operation requests on an object reference are always invoked on the target object, even after it has changed its interface, implementation, and location, requires a mechanism not explicitly part of the exiting CORBA 2.0 specification. This redirection of messages from one version of a Metamorphic Object to another is introduced here as **Forwarding**.

Definition 4-8 Forwarding: The redirection of client messages from one version of a Metamorphic Object to another to enable operations to be invoked on the most recent version using the original object reference.

The concept of Forwarding is illustrated in Figure 4-3. Note the distinction between Polymorphism and Metamorphism using Forwarding. With polymorphism a client process explicitly uses different object references to access different objects. With the use of forwarding to realize a metamorphic object, the client uses the same object reference on the same object, which to the client, appears to dynamically change its interface. The underlying mechanisms required to realize the new version of the metamorphic object are hidden from the client.

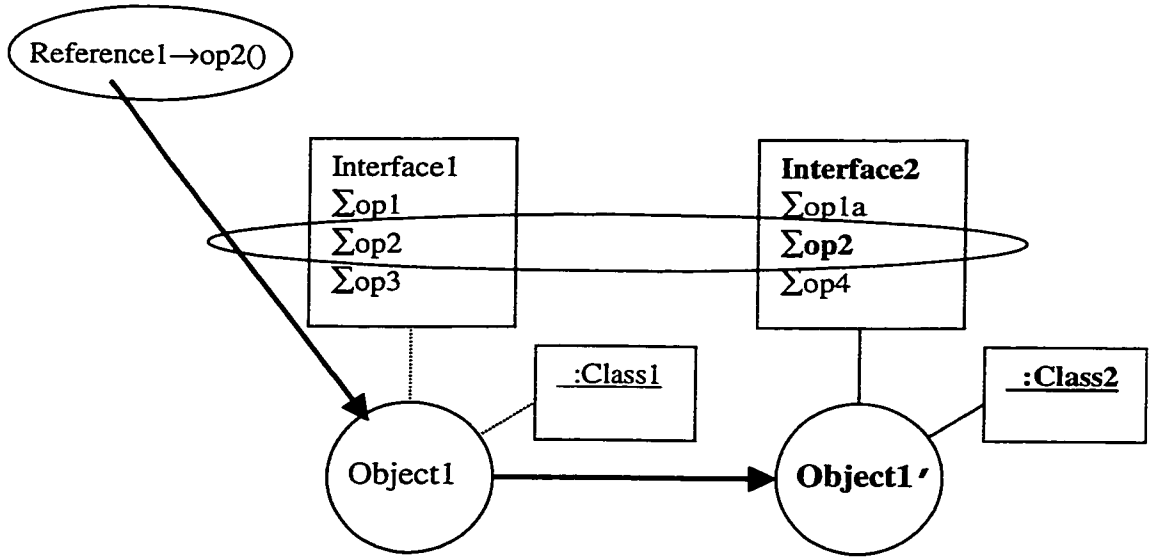

Figure 4-3 Use of Forwarding to realize a Metamorphic Object.

The Forwarding mechanism enables the new version of the metamorphic object to be implemented with a new server process on a new host located elsewhere in the domain, allowing a complete replacement of its implementation. This new implementation can employ a new interface and class definition that was unknown when the initial implementation of the metamorphic object was created.

The forwarding mechanism can be configured in two ways. The simplest approach is to sequentially chain the forwarding servants as shown in Figure 4-4. This requires that each forwarding servant maintain a reference to the next version of the metamorphic object. No references are maintained to previous versions. The disadvantage is that the communication delay caused by each forwarding link increases proportionally with the number of versions of the object.

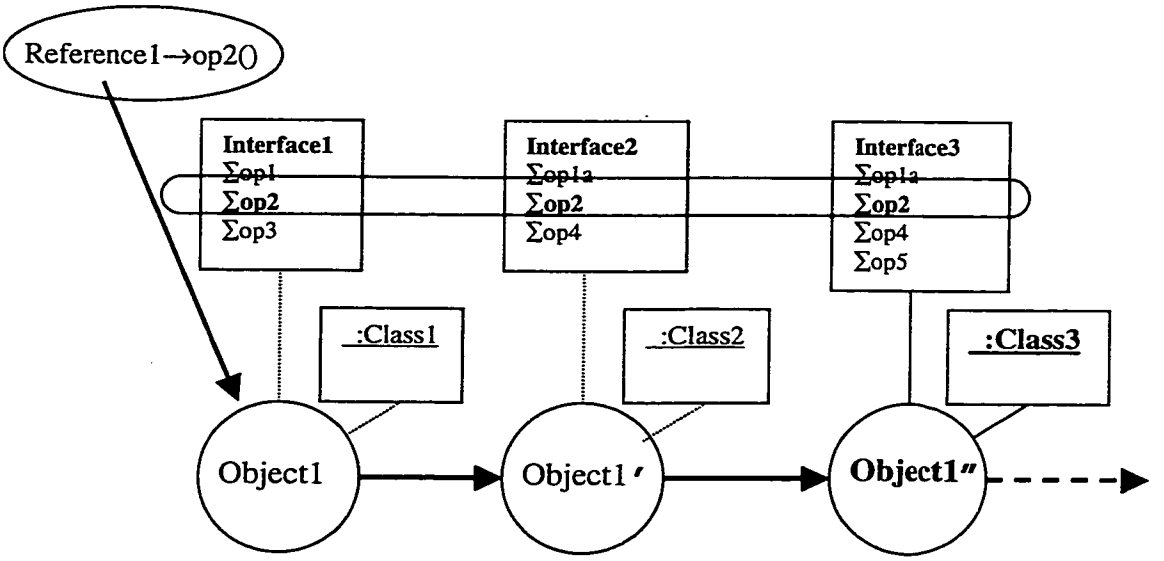

Figure 4-4 Forwarding using sequential chaining.

Another approach to forwarding is to use direct chaining as shown in Figure 4-5. This requires that each forwarding servant maintain a reference to its previous version so that it can change its forwarding reference to the current version of the object. This complicates the implementation of the forwarding servant and increases the time to version an object proportionately to the number of previous versions. The best approach the forwarding depends on the requirements of the application.

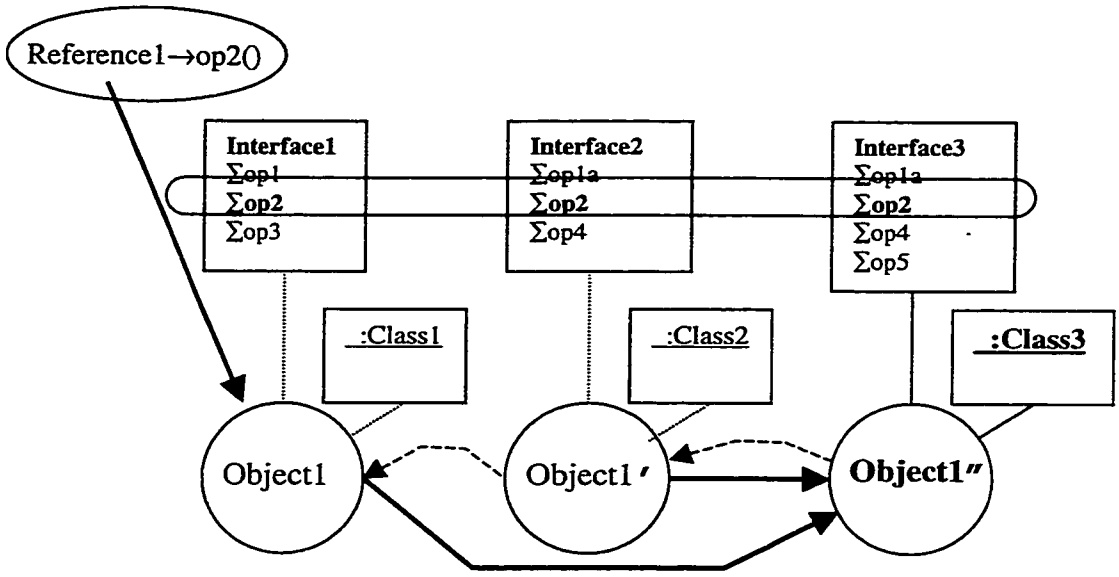

Figure 4-5 Forwarding using direct chaining.

The problem faced by forwarding is how to invoke an operation on the new version of the metamorphic object without knowledge of its interface or implementation. This is achieved through the use of the Dynamic Invocation Interface (DII) in CORBA. As described in section 3.3, the DII enables client applications to send operation requests to a remote CORBA object without predefined knowledge of the object's interface. The client creates a Request object and populates it with the target object reference, the name of the operation, and any parameters to be passed. The object reference used can be of type CORBA::Object. Since an object reference to any CORBA object can be widened to CORBA::Object, a reference of this type can be used on any interface, even if it was unknown when the caller was created. When the request is invoked, the ORB verifies that the operation is supported by the target object's interface before execution. The DII enables an operation to be invoked on an object with a different interface than the original target object as long as the signatures of the specific operation match. In this way, the Dynamic Invocation Interface in CORBA is able to support the versioning process for a new interface, class, and implementation required for metamorphic objects through the use of forwarding.

The design of the Dynamic Versioning Service presented in Chapter 5 includes the *Forwarder Pattern* that details the implementation of forwarding using the DII. It describes the process for forwarding a request to the new version of the metamorphic object once it has been versioned. The process of versioning a metamorphic object is also described in Chapter 5. It requires a mapping function from one version to the next to be able to incarnate the new version of the metamorphic object in a new process. This versioning criterion is described in the final section of this chapter.

4.3 Criteria for Versioning of Metamorphic Objects

As was shown in Figure 4-3, a new version of a metamorphic object is created with a new interface, class, and implementation. A transformation criterion between versions is required for the interface and class definitions. This criterion is used for two purposes: (1) to determine how to map the attributes to the new version during the versioning process, and (2) to determine which operations are compatible between versions during execution of the new version. The first is termed the *versioning criteria* and the second is termed the *metamorphism criteria*. Because these criteria are used for different purposes, the process of versioning of an object and the ability of a client to invoke operations on the new version, they are treated separately. This section presents the axioms required to define these two transformation functions.

The **versioning criterion** applies only to the mapping of attributes between versions. This is used by the Dynamic Versioning Service to copy the appropriate attribute values from the initial object, MO_{i-1}, to the persistent store for the state of the new version, MO_i. The persistent state of an object is determined by a set of its internal attribute values. These persistent attributes are included in the object's implementation class definition. They may or may not be exposed in the object's IDL Interface definition. Since the configuration manager in the Dynamic Versioning Service requires knowledge of these persistent attributes and their types, it is assumed that all persistent attributes are included in the IDL Interface. The DVS could be restricted to attributes defined only in the class

but that would require an additional facility for making the persistent attributes publicly known. Therefore, the versioning criterion applies to attributes defined in the IDL Interface for a metamorphic object.

The versioning criterion can be defined as the set of attributes defined in the interface for the initial version of the metamorphic object that exist in the interface definition for the new version. The signature of each attribute, $\sum s$, consisting of a symbolic name, ζ, and IDL type, τ, must match for the attribute to be included in the set of versioning criteria. Research in the area of Abstract Data Types (ADT) has shown that this versioning criterion can be stated as a partial finite mapping of *sorts* [28, 30]. The following definitions are proposed to establish the versioning criterion for metamorphic objects.

> Definition 4-9 Object Interface: An object interface, M_i, can be defined as a set of attributes, $s \subseteq S_i$, each defined as a sort of symbol and type, $\sum s \in (\zeta, \tau)$, and a set of operations $f \subseteq F_i$.

> Definition 4-10 Operation Signature: An operation is defined by a signature of a function symbol, $\sum f$, and an ordered set of sorts representing the argument parameters, $(\zeta, s_{arg} \subseteq S)$, and a sort representing the return value, s_r. An operation signature is defined as: $\sum f \in (\zeta, s_{arg} \subseteq S, s_r)$.

The initial version of a metamorphic object has an interface M, and corresponding class and implementation, and the new version of the metamorphic object has an interface N. The versioning criteria can be defined as follows.

> Definition 4-11 Versioning Criteria: Let ρ_s be a partial finite mapping function of attributes for the relationship between the interfaces, $M \times N$. That is, ρ_s is a metamorphism on M with values in N, and is written as either $m\ \rho_s\ n$ or $\rho_s\ (m, n)$.

The partial finite mapping, ρ_s, of attributes from interface *M* to interface *N* determines the compatibility of the *states* between the versions *M* and *N* of the metamorphic object. The mapping function ρ_s is used during the versioning process to determine the compatibility of the persistent states of versions *M* and *N* and to copy the appropriate attribute values from *M* to the matching attributes in *N*. The state compatibility of version *M* and *N* is said to be *total* if and only if the set of all attribute signatures, consisting of a symbol and type, included in version *M* map to the corresponding set of all attribute signatures included in version *N*. The degree of compatibility between versions as determined by the versioning criteria, ρ_s, is categorized as *total*, *compatible*, *partially compatible*, and *disjoint*. The compatibility of versions of an object is *total* if and only if all the state attributes defined in version *M* are the same as all attributes defined in version *N*. Versions are *compatible* if and only if all attributes defined in *M* exist in *N*. Versions are *partially compatible* if and only if some attributes defined in *M* exist in *N*. Versions are *disjoint* if none of the attributes defined in *M* are defined in *N*. These are defined in an abstract way below.

Definition 4-12 Degrees of Versioning Compatibility:

The state compatibility of versions *M* and *N* is said to be *total* iff

$$\{\forall(\Sigma s \in (\zeta,\tau)) \subseteq M\} \xrightarrow{\rho_S} \{\forall(\Sigma s \in (\zeta,\tau)) \subseteq N\}.$$

Versions *M* and *N* are said to be *compatible* iff

$$\{\forall(\Sigma s \in (\zeta,\tau)) \subseteq M\} \xrightarrow{\rho_S} \{\exists(\Sigma s \in (\zeta,\tau)) \subseteq N\}.$$

Versions *M* and *N* are said to be *partially compatible* iff

$$\{\exists(\Sigma s \in (\zeta,\tau)) \subseteq M\} \xrightarrow{\rho_S} \{\exists(\Sigma s \in (\zeta,\tau)) \subseteq N\}.$$

Versions *M* and *N* are said to be *disjoint* iff

$$\{(\Sigma s \in (\zeta,\tau)) \subseteq M\} \cap \{(\Sigma s \in (\zeta,\tau)) \subseteq N\} = \varnothing.$$

Note that any mapping, ρ_s, is only isomorphic if the compatibility of the versions is total. Version *M* may be compatible with version *N* but version *N* may be only partially compatible with version *M*, and visa versa. It is the responsibility of the application developer to determine the degree of compatibility required between versions. For example, it may be desirable to version a metamorphic object to a new interface that did not contain any matching attributes (versions *M* and *N* are *disjoint*) if the transfer of state information was not required. The compatibility of the metamorphic object after versioning is determined by the operations that are common between the versions, not the attributes. The mapping of operation signatures is termed the *metamorphism criteria.*

The metamorphism criteria is defined as the partial finite mapping, ρ_f, of operations from interface *M* to interface *N*. This determines the compatibility of the *operations* between the two versions. To determine the degree of metamorphism it is sufficient to demonstrate that the principle of polymorphism applies to operations common between versions of a metamorphic object. This relies on CORBA's support for *polymorphic assignments* and dynamic binding [7]. "The type of object reference used to access an object does not determine the code executed for its operation and attributes. Instead this is determined by the implementation class that the object is an instance of." [7, p198]. Therefore, if the signature of an operation is identical between any two interfaces, then (1) the same operation request can be invoked on any object supporting either interface, and (2) the code that is executed is determined by the implementation class of the object receiving the request. This is based on the principle of *ad hoc polymorphism* [8, 9], which allows an operation to be invoked on any object whose interface supports that operation signature. This differs from *inheritance polymorphism* that allows an instance of an object's subclass to be substituted for the target object class in an operation invocation. Since an operation can be invoked on any class of metamorphic object that support the operation's signature, it is only necessary to determine the degree of compatibility between two versions of a metamorphic object. The compatibility is based on the number of matching operating signatures between the two versions.

The metamorphism criterion for compatible operations between versions can be defined in the same manner as the versioning criterion for attributes. Let the operations of an interface, M_i, be defined as a set of operations $f \subseteq F_i$. Each operation is defined by a signature of a function symbol, an ordered set of sorts representing the argument parameters, and a sort representing the return value, $\Sigma f \in (\zeta, s_{arg} \subseteq S, s_r)$. The initial version a metamorphic object has interface M and the new version of the metamorphic object has interface N. The metamorphism criteria between interface M and N is defined below.

> Definition 4-13 Metamorphism Criteria: Let ρ_f be a partial finite mapping function of operations for the relationship between the interfaces, $M \times N$. That is, ρ_f is a metamorphism on M with values in N, and is written as $m\ \rho_f n$ or $\rho_f(m, n)$.

The partial finite mapping, ρ_f, of operations from interface M to interface N determines the metamorphism compatibility between the versions. The compatibility of operations between version M and N is said to be *total* if and only if the set of all operation signatures included in the interface for version M map to the corresponding set of all operation signatures in version N. The degree of compatibility between versions, as determined by the metamorphism criteria, ρ_f, is categorized as *total*, *compatible*, *partially compatible*, and *disjoint*. The metamorphism compatibility of versions of an object is *total* if and only if all the operation signatures defined in version M are the same as all operation signatures defined in version N. Versions are *compatible* if and only if all operation signatures defined in M exist in N. Versions are *partially compatible* if and only if some operation signatures defined in M exist in N. Versions are *disjoint* if none of the operation signatures defined in M are defined in N. These are defined in an abstract way in Definition 4-14.

Definition 4-14 Degrees of Metamorphism Compatibility:

The metamorphism compatibility of versions M and N is said to be *total* iff

$$\left\{\forall\left(\Sigma f \in (\zeta, s_{\arg} \subseteq S, s_r)\right) \subseteq M\right\} \xrightarrow{\rho_f} \left\{\forall\left(\Sigma f \in (\zeta, s_{\arg} \subseteq S, s_r)\right) \subseteq N\right\}.$$

Versions M and N are said to be *compatible* iff

$$\left\{\forall\left(\Sigma f \in (\zeta, s_{\arg} \subseteq S, s_r)\right) \subseteq M\right\} \xrightarrow{\rho_f} \left\{\exists\left(\Sigma f \in (\zeta, s_{\arg} \subseteq S, s_r)\right) \subseteq N\right\}.$$

Versions M and N are said to be *partially compatible* iff

$$\left\{\exists\left(\Sigma f \in (\zeta, s_{\arg} \subseteq S, s_r)\right) \subseteq M\right\} \xrightarrow{\rho_f} \left\{\exists\left(\Sigma f \in (\zeta, s_{\arg} \subseteq S, s_r)\right) \subseteq N\right\}.$$

Versions M and N are said to be *disjoint* iff

$$\left\{\left(\Sigma f \in (\zeta, s_{\arg} \subseteq S, s_r)\right) \subseteq M\right\} \cap \left\{\left(\Sigma f \in (\zeta, s_{\arg} \subseteq S, s_r)\right) \subseteq N\right\} = \varnothing.$$

A high degree of compatibility is required for the metamorphism criteria to be viable. If two interfaces are disjoint with respect to their operations, then there is no logical reason to version a metamorphic object between the two interfaces, even though such a versioning could be applied. If the interfaces are disjoint, then no operations invoked by a client will be executed since the new version will not support them (except for the base-class operations of MetamorphicObject, which are not included in the measure of compatibility). This would make the object unusable with its initial object reference. If a complete change in interface and object reference is required, then a new and separate object should be introduced into the application. Metamorphism is intended for incremental changes in object behavior in order to support a controlled evolution of distributed applications. As such, it is required that the degree of compatibility for a given metamorphism criteria between versions be *total*, *compatible*, or *partially compatible*.

In the case where two interfaces provide *total compatibility*, a metamorphic object may be versioned between the two interfaces in order to change its location (host), implementation (server process), or class definition (internal functionality), but maintain the same client operations. The attributes included in the two interface definitions may be different, affecting only the transient or persistent state of the object and its versioning criteria. Reasons to create a new version of a metamorphic object with a totally compatible interface may include:

- corrections to implementation deficiencies (bugs) in the code;
- providing a completely new implementation, for example by porting the object from a C++ implementation on a Unix host to a Java implementation on a WindowsNT host, or;
- changes to internal components of the object, such as its persistent store.

The two interfaces would have separate and distinct interface names in IDL. Although versioning of IDL interfaces (using a number system, i.e. 1.0) is included in the CORBA 2.0 specification, most vendors do not yet implement versioning as part of the Interface Repository [7]. Thus, two distinct interfaces require distinct symbols for their IDL names in order for both to exist in the same Interface Repository. The use of interface version numbers in the Interface Repository would be preferable for small changes in the version of a metamorphic object. Configuration management of metamorphic objects, although not addressed here, would be a sizable task for a commercial system.

Versioning of a metamorphic object to a *compatible* interface is similar to replacing an object of one class with an instance of its subclass, where the subclass is used to add new functionality, including attributes and operations, or to over-ride the implementation of operations in the parent class. The important difference is that the new version need not be a subclass of the initial version, i.e. metamorphism does *not* rely on inheritance polymorphism. This is illustrated in Figure 4-6.

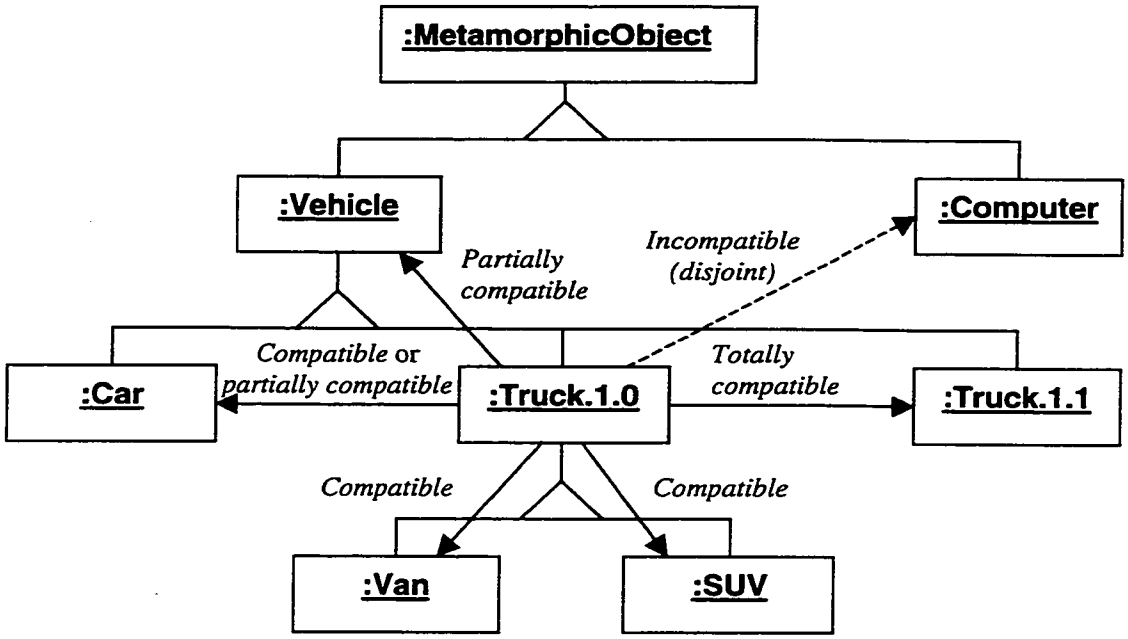

Figure 4-6. Compatibility between versions of MetamorphicObject subclass Truck.

Instances of Truck would be compatible with Car, Van, or SUV (Sports Utility Vehicle), but not Computer. The subclass of Truck, Van, and SUV could be used to extend Truck, whereas Car could have different behavior for the same operations defined in the interface for Truck as well some different operations. If Car supported all of the operation signatures defined for Truck, then it would be dubbed *compatible* even if the functionality of the operations was different or the attributes were different. If Car supported some but not all of the operation signatures defined for Truck, then it would be termed *partially compatible* with Truck. An instance of Truck (a MetamorhicObject) could be versioned to Car, Van, or SUV with varying degrees of compatibility.

Figure 4-6 also illustrates some of the complexities of dealing with MetamorphicObjects. This includes configuration management of the class and interface hierarchy with multiple versions of classes and interfaces each potentially having separate subclasses. This would become even more complicated in a distributed environment where the hierarchies of interfaces, classes, and implementations must be consistent throughout the

domain. A sophisticated Integrated Development Environment (IDE) with a distributed source and implementation code management facility would be required.

A complete CORBA service used to deploy Metamorphic Object applications would also have to provide a Version Manager Service to maintain a directory of all versioning and metamorphism criteria between any and all compatible versions. The compatibility of different mappings from a given interface, *M*, is determined as follows [30]:

ρ is *total* iff $dom(\rho_f) = M$.

ρ_1 and ρ_2 are *compatible* iff $\rho_1[m] = \rho_2[m]$ for all $m \in dom(\rho_1) \cap dom(\rho_2)$.

ρ_1 and ρ_2 are *disjoint* iff $dom(\rho_1) \cap dom(\rho_2) = \varnothing$.

$\rho_1 \subseteq \rho_2$ iff ρ_1 and ρ_2 are compatible and $dom(\rho_1) \subseteq dom(\rho_2)$.

If ρ_1 and ρ_2 are compatible then the mapping $\rho_1 \cup \rho_2$ is characterized by

$$\rho_1 \cup \rho_2[m] = \begin{cases} \rho_1[m], \text{ if } m \in dom(\rho_1) \\ \rho_2[m], \text{ if } m \in dom(\rho_2) \\ \text{undefined otherwise} \end{cases}$$

The complexity of maintaining these mappings could be of order *O(exp(n))*, where n is the number of interface definitions in the domain, including versions of the same base interface, since any instance of a metamorphic object can, in theory, be mapped to any other type of metamorphic object.

The implementation of the Dynamic Versioning Service for CORBA, described in Chapter 5, does not attempt to provide the extensive configuration tools required for managing a distributed metamorphic object application. The versioning criteria and metamorphism criteria for the test applications in Chapter 6 are based on the theory provided here but are not determined by an automated service. The Dynamic Versioning Service is an experimental system designed and developed to demonstrate metamorphic functionality and to measure the performance impact of using the concept of Forwarding. It also demonstrates that metamorphic objects can be created and versioned as described by the concepts presented in this thesis.

5 A Dynamic Versioning Service for CORBA

An architectural framework is presented here for realizing metamorphic objects in a distributed computing environment using CORBA. The Dynamic Versioning Service is a specification for developing distributed applications that support dynamic reconfiguration using the concept of Forwarding presented in Chapter 4. The "Forwarder" design pattern is presented as a blueprint to implement this concept. The core server components provided by the Dynamic Versioning Service are specified using the Unified Modeling Language (UML) and, where appropriate, the public interface is specified using the OMG Interface Definition Language (IDL). These include the base-classes for MetamorphicObject, MOFactory, MOLoader, and MOManager. An IDL specification and UML design for a Metamorphic Object Store to provide persistence to any subclass of MetamorphicObject is also presented. Other components of the Dynamic Versioning Service required to support a commercial implementation are discussed but not implemented here. Sequence diagrams are used to illustrate the operation of the Dynamic Versioning Service in creating and versioning metamorphic objects and the process of Forwarding. This specification for a Dynamic Versioning Service is used to develop the test application presented in Chapter 6.

5.1 Functions of a Dynamic Versioning Service

A Dynamic Versioning Service for CORBA requires many components to implement and manage the versioning of Metamorphic Objects throughout a distributed environment. Such a service requires the following functions:

1. **Server Framework**. Provides the core components of any server process for Metamorphic Objects. This includes the base-classes for MetamorphicObject, MOFactory, and MOManager that are sub-classed for each application. It also

includes a generic MOLoader required to incarnate object servants and support the Forwarder pattern.

2. **Persistence.** Provides a centralized or distributed persistent store that is accessible by all metamorphic object servers in the domain and capable of storing the attributes of any metamorphic object. The store must be able to dynamically add new schema for new metamorphic object classes or versions, without interruption to any existing objects.
3. **Forwarding.** Maintains object reference consistency throughout the network. This is achieved by redirecting all messages to the new incarnation of the object. No object reference should be made invalid because of the versioning process. No client process should block or receive an exception because a metamorphic object was versioned to a new interface, class, or implementation.
4. **Versioning.** Provide a configuration management application to (1) select and transform a selected instance of one metamorphic object class to an instance of another compatible class on another server implementation, and (2) migrate the persistent state of the object to the persistent store of the new version of the object. It must also be able to select and version large sets of objects across multiple server processes.
5. **Interface Management.** Maintains a centralized or federated Interface Repository of IDL Interface definitions for all versions of all Metamorphic Object classes in the network. A configuration tool is required to add, delete, and modify versions of interface definitions in the Interface Repository and maintain consistency of the interface hierarchy throughout the domain.
6. **Implementation Management.** Maintains a centralized or federated Implementation Repository for all server implementations (executables) deployed throughout the network. These must be Linked with the corresponding interface definitions for all server classes in the Interface Repository. A tool is required for locating one or more implementations (servers) of a Metamorphic Object class from the corresponding IDL interface.
7. **Criteria Management.** Maintains a repository of translation schema from one Metamorphic Object class definition to another for all compatible class

definitions. This includes management of all versioning criteria and metamorphism criteria defined for each version of each class of metamorphic object in the domain.

8. **Reference Rebinding.** Provides a means for clients to re-establish a direct connection to the new version of the object. This could involve updating of the symbol binding to the object reference in the Naming Service. The re-binding of object references must be performed under program control, not as a result of an object fault exception. This would provide an incremental and controlled evolution of distributed applications.

Note that the Dynamic Versioning Service is not responsible for any modifications to the client application required to adapt to the changes offered by the new version of the object class. The client applications may be designed to dynamically query an object's interface in the IFR and exploit any new functionality using the DII. The DVS specifies that an Invalid Operation exception is to be raised by the server when a client attempts to invoke an operation signature that is no longer supported. The client applications must handle these exceptions. It is the responsibility of the application manager to ensure that the consequences of versioning a metamorphic object is dealt with in all related client processes.

The design and implementation of all the functionality required for a complete CORBA service is clearly beyond the scope of this thesis. The aim of the Dynamic Versioning Service prototype presented here is to demonstrate the ability to modify the interface and implementation of one or more instances of a Metamorphic Object (MO), without invalidating any object references in the system. To support these Metamorphic Objects, the DVS prototype provides only the following essential functions: **Server Framework**, **Persistence**, **Forwarding**, and **Versioning**. The minimum functionality required for **Interface** and **Implementation Management** is provided by the Orbix® ORB [7, 41, 42]. **Criteria Management** is performed manually using a simple Manager client application that performs the **Versioning**. **Reference Rebinding** is not implemented although an approach similar to the one described in [21] could be used. The detailed

designs for the essential components of the Dynamic Versioning Service are presented in section 5.4.

5.2 Development of DVS Applications

The Dynamic Versioning Service is fully compliant with the CORBA 2.0 specification and is implemented at the application level without modification to the ORB or IDL compiler. This allows the framework defined here to be used for the development of dynamically reconfigurable applications using virtually any vendors' ORB. The DVS uses the CORBA Naming service but not the Transaction Service, Persistent Object Service, or any other CORBA services, although other services could be used in a DVS application. This section discusses the development issues of a DVS application.

The Dynamic Versioning Service defines a *client* to be any process that invokes an operation on an object using a CORBA object reference. This includes objects on a server process that hold object references as attributes through aggregation, i.e. composite objects. A *manager* is a client application that can invoke operations to create, destroy or version a MO on a server. A *server* is an application that incarnates a MO by providing an implementation to service requests on it. Servers can also provide a factory object that can create, destroy, or version instances of a MO. A Dynamic Versioning Service application consists of a number of metamorphic object servers that maintain the persistent state of all metamorphic objects in a shared MOStore. There are no restrictions on client processes. They may be other DVS server processes, or typical CORBA client applications using either static proxies or the DII to access server objects.

Applications may contain any number of versions of MetamorphicObject classes implemented on any number of server processes. The only physical limitation, aside from the memory and processing capabilities of the host machines, is the ability to maintain multiple versions of the IDL Interface on all hosts in the domain. Because most CORBA 2.0 compliant Interface Repositories do not support versioning of IDL Interfaces, this

requires that each version of Interface be identified with a unique symbol. The following criteria, based on the definitions in chapter 4, define a new version of Metamorphic Object:

- A version of a MetamorphicObject class is defined as the unique set of IDL Interface, Class, and server Implementation.
- A version of a MO class can have any IDL Interface that directly or indirectly inherits from MetamorphicObject. Since IDL supports multiple inheritance, it is possible to modify any Interface to be a version of a Metamorphic Object. The server implementation must use the DVS framework.
- A new version of a MO class can be defined by sub-classing any version of a MO Interface. A corresponding server implementation must use the DVS framework.
- A new version of a MO class can be defined by modifying an existing Interface to create a new Interface that shares the same parent Interface. A corresponding server implementation must use the DVS framework.
- The Interface for a version of MO may have attributes that can be any object type including other CORBA objects as long as they are supported by the persistent store.
- The Interface for a version of MO may have operations that support any type of argument definable in IDL.
- The Interface for a version of MO may have operations that support all IDL parameters passing modes including IN, OUT, INOUT, and RETURN types.

The result of the development process for a new version of MO is a set of source files. These include an IDL file that contains the interface definition of the new MO class, C++ files for the servant object that implements the desired operations of the MO class, and C++ files for the server implementation, including any factory objects and other components of the server. Note that a new version of a MO may only require a new C++ servant class and may not require a change in the interface.

These files are installed on one or more server hosts in the network. This requires that the IDL definition of the new MO class be installed in the Interface Repository of every host

to maintain consistency of the class hierarchy across all servers in the network. The C++ code must then be installed and compiled on each of the hosts supporting the MO. The server executable is then registered with the ORB's implementation repository so that it can be launched [7]. When the server starts-up, it must be configured to access the persistent store for the new class of MO and any other services, such as the Naming Service. This installation could be performed manually or through the use of an automated process.

The development process also defines a mapping scheme from one version of a MO class to another. This versioning criterion is used to translate the persistent state of a MO from the current schema in which it is stored to a new schema. The server framework of the DVS provides the mechanism for processing this versioning criterion when the MO is versioned to transfer its persistent state to its new incarnation. The versioning process is explained in section 5.5.

Most DVS applications will also make use of the CORBA Naming Service to store references to MOs. The Naming Service is intended to be the primary means by which a client initially obtains an object reference. Once an object reference is bound to a name in the Naming Service, it must remain valid until a server explicitly changes it. Rebinding the name to an object reference to the new version of the MO could provide a means for clients to re-establish a direct connection to the new version of the object. The object references bound in the Naming Service are **not** modified in the DVS prototype when a MO is versioned.

5.3 The Forwarder Design Pattern

The Forwarder Pattern is a key component of the framework for the Dynamic Versioning Service. The format of a *Design Pattern Template* for CORBA applications as defined by [34, 35] is briefly described. The Forwarder Pattern is then presented using this format. The Forwarder pattern fulfils all of the requirements for a CORBA design pattern as

specified in [35]. The implementation and use of the Forwarder Pattern is included in the detailed design of the DVS in the following sections.

5.3.1 CORBA Design Pattern Template

A design pattern is a problem-solution pair, accompanied by an associated context and an identification of the forces that more precisely define the problem including an example. The elements of a design pattern are described below. A detailed description of each element can be found in [35].

- **Most Applicable Scale.** The architectural level to which the pattern applies including: Objects and Classes, Framework, Application, System, Enterprise, Global/Industrial. These are defined in the Scalability Model [35].
- **Solution Type.** The four types included in the pattern catalog are Software, Technology, Process, and Role. Most patterns are of type Software.
- **Solution Name.** A new and unique term introduced to reference the pattern.
- **Intent.** A brief statement of the problem intended to be solved by the pattern.
- **Diagram.** A visual abstraction of the solution.
- **Primal Forces.** The forces, as defined in the Scalability Model, that are addressed, including the management of Functionality, Performance, Complexity, Change, IT Resources, and Technology Transfer.
- **Applicability at This Scale.** A list of motivating factors affecting the use of this pattern at a particular scale.
- **Solution Summary.** A detailed explanation of how the pattern solves the problem in relation to the forces identified.
- **Benefits.** The benefits of applying this solution including comparisons to related patterns.
- **Consequences.** Any undesirable consequences of applying the solution with emphasis on the difficulties unique to the CORBA environment.

- **Variations of the Solution.** Any alternative designs or options for implementing the pattern.
- **Rescaling to Other Levels.** The relevancy of this pattern to other scales.
- **Related Solutions.** References to other related patterns.
- **Example.** An example of the solution being applied to a particular problem using CORBA.
- **Background.** Any useful background information about the development of the pattern.

As seen above, a design pattern provides a concise statement of a problem and solution that can be easily referenced to determine its relevancy to a given problem. The Forwarder pattern is presented using this format, but the full application of this solution is best illustrated in its role as part of the DVS.

5.3.2 Forwarder

Most Applicable Scale: Microarchitecture

Solution Type: Software

Solution Name: Forwarder

Intent: To enable the versioning of an active server object to a new interface definition and implementation without changing or invalidating any references to the object held by client processes.

Diagram:

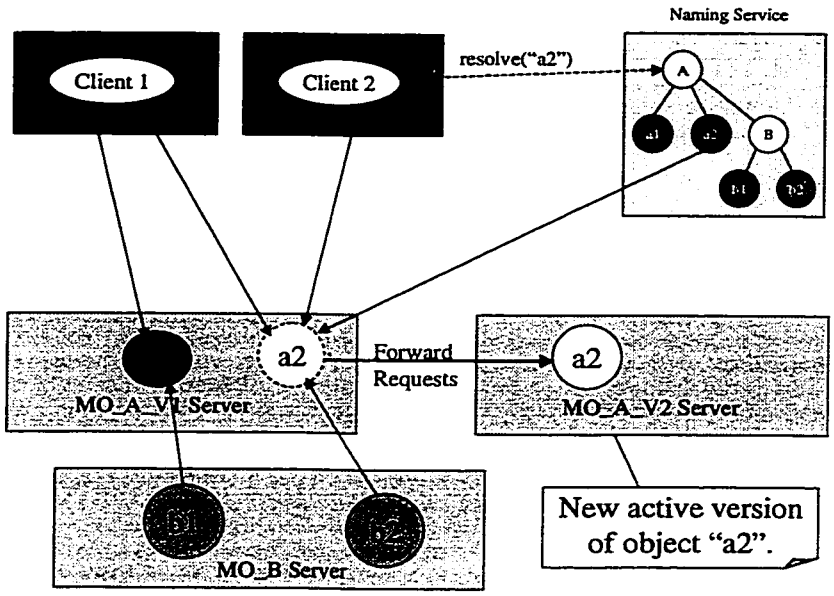

Figure 5-1 Forwarding of operation requests to a new Interface and Implementation.

Primal Forces: Management of Change, Management of Functionality

Applicability at This Scale

1. The interface, functionality or implementation of individual server objects is expected to change during the lifetime of the application.
2. It is a real-time or mission-critical application involving distributed objects referenced by client processes or objects on other servers.
3. No loss of service is acceptable for modifying active server processes.
4. No existing object reference can be made invalid because the application is modified.
5. A small degradation in performance is acceptable.
6. CORBA is used as the infrastructure mechanism.

Solution Summary

The Forwarder pattern provides a mechanism to maintain the validity of all object references held by client processes during and after versioning. An object that can be versioned to a new interface or implementation class definition is defined as some subclass of MetamorphicObject (MO). A new version of a MO is created in a new server process with a new interface or implementation that may be unknown when the initial object was created. After a new MO implementation has been incarnated, all operations using an existing object reference are invoked on the new version of the MO. The previous incarnation of the MO is changed to forward all operations to the new version of the MO to be processed, as shown in Figure 5-1. Any results from the new MO are passed back through the previous MO to the client. This maintains the illusion to the client that it is referencing the original MO even though its interface, class definition, and implementation have changed. If the new version of the MO does not support the requested operation, an Invalid Operation exception will be raised for the client. The ability to forward invocations to the new MO is provided by the *Forwarder Pattern* in the implementation of the servant class for a CORBA object.

The Forwarder Pattern uses two different servant implementations for a given MO interface: the *Implementer Class* and the *Forwarder Class*. The Implementer servant class provides the desired functionality for the current active version of the MO. When a MO is versioned, the Implementer servant is replaced with the Forwarder servant in the initial active server process. The Forwarder servant is incarnated to serve requests on the original MO object reference by forwarding them to the new version. The Forwarder Class has the same implementation methods as the Implementer Class but no attributes other than a CORBA object reference to the new version of the MO. An Example of the Forwarder is shown in the UML class and sequence diagrams in Figures 5-2 and 5-3, respectively.

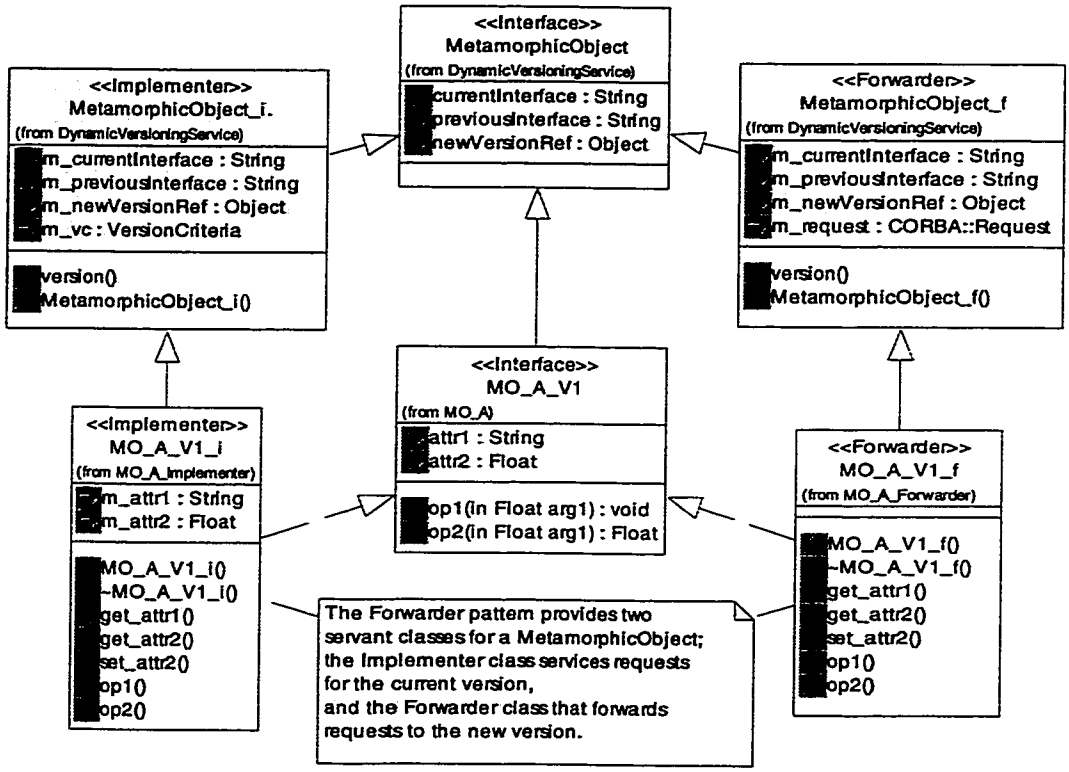

Figure 5-2 Example Class Diagram for the Forwarder Pattern.

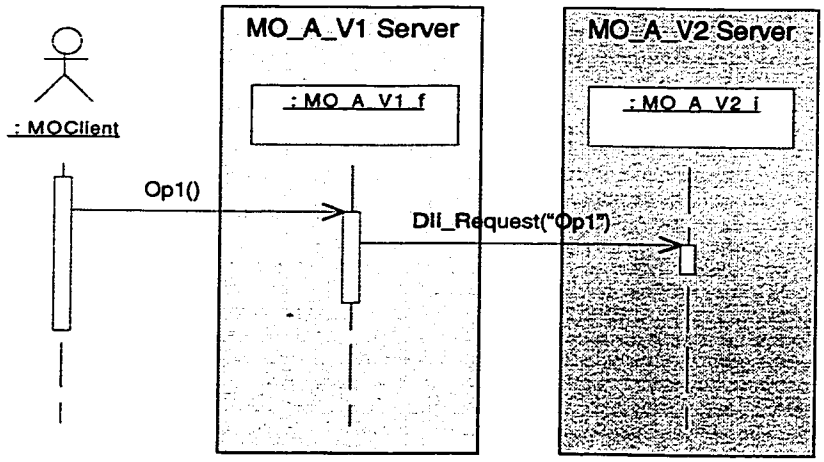

Figure 5-3 Example Sequence Diagram for Forwarding

Each Forwarder operation uses the Dynamic Invocation Interface (DII) invocations to forward the request to the new version of the object. A DII Request message is created using the symbol of the operation and all arguments; it returns a value of the same type, if required. The signature of each Forwarder class can be derived directly from the IDL specification for the MO class. The C++ implementation code for each Forwarder operation can be generated automatically from the IDL specification because each operation invokes a DII request whose parameters and types are defined by its IDL operation. This allows a macro to generate a companion Forwarder class for each Implementer class written by the developer. Because the Forwarder servant does not maintain the state of the MO after it has been versioned, there is no requirement for the Forwarder to have any attributes other than an object reference to the new version of the MO. Each Forwarder class only supports the interface for its version of the MO. This allows the implementation code for the Forwarder servant to be deployed with each version of the MO class.

A *Loader* and *Manager* perform the replacement of the Implementer servant with the Forwarder servant during versioning. The Manager is responsible for versioning the MO and removing its Implementer servant from the original server process. This does not destroy the MO, as the next operation invocation on it will cause the ORB to invoke the Loader. The Loader will then incarnate the Forwarder servant to handle the operation request.

Benefits

- Allows distributed applications to be modified "on-the-fly".
- New interfaces and functionality can be defined and implemented after a system is deployed.
- No change is required to the Implementer implementation class; a separate Forwarder class is generated from the IDL interface.
- Object reference integrity is maintained during and after versioning.
- No client or server process is blocked during versioning.

- No change is required in client processes.

Consequences

- Increased code complexity.
- Two implementation classes are required for each IDL interface.
- A Loader and Manager object is required in each server process.
- A delay equivalent to the time of one operation invocation is added to each client operation invocation for each forwarding link.

Variations of the Solution

A single servant implementation class could be created that incorporates the Forwarder DII request in each operation. This would require every operation to be comprised of an Implementer segment and a Forwarder segment. The servant could then change mode from Implementer to Forwarder by the setting of an attribute value. The Forwarder segment would be executed instead of the Implementer segment after the object has been versioned. This solution increases the complexity of the implementation class and could increase the complexity of the persistent store for the object.

Rescaling to Other Levels

Forwarding could be implemented at the Object level as described above. The general concept of forwarding could also be implemented at the framework and application level.

Related Solutions

- Instance Reference – This pattern provides a mechanism for mapping from the implementation of an object's interface to a specific object instance. This is similar to the mapping between the interface and Implementer or Forwarder servant instance [35].
- Binding Modification – Although not presented as a pattern, this is a concept of modifying the object reference bindings in the client process to redirect

operation invocations to new versions of an object in a new server process [21]. This is an alternative approach to Forwarding for dynamic reconfiguration of distributed applications.

- Dynamic Attributes – This pattern enables the modifying of object attributes at run-time without being forced to recompile the object's IDL [35].

Example

A distributed application is developed to manage bank accounts of type `CheckingAccount`, shown in Figure 5-4, uses the forwarder pattern for the C++ servant implementation. The two servant classes, shown in Figures 5-5 and 5-6, have the same signature. The `CheckingAccount` operation `withdrawal()` only changes the balance, no fees are charged.

```
// IDL
interface CheckingAccount : DVS::MetamorphicObject {
      exception InvalidAmount { float amount; };
      readonly attribute string accountNumber;
      readonly attribute float balance;

      void deposit (in float amount)
            raises (InvalidAmount);

      float withdrawal (in float amount)
            raises (InvalidAmount);
};
```

Figure 5-4 Example Application for Forwarder pattern.

```
// C++ Implementer Servant
CORBA::Float CheckingAccount_i::withdrawal (CORBA::Float f,
                                    CORBA::Environment& IT_env)
                    throw (CORBA::SystemException,
                           DVSCA::CheckingAccount::InvalidAmount){
      float amountReturned;

      if (f < 0.0)
            throw DVSCA::CheckingAccount::InvalidAmount(f);

      if (f > m_balance){
            amountReturned = m_balance;
            m_balance = 0.0;
      }
      else {
            m_balance -= f;
            amountReturned = f;
      }
      this->CORBA::Object::_save(); //save to persistent store
      return amountReturned;
}
```

Figure 5-5 Example C++ Implementer servant.

```
//C++ Forwarder Servant
CORBA::Float CheckingAccount_f::withdrawal (CORBA::Float f,
                                    CORBA::Environment& IT_env)
                    throw (CORBA::SystemException,
                    DVS::MetamorphicObject::InvalidOperation){

      CORBA::Float returnValue;
      m_request.reset();
      m_request.setTarget(m_newVersionRef);
      m_request.setOperation("withdrawal");
      m_request.set_return_type(CORBA::_tc_float);
      m_request << CORBA::inMode << f;
      try {
            m_request.invoke();
      } catch (...){
            throw DVS::MetamorphicObject::InvalidOperation();
      }
      m_request >> returnValue;
      return(returnValue);
}
```

Figure 5-6 Example C++ Forwarder servant.

The application is deployed and many instances of `CheckingAccount` are created with several client applications holding references to them. It is later

decided to modify the behavior and interface of `CheckingAccount` to add a service charge for withdrawals. A new IDL Interface for `PremiumAccount` is created, shown in Figure 5-7, along with a new C++ implementation using the Forwarder pattern, as shown in Figures 5-8 and 5-9.

```
//IDL
interface PremiumAccount : DVS::MetamorphicObject {
      exception InvalidAmount { float amount; };
      readonly attribute string accountNumber;
      readonly attribute float balance;
      readonly attribute float interestRate;
      readonly attribute float serviceCharge;

      void deposit (in float amount)
            raises (InvalidAmount);

      float withdrawal (in float amount)
            raises (InvalidAmount);
};
```

Figure 5-7 Example IDL Interface for a new version of MO.

```
//C++ Implementer Servant
CORBA::Float PremiumAccount_i::withdrawal (CORBA::Float f,
                                 CORBA::Environment& IT_env)
                  throw (CORBA::SystemException,
                        DVSPA::PremiumAccount::InvalidAmount){
      float amountReturned;

      if (f < 0.0)
            throw DVSPA::PremiumAccount::InvalidAmount(f);

      if (f > (m_balance + m_serviceCharge)){
            amountReturned = (m_balance - m_serviceCharge);
            m_balance = 0.0;
      }
      else {
            m_balance = m_balance - (f + m_serviceCharge);
            amountReturned = f;
      }
      this->CORBA::Object::_save();
      return amountReturned;
}
```

Figure 5-8 Example of C++ Implementer servant for the new version.

```
// C++ Forwarder Servant
CORBA::Float PremiumAccount_f::withdrawal (CORBA::Float f,
                                  CORBA::Environment& IT_env)
                    throw (CORBA::SystemException,

       DVS::MetamorphicObject::InvalidOperation){

       CORBA::Float returnValue;
       m_request.reset();
       m_request.setTarget(m_newVersionRef);
       m_request.setOperation("withdrawal");
       m_request.set_return_type(CORBA::_tc_float);
       m_request << CORBA::inMode << f;
       try {
             m_request.invoke();
       } catch (...){
             throw DVS::MetamorphicObject::InvalidOperation();
       }
       m_request >> returnValue;
       return(returnValue);
}
```

Figure 5-9 Example of C++ Forwarder servant for new version.

Individual instances of `CheckingAccount`, existing on one or more active `CheckingAccount` servers, are selected and versioned to a `PremiumAccount` on one or more new `PremiumAccount` Servers. The C++ Implementer servant is replaced with the Forwarder servant on the `CheckingAccount` servers for these instances. Because the etherealization and incarnation of a servant object is internal to the CORBA object, it does not change the object from the client's perspective. When clients invoke the withdrawal operation on a versioned `CheckingAccount` using existing `CheckingAccount` object references, the `CheckingAccount` Forwarder servant receives the request, creates a matching DII request and sends it to the new version of the object on a `PremiumAccount` server. The use of the DII allows the Forwarder to invoke the operation on an object without knowing its interface. This is essential since the `PremiumAccount` interface was defined after the `CheckingAccount` Forwarder was compiled. The amount is withdrawn from the `PremiumAccount`, a service

charge is applied, the balance is saved to a persistent store, and the amount requested is returned to the Forwarder servant on the `CheckingAccount` server. The Forwarder returns the value to the calling client without any other action. When the client sends a request for the balance it will travel the same path through the `CheckingAccount` Forwarder to the `PremiumAccount` Implementer. The value of the balance returned to the client will be equal to the initial balance minus the withdrawal amount and the service charge.

Background

The requirement to dynamically reconfigure distributed applications, especially real-time or mission-critical applications such as Telecommunication Management Networks, has led to the development of a Dynamic Versioning Service for CORBA. This service employs the Forwarder pattern to redirect operation invocations to new versions of Metamorphic Objects. These objects can be reincarnated on new server processes with new IDL interface and implementation class definitions. This allows distributed applications to be incrementally evolved over time without ever shutting down an existing server process or invalidating an existing object reference. A prototype service was developed using Orbix®2.3 for C++ and Solaris. Orbix provides the base-class for the Loader used to load the Forwarder servant. A similar mechanism would be required for a different ORB.

5.4 Design of the Dynamic Versioning Service

The Dynamic Versioning Service is comprised of three main components: (1) the server processes, (2) the persistent store, and (3) a service for configuration management of versions of interfaces and implementations. A detailed design for the first two components, essential to the versioning of metamorphic objects, is presented here. A service to manage the configuration of multiple versions of metamorphic object in a

commercial deployment is beyond the scope of this thesis. As such, the operation of a Version Manager Service is only discussed briefly.

5.4.1 The MetamorphicObject Base-Class

The basis of the Dynamic Versioning Service is the base class MetamorphicObject from which all metamorphic objects must inherit. The definition for MetamorphicObject is comprised of three parts: its IDL Interface and the implementation class definitions for its Implementer servant and its Forwarder servant. The IDL definition is shown in Figure 5-10, and the UML diagram for its implementation classes is shown in Figure 5-11.

```
// IDL in Module DVS, Base class for MetamorphicObjects

interface MetamorphicObject {

        exception InvalidOperation { string reason; };

        attribute string currentInterface;
        attribute string previousInterface;
        attribute Object newVersionRef;
};
```

Figure 5-10 IDL Interface for MetamorphicObject.

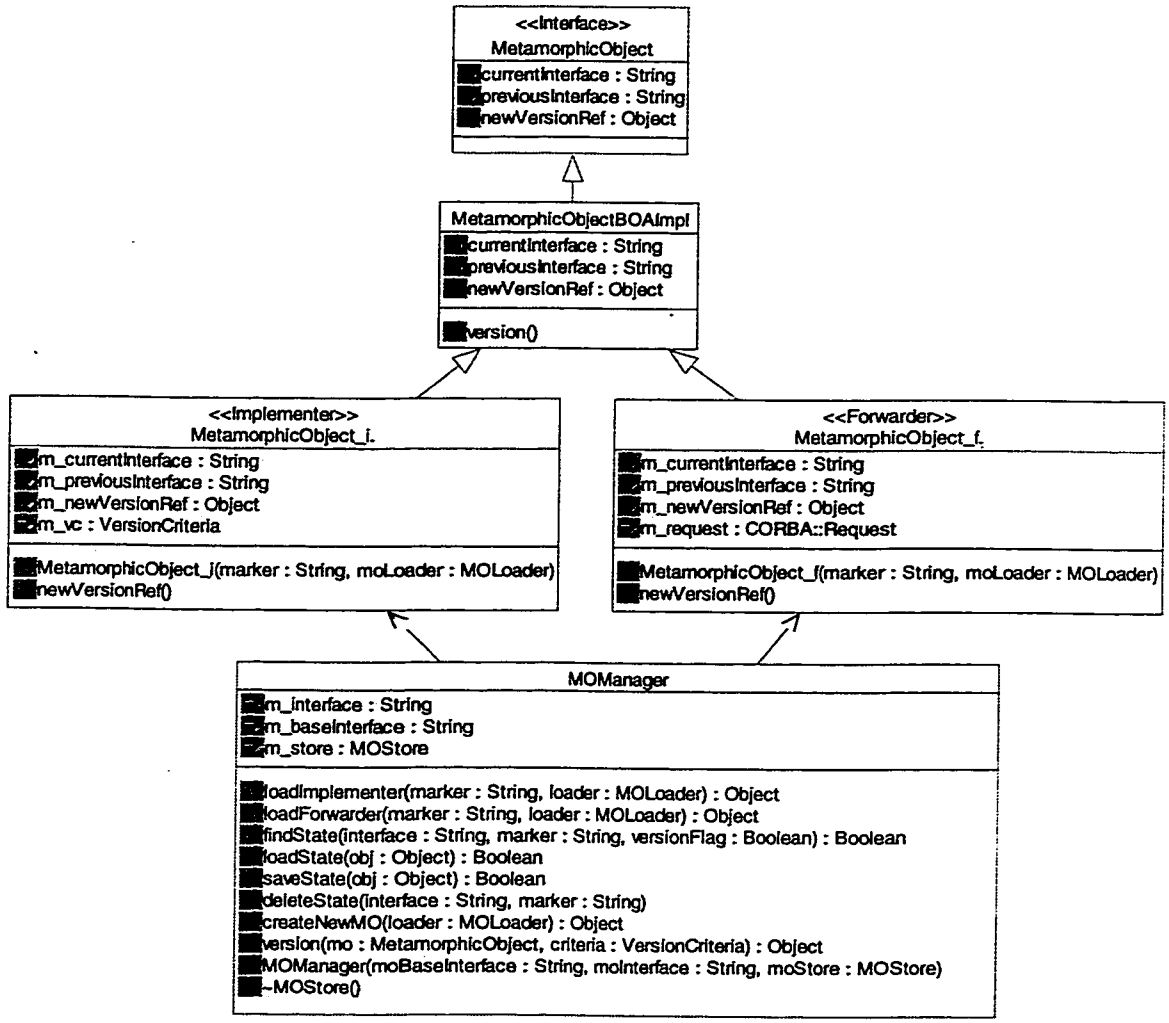

Figure 5-11 Implementation classes for MetamorphicObject

As shown above, the base class of MetamorphicObject provides no operations, only attributes for the current and previous interface names that it supports and, after it is versioned, an object reference to the new version. This reference, of type CORBA::Object, is provided so that client applications can update their references to the new version of the object under application control. The functionality of a MetamorphicObject required for the DVS is actually implemented in the Implementer and Forwarder classes for the application specific subclass, using the Forwarder pattern described earlier.

The class MOManager, shown in Figure 5-11, is internal to the server; no IDL interface is provided. The MOManager provides two basic functions: incarnation of the Implementer or Forwarder servant, and interaction with the MOStore to provide persistence for the state of the Implementer servant. This class must be sub-classed for every sub-class of MetamorphicObject and the following operations must be over-ridden to include the persistent attributes of the application object: `loadState()`, `saveState()`, and `version()`. The MOManager collaborates with MOStore by passing a sequence of the persistent attributes as an Attribute Named-Value List as defined below.

```
// IDL in module DVS

struct AttributeNV {            // Struct of Attribute Named-Values
        string  name;           // the IDL name of the attribute
        any     value;          // the value, TypeCode of the attribute
};

// List of Attribute NamedValues
typedef sequence<AttributeNV> ANVList;
```

Figure 5-12 IDL interface for an Attribute Named-Value List.

The modifications to MOManager for each MetamorphicObject subclass involve adding each persistent attribute to the ANVList in the three operations to be over-ridden. The other changes require the implementation class name for the Implementer and Forwarder class to be changed in all operations. In C++, this can be done by changing a define statement in the MOManager header file. The signature of the MOManager class is not changed.

The MOManager was created to interact with the MOStore developed for the DVS. If a different persistence mechanism is used, such as an Object-Oriented Database Management System (OODBMS), the MOManager could be replaced with an interface

to the OODBMS. The OODBMS would then be responsible for incarnating servant objects and providing persistence for the state of the Implementer.

5.4.2 Servers for Metamorphic Objects

Metamorphic objects are incarnated by a server process. Each server resides on an ORB on a host machine in a network. Each ORB is associated with an Interface Repository and Implementation Repository. Each host may implement a separate IFR and IR or may share them with other hosts. The executable code for each server process is registered with the IR for the host machine by a symbolic name. This allows server processes to be started manually or automatically by the ORB. The symbolic name of the server is used as part of the Interoperable Object Reference used to reference server objects. The executable code for a DVS server typically includes the following:

1. Base-class implementation for MetamorphicObject including the Implementer and Forwarder implementation servants.
2. Application specific sub-classes of MetamorphicObject.
3. Base-class for metamorphic object factories (MOFactory) or any application specific sub-class of MOFactory.
4. Metamorphic object Loader class (MOLoader) for incarnating a metamorphic object implementation.
5. Metamorphic object manager base-class (MOManager) or application specific subclass.
6. Implementation class of the metamorphic object store (MOStore) to provide persistence for metamorphic objects. This is optional as the MOStore may be implemented as a separate server to serve multiple MO servers.

The server process must create one instance of MOLoader, MOStore, an application specific subclass of MOManager, and an application specific subclass of MOFactory. The server may contain other components such as filters, references to Naming Contexts

in a Naming Service, etc. The creation, destruction, and versioning of instances of the MetamorphicObject subclass is performed by the MOFactory in conjunction with the MOManager. Only the application specific subclasses of MOFactory and MetamorphicObject provide an IDL interface through which client applications may access these objects. Only the resources allocated to the process by the host limit the number of instances of the MetamorphicObject subclass that can exist on the server. A typical MetamorphicObject server is shown in Figure 5-13, its UML diagram is shown in Figure 5-14.

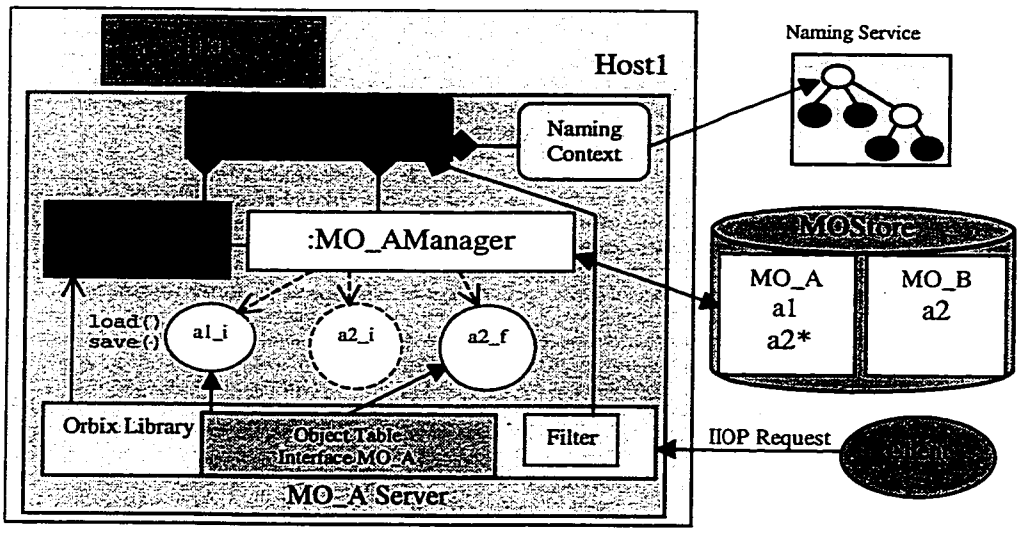

Figure 5-13 Typical Components of a Server for the DVS.

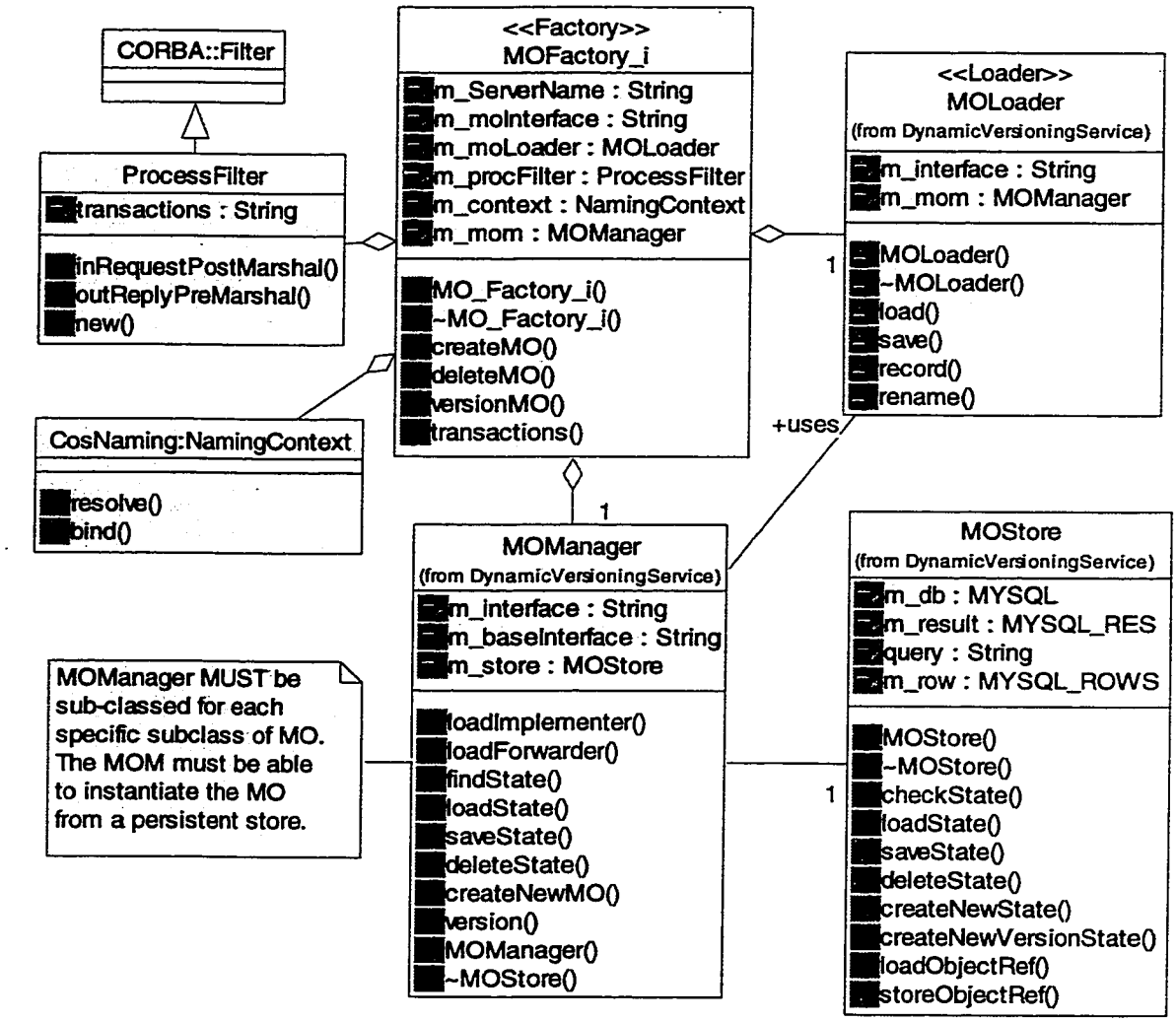

Figure 5-14 UML Class diagram for a MetamorphicObject server.

The system Configuration Manager application must know the name and location of all servers in the network and the type and version of metamorphic object that they support. The Configuration Manager must be able to obtain a reference to the MOFactory on each server, possibly using the Naming Service, in order to create, destroy, and version a metamorphic object. The functions of the Configuration Manager in the versioning process are discussed in section 5.5.

5.4.3 A Metamorphic Object Store

The concept of persistence is an important part of the DVS. The persistent store for metamorphic objects must provide the basic functionality of the Persistent Object Service described in Chapter 3. A Relational Database Management System (RDBMS), Object Oriented Database Management System (OODBMS), or file system may be used to implement the persistent store. A critical requirement of the persistent store is that it must be able to accept and implement a new schema dynamically, without loss of access to the persistent state of any MO. Another requirement is that the persistent state of any class of MO is accessible to any server in the domain. The persistent store must also provide a mechanism for translating the persistent state of a MO from one schema to another (one class to another). This typically consists of operations to copy and modify stored values from one location in the data store to another. A decision was made to use a relational database for the design of the MOStore for the prototype Dynamic Versioning Service. A RDBMS offered the desired combination of ease of implementation, flexibility, dynamic creation of new schema, and the ability to copy attribute values from one schema (table) to another. The MOStore provides a generic IDL interface that could also be implemented using an OODBMS or simple file system. This allows the use of a three-tier architecture for the MOStore as shown in Figure 5-15.

The MOStore provides persistence for all metamorphic objects by saving their attribute values in a shared relational database. The MOManager on each server communicates with this relational database through an instance of MOStore. This instance may reside on each server or may be incarnated on a separate MOStore server using CORBA IIOP to communicate with the MOManagers. Each instance of MOStore acts as a client to the relational database. MOStore provides an IDL Interface, shown in Figure 5-16, through which the MOManager uses the MOStore to provide persistence for each metamorphic object.

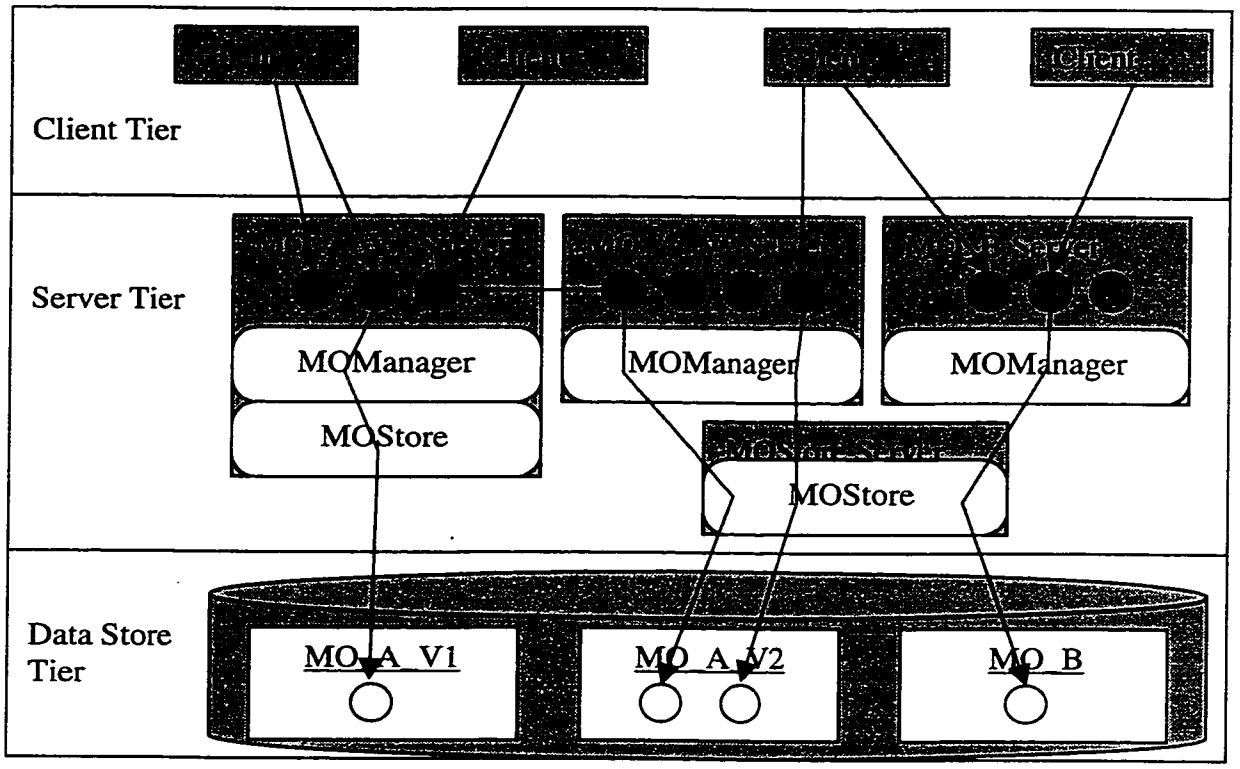

Figure 5-15 Three-Tier Architecture for MOStore.

The MOStore implementation class does not contain the definition of any metamorphic object subclasses. It determines which schema (table) to use to store an object's attributes by the symbolic name for its IDL interface. This is the fully scoped IDL interface name including the name of the module in which it is defined. The fully scoped IDL interface name is used for the relational database table that stores the state for all instances of that interface, regardless of which server they reside on. The attributes themselves are passed between the MOManager and MOStore in an ANVList as a set of symbol-value pairs. The MOStore maps an attribute name to a corresponding table column name. The data type of the column corresponds to the type of the attribute as defined in the schema. The value of the attribute is saved in the corresponding table column for the row created for that instance of MO. Each row uses the object's marker as the unique key to the table. The object's marker is assigned when it is created and must be unique and constant for each metamorphic object, even after it has been versioned.

The value of each attribute in the ANVList is passed as the CORBA type Any. An Any contains the value and CORBA type code of the attribute. The MOStore uses the type code to verify the attribute type and to determine how to store it in the relational database. Because type checking of attribute values is performed at run-time, the implementation of the MOStore is independent of any metamorphic object class.

```
// IDL in Module DVS, MOStore for MetamorphicObjects
interface MOStore {
      boolean checkState(in string moInterface,
                         in string moMarker,
                         inout boolean versionFlag);

      boolean loadState(in string moInterface,
                        in string moMarker,
                        inout ANVList attrList);

      boolean saveState(in string moInterface,
                        in string moMarker,
                        inout ANVList attrList);

      boolean deleteState(in string moInterface,
                          in string moMarker);

      string createNewState(in string moInterface);

      boolean createNewVersionState(in string newInterface,
                                    in string moMarker);

      Object loadObjectRef(in string moInterface,
                           in string moMarker,
                           in string objAttrName);

      boolean storeObjectRef(in string moInterface,
                             in string moMarker,
                             in string objAttrName,
                             in Object objRef);
};
```

Figure 5-16 IDL Interface for MOStore.

The MOStore is also responsible for loading the saved attribute values of a metamorphic object into the Implementer servant when it is created. All metamorphic objects residing on a server may be incarnated when the server starts up or may be incarnated on demand when first referenced by a client application. In the latter case, the MOLoader is called by

the ORB to load the object. The MOLoader requests the MOManager to incarnate the appropriate servant, either the Implementer or Forwarder, and load its persistent state from the MOStore. Loading of an Implementer's state is only required if the server process terminates or when a new version of a metamorphic object is first accessed. Thus, the MOStore provides persistence for fault-tolerance and a means to transfer the state of a metamorphic object from one version to another. The operation of the MOStore in the versioning process is described in the following section.

5.5 Versioning of Metamorphic Objects

Versioning of metamorphic objects is performed by a Configuration Manager application. The Configuration Manager is a client application with the functions and privileges to reconfigure a distributed CORBA application implemented with the DVS. The Configuration Manager is responsible for deploying new servers, registering them with the appropriate IFR and IR, and registering the versioning criteria with the Version Manager Service (VMS). In a complete implementation of the DVS, the VMS would be responsible for maintaining a record of the configuration of all servers and all versioning criteria. The Configuration Manager would use the VMS to select the server and versioning criteria required for transforming a specific metamorphic object to a new version. The roles of a Configuration Manager application are illustrated in the UML Use Case diagram shown in Figure 5-17. The functions of the VMS and the Configuration Manager roles for deploying servers are performed manually in the DVS prototype. This section describes the process of versioning a specific instance of metamorphic object from an initial interface and implementation to a new interface and implementation.

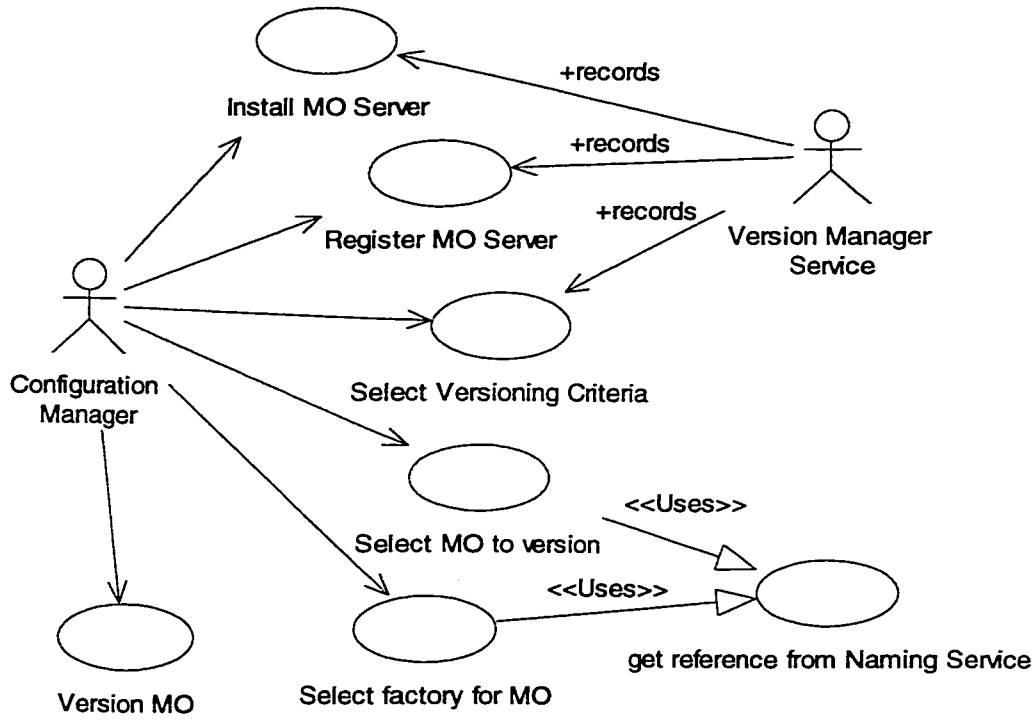

Figure 5-17 Use Case diagram for the Configuration Manager in the DVS.

5.5.1 Selecting a Metamorphic Object to Version

Metamorphic Objects are versioned using the Configuration Manager application. The Configuration Manager must first select the specific instances of the MetamorphicObject subclass that are to be versioned. These metamorphic objects are typically registered in the CORBA Naming Service by a symbolic name. The Configuration Manager can obtain references to the selected objects from the Naming Service.

The Configuration Manager must now determine which server currently incarnates the selected metamorphic object. This requires that the Version Manager Service maintain a directory of server objects. The Configuration Manager requires knowledge of the MO's server because it must invoke the `versionMO()` operation on the MOFactory for the

server. The IDL definition for MOFactory is shown in Figure 5-18. A decision was made to implement the `versionMO()` operation in MOFactory instead of MetamorphicObject. There were two reasons. First, the CORBA Object Model specifies that client applications should not be required to have knowledge of a CORBA object's physical location. Since the versioning process involves moving the incarnation of the MO from one server process to another, adding `versionMO()` to the MetamorphicObject class would violate the CORBA Object Model. The CORBA Object Model also specifies that object Creation and Destruction should be the responsibility of a Factory object [8] on each server process [35]. Since versioning of an object involves the same knowledge of location as Creation and Destruction, versioning should also be a responsibility of the Factory.

```
// IDL in module DVS,  Base class for MetamorphicObjects Factories
interface MOFactory {

        exception CanNotCreate { string reason; };
        exception CanNotDelete { string reason; };
        exception CanNotVersion { string reason; };
        exception AlreadyVersioned { Object newVersionRef; };

        attribute string moInterface;   // The MO subclass IDL Interface

        MetamorphicObject createMO()
                raises (CanNotCreate);

        void deleteMO(in MetamorphicObject mo)
                raises (CanNotDelete);

        void versionMO(in MetamorphicObject mo, in VersionCriteria vc)
                raises (AlreadyVersioned, CanNotVersion);

        string transactions();          // used for monitoring
};
```

Figure 5-18 IDL interface for MOFactory.

Once the Configuration Manager has obtained references to the selected metamorphic object and the Factory for the server process on which it is incarnated, it must select the new version for the MO. As described in Chapter 4, the new version consists of a new

IDL interface, implementation class, and server process. The Configuration Manager would select the version criteria to map persistent attributes from the initial version definition to the new one. In a complete DVS, the Version Manager Service, defined in Figure 5-19, would provide this information. Once the Configuration Manager has the VersionCriteria for the selected metamorphic object, it would then invoke `versionMO()` on the MOFactory as defined in Figure 5-18, or the corresponding MOFactory subclass operation. The VersionCriteria and the reference to the selected MO are passed to the MOFactory to begin the process.

```
// IDL in module DVS //***** Dynamic Versioning Service *****//

// Versioning Criteria for MetamorphicObjects

   typedef sequence<string> Compatibles;

   struct VersionCriteria {        // Criteria for versioning a MetamorphicObject
         string sourceInterface;   // fully scoped IDL interface of source MO
         string destInterface;     // fully scoped IDL interface of destination MO
         string destServer;        // destination server implementation
         string destHost;          // host name of the destination server
         string destIFRHost;       // IFR host name for the server (== host)
         Compatibles attrList;     // list of compatible attributes in destination
         Compatibles opList;       // list of compatible operations in destination
   };

   typedef sequence<VersionCriteria> VCList;        // list of registered criteria

// One VersionManager Service is required in the domain

interface VersionManager {

         // register a new criteria with the VersionManager
         void  setCriteria(in VersionCriteria vc);

         // list all registered criteria registered with the VersionManager
         VCList listCriteria ();

         // get a specific criteria, may be more than one implementation
         VCList getCriteria(in string sourceInterface,
                            in string destInterface);
};
```

Figure 5-19 VersionCriteria Provided by Version Manager Service.

5.5.2 The Versioning Process

The process of versioning a metamorphic object to a new implementation and interface is performed by the MOManager on the server process that initially incarnates the MO. The MOManager is responsible for mapping the attributes of its corresponding MO class to a new schema using the version criteria provided by the Configuration Manager when invoking `versionMO()` on MOFactory. The MOManager creates an ANVList that contains only the attributes that are common between the initial version and the new version of the MO. This ANVList is provided to the MOStore along with the interface name of the new version. The MOStore then saves these attribute values in the data store for the new interface. The versioning operation is illustrated in the following UML Sequence diagram.

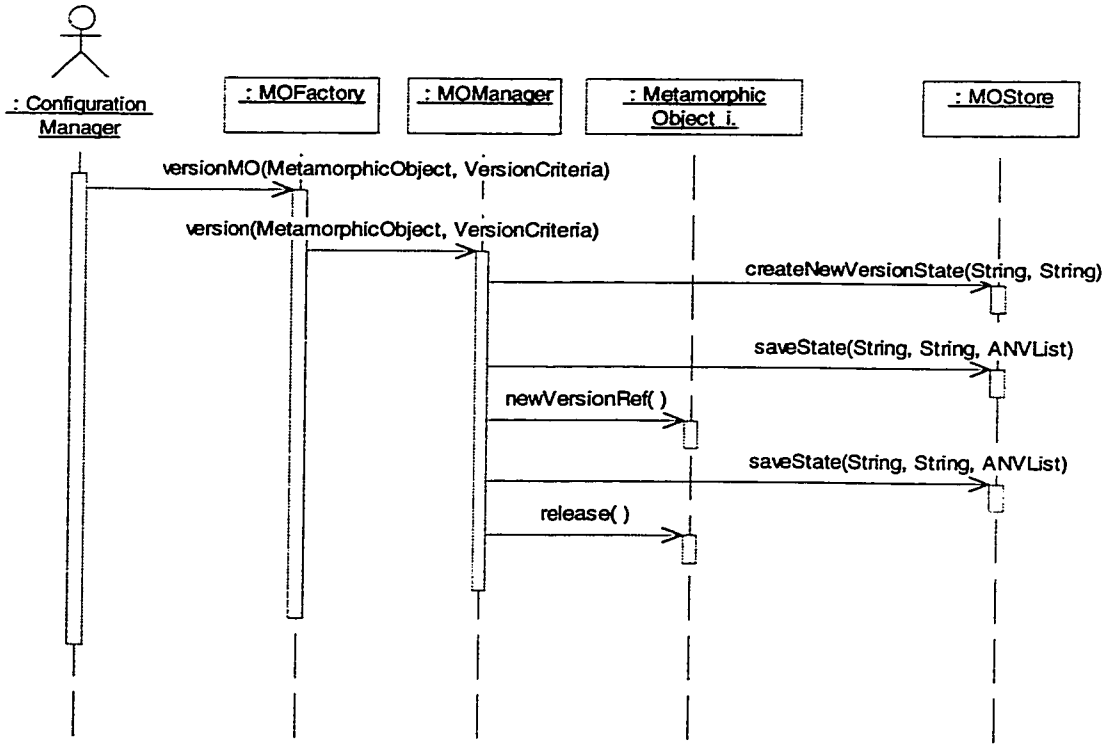

Figure 5-20 UML Sequence diagram of the versioning operation.

The `version()` operation causes the following steps to be performed by the MOManager:

1. The persistent state of the MO is marked to indicate that the MO has been versioned and a reference to the new version of a MO is stored. The `version()` operation passes string parameters that indicate the new name of the IDL interface, the new server name, the host on which the server is to run, and the location of the IFR that holds the new Interface definition. These are used by a `string_to_object()` operation [7] to obtain a CORBA object reference to the new MO when the Forwarder servant is incarnated. Once marked as versioned, the attribute states saved in the old version of the MO are no longer used to incarnate a servant.
2. A new data store entry is created using the database schema for the new version to store the persistent state of the new version of the MO. In a complete DVS, the MOStore would be able to generate the schema for the database from the IDL definition of the new version of MO. In the prototype, this schema is entered manually.
3. A mapping function is installed for the new version of the MO class. This mapping function translates the *n*-tuple of values corresponding to the *n* attributes defined in the IDL Interface considered relevant to the object's state to the attributes defined in the new IDL interface. This also includes the mapping of any object references held as attributes of the MO. The developer provides this mapping function when the new version of the MO class is defined.
4. A server supporting the new version of the MO must be launched. This server contains a MOManager that can instantiate the new Implementer and Forwarder servants for the MO. The MOManager can also access the database that maintains the persistent state for the version of a MO.
5. When an instance of a MO is versioned, the MOManager invokes the mapping function to set the persistent state of the new version of the MO. When the Forwarder for the old version invokes a DII request on the new version of the MO, the new MO is incarnated with an Implementer servant that is linked to its

new persistent state by the new MOManager. This new MO and its persistent state is now the active version of the MO even though it can be (indirectly) accessed using object references to the old version.

During the versioning process, the persistent state of the MO is modified to indicate that it has been versioned and to include a reference to the new version. The Implementer servant is then deleted from memory on the initial server process. This temporarily leaves the metamorphic object with no incarnated servant on either the initial version server or on the new version server. To complete the versioning of a metamorphic object, the Forwarder servant must be incarnated on the initial server process and the new Implementer servant, with the versioned persistent state, must be incarnated on the server process for the new version.

On the server for the initial version of the MO, the Implementer servant for the specific instance of MO that was versioned is replaced with the Forwarder servant when the next operation is invoked on the MO using an existing object reference. The loading of the Forwarder servant is performed using the MOLoader. A Loader is an object in Orbix 2.3 that services objects faults on the ORB [7]. When a server receives an invocation for an object that is not currently in memory, an object fault occurs and the server's Loader(s) are called to try to instantiate a servant object for the MO. The MOLoader uses the object marker and interface name of the desired MO (derived from the object reference by the ORB) to query the MOManager for the persistent state of the object. If the MOManager cannot locate the desired object, the load fails and returns a null object reference to the ORB. This causes the ORB to return an OBJECT_NOT_EXIST exception to the calling client. If the MOManager locates the persistent state for an object with a matching marker and interface, a servant is instantiated and initialized to the state in the database. Once a metamorphic object has been versioned, the persistent state for the old version is marked as being versioned. The state of a versioned MO consists of the three attributes defined for the MetamorphicObject base-class: the previous interface, current interface, and the object reference to the new version. The servant is returned from the MOLoader to the ORB and the operation is then invoked on it. The loading of

the Forwarder servant, shown in Figure 5-21, is transparent to the client as it is only performed once.

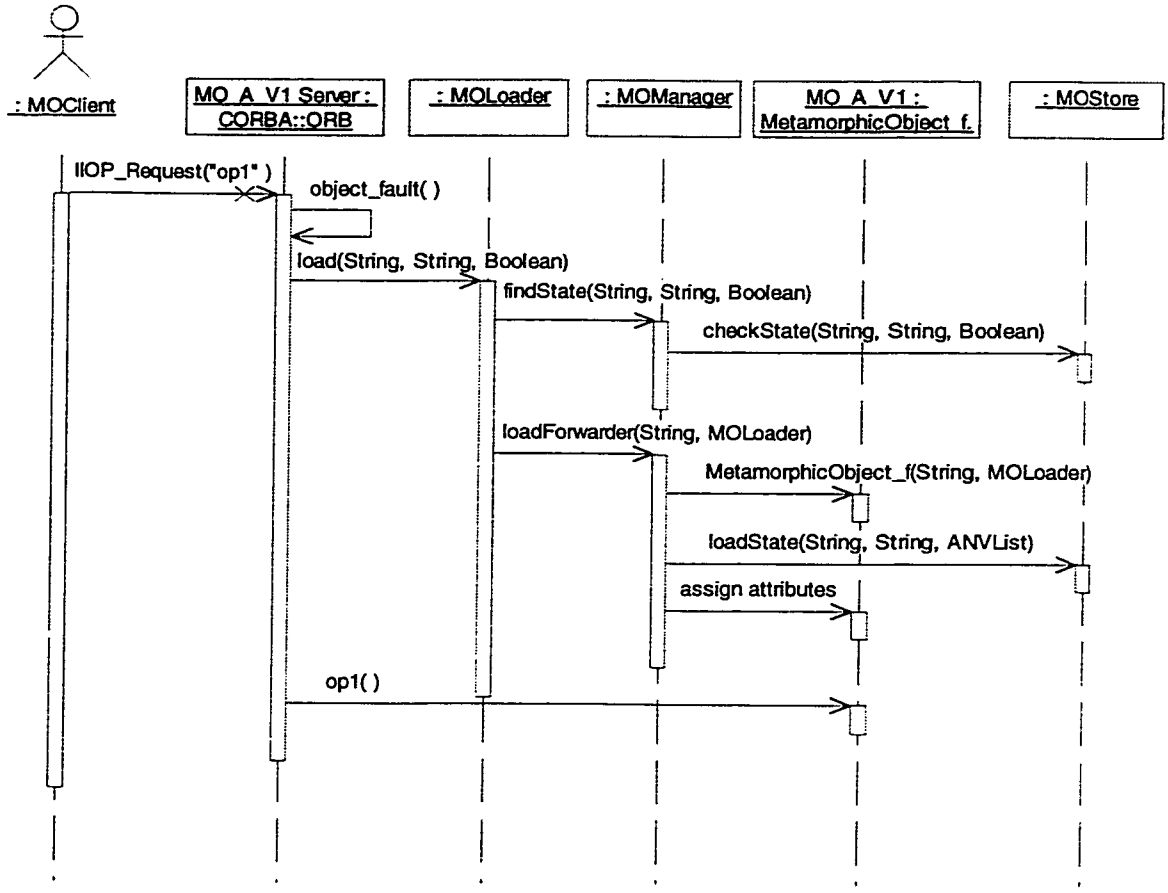

Figure 5-21 Loading of the Forwarder servant for the old version of MO.

When the server receives the next operation on the MO from a client using an old object reference, an object fault occurs and the loader is invoked. When the loader queries the MOManager for the object, the MOManager detects that the object has been versioned and instantiates a Forwarder servant for the MO. This Forwarder is initialized with the object reference to the new MO on another server. The operation on the MO is then invoked on the Forwarder servant. The matching Forwarder operation constructs the DII request using the parameters passed from the client and invokes the request on the new version of the MO. Because no servant exists for the new version of the MO, the ORB for

the new version server raises an object fault. The MOLoader is called to load the Implementer servant. If successful, the DII request operation from the initial version is invoked on the new version Implementer. The Implementer servant for the new version processes the request and returns any result to the Forwarder servant on the initial version server, as shown in the sequence diagram in Figure 5-22. The Forwarder then returns this result to the client. The Implementer servant for the new version of MO is only loaded once, when it is first accessed.

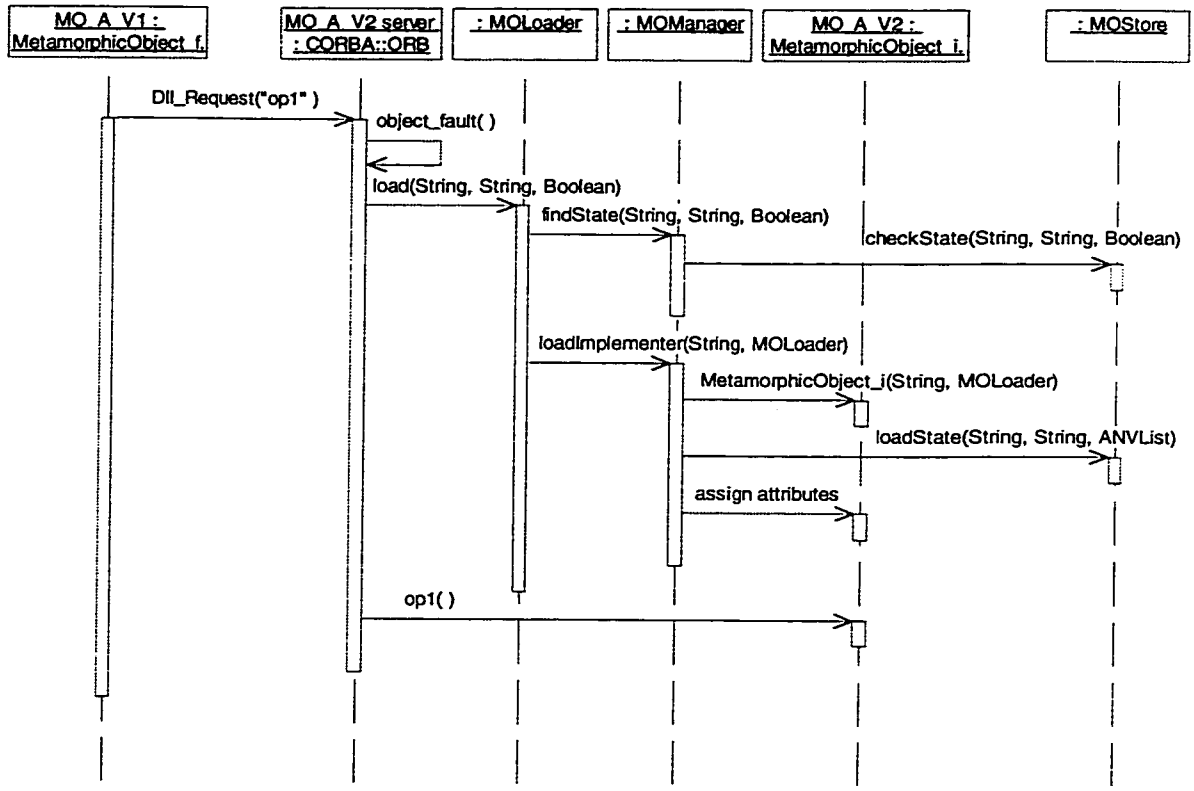

Figure 5-22 Loading of the Implementer servant for the new version of MO.

The loading of the Forwarder servant for the old version and the Implementer servant for the new version is initiated by the first client operation invocation after the MO has been versioned. The Forwarder and new Implementer could have been loaded explicitly by the MOManager that performed the versioning operation. However, since the MOLoader was already implemented to provide persistence for metamorphic objects, the MOLoader

provided an existing mechanism for installing the Forwarder and Implementer servants on their respective servers. Thus, the client process must experience a one-time delay when first accessing a newly versioned object. Had the MOManager loaded the Forwarder and Implementer, the client process would have been blocked for about the same period of time. In either case, once the Forwarder and Implementer are installed, the delay added to the invocation time of a forwarded operation, as illustrated in Figure 5-3, is equivalent to one additional IIOP message being sent.

The behavior and performance of the Dynamic Versioning Service prototype described in this chapter are evaluated using a test application in Chapter 6. The pros and cons of the DVS prototype presented here are discussed in Chapter 7, along with recommendations for further development and applications of the DVS.

6 Evaluation of the Dynamic Versioning Service

A test application was implemented to evaluate the Dynamic Versioning Service for CORBA presented in Chapter 5. This involved four activities: implementation of the Dynamic Versioning Service (DVS), implementation of a suitable DVS application, functional tests to verify the operation of the DVS and the test application, and performance tests to determine the effects of versioning of Metamorphic Objects. The tests focus on evaluating the Forwarder pattern, not the services required by a complete DVS, such as the Version Manager Service. This chapter describes the implementation and evaluation of a Banking application using the DVS.

6.1 Implementation of the Dynamic Versioning Service

The core of the Dynamic Versioning Service (DVS) was implemented in C++ using Orbix 2.3 [41, 42] for Solaris 2.5. This included the MOLoader class, MOFactory class, the base class for MetamorphicObject, and the base class for MOManager. It also included an implementation of the Metamorphic Object Store (MOStore) using a MySQL [37] relational database. The implementation of the Version Manager Service is beyond the scope of this thesis so the versioning criterion was specified manually for each test object. However, the components that were implemented are sufficient to construct and deploy a test application using the Dynamic Version Service. This included the development of the MOLoader class and MOStore application as complete and re-usable components for any DVS application. The base classes for MetamorphicObject and MOManager are also complete but must be sub-classed for each metamorphic object class. Likewise, the MOFactory should be sub-classed for servers to provide an application specific IDL interface, but could be used directly for the creation, deletion, and versioning of any MetamorphicObject subclass.

The most complicated of these components are the MOManager and MOStore. These two components must interact for MO servant creation, deletion, versioning, and persistence. The MOManager is sub-classed and its operations are over-ridden to process the specific classes and attributes of its related MetamorphicObject subclass. This makes each MOManager subclass specific to a MetamorphicObject subclass. The MOStore, however, must be generic and able to provide persistence to any MetamorphicObject subclass. Also, the MOStore must be able to copy the persistent state of a specific MO instance from one class definition and implementation to another, as directed by the MOManager. The persistent attributes of a MetamorphicObject are passed between the MOManager and MOStore as an Attribute NamedValue List (ANVList), as described in Chapter 5. The use of a generic IDL interface and the ANVList allows the MOStore to be implemented as a separate process shared by multiple MO servers, or incorporated as a component of each server providing concurrent database access. The test implementation used the latter approach.

The MOStore requires a mechanism for storing the persistent state of an object according to a class schema. This could be implemented using a file system with each class schema maintained as a separate file and instances of the class as entries in the file [7]. The MOStore could also be implemented using a Relational Database Management System (RDBMS) [5, 6, 7, 36], with class schema mapped to tables, or with a Object-Oriented Database Management System (OODBMS) [7, 47]. The only requirement of the persistent store for MetamorphicObjects in addition to being able to store object attributes is that new class schema can be added without shutting down the entire persistent store. If the persistent store must be shut down to add a new class, all objects in the system would be temporarily unavailable which would defeat the concept of dynamic versioning.

The MOStore was implemented using a relational database, which limited the types of objects and attributes that could be saved. As the focus of this thesis is not database design, a shareware RDBMS, MySQL [37], was chosen for convenience and easy of use. A decision was made to limit the types of attributes that can be made persistent to the basic types shown in Table 6-1. Complex attributes such as array, sequence, and union

are not supported in this implementation because of the complexity involved in mapping these structures to a relational database. An implementation of MOStore using an OODBMS would be able to support all IDL attribute types.

Table 6-1 Attribute Types Supported by MOStore.

IDL	C++ typedefs	C++ native types	CORBA Type Code
`short`	CORBA::Short	short	CORBA::_tc_short
`long`	CORBA::Long	long	CORBA::_tc_long
`unsigned short`	CORBA::UShort	unsigned short	CORBA::_tc_ushort
`unsigned long`	CORBA::ULong	unsigned long	CORBA::_tc_ulong
`float`	CORBA::Float	float	CORBA::_tc_float
`double`	CORBA::Double	double	CORBA::_tc_double
`char`	CORBA::Char	char	CORBA::_tc_char
`boolean`	CORBA::Boolean	bool	CORBA::_tc_boolean
`octet`	CORBA::Octet	unsigned char	CORBA::_tc_octet
`string`	CORBA::String	char*	CORBA::_tc_string
`interface X`	CORBA::Object	pointer	CORBA::_tc_Object

An instance of a MetamorphicObject class is saved as a row in a database table where each column corresponds to each persistent attribute. The field type of the column defines the type of the attribute. This is a common technique for mapping object-oriented models to relational database schema [5, 7, 36]. Each of the attribute types shown in Table 6-1 map to a corresponding field type in SQL with the exception of `interface X` which represents a CORBA Object. Since all MetamorphicObjects must be able to store a pointer to the new version of the MO, object pointers must be made persistent in the MOStore.

Object references are made persistent by creating a separate table in the database for each MO attribute type that is saved as an object pointer. This object attribute table contains the information required to construct a IOR to an object of class CORBA::Object. An

IOR can be used to reference any subclass of CORBA::Object (i.e. any CORBA object) regardless of what server it resides on. The MOStore passes an object attribute in the `ANVList` (described in Chapter 5) as a pointer of type CORBA::Object, widened or narrowed by the MOManager accordingly. The MOStore converts the object reference to a set of strings representing the host, implementation (server), marker, interfaceHost (for the Interface Repository), and interfaceMarker (IDL interface name) of the target object. These strings are obtained by invoking the operations `_host()`, `_implementation()`, `_marker()`, `_interfaceHost()`, and `_interfaceMarker()` on any CORBA object. These strings are sufficient to construct an CORBA IOR using the operation `stringToObject()`[7]. When the MOStore is requested to load an object reference to the new version of the MO or as an MO attribute, the MOStore reads these strings from the appropriate table in the relational database, constructs a CORBA IOR using `stringToObject()`, and returns the object reference in the `ANVList` to the MOManager. The MOManager then assigns the object reference to the MO being loaded.

Since an MO can reference other sub-classes of MO, it would be possible to construct complex objects using aggregation even though each object could only contain basic attribute types and object pointers. This use of aggregation is demonstrated in the DVS Bank application described next. The database schema for the MOStore for the DVS Bank is also presented.

6.2 Implementation of the DVS Bank Application

The test application is based on the premise of a banking system that enables an account manager to version client accounts and loans from one class definition to another. There are two independent sets of MetamorphicObject sub-classes that are accessed by the client application: CheckingAccount with PersonalLoan, and PremiumAccount with LineOfCredit. The CheckingAccount and PersonalLoan application was written with no reference to PremiumAccount and LineOfCredit, and visa versa. This demonstrates the

ability to version an object of an existing class to a new class definition that was unknown when the object was created. The UML class diagram and IDL definition for the DVS CheckingAccount application are shown in Figures 6-1 and 6-2, respectively. The class diagram and IDL definition for the DVS PremiumAccount application are shown in Figures 6-3 and 6-4.

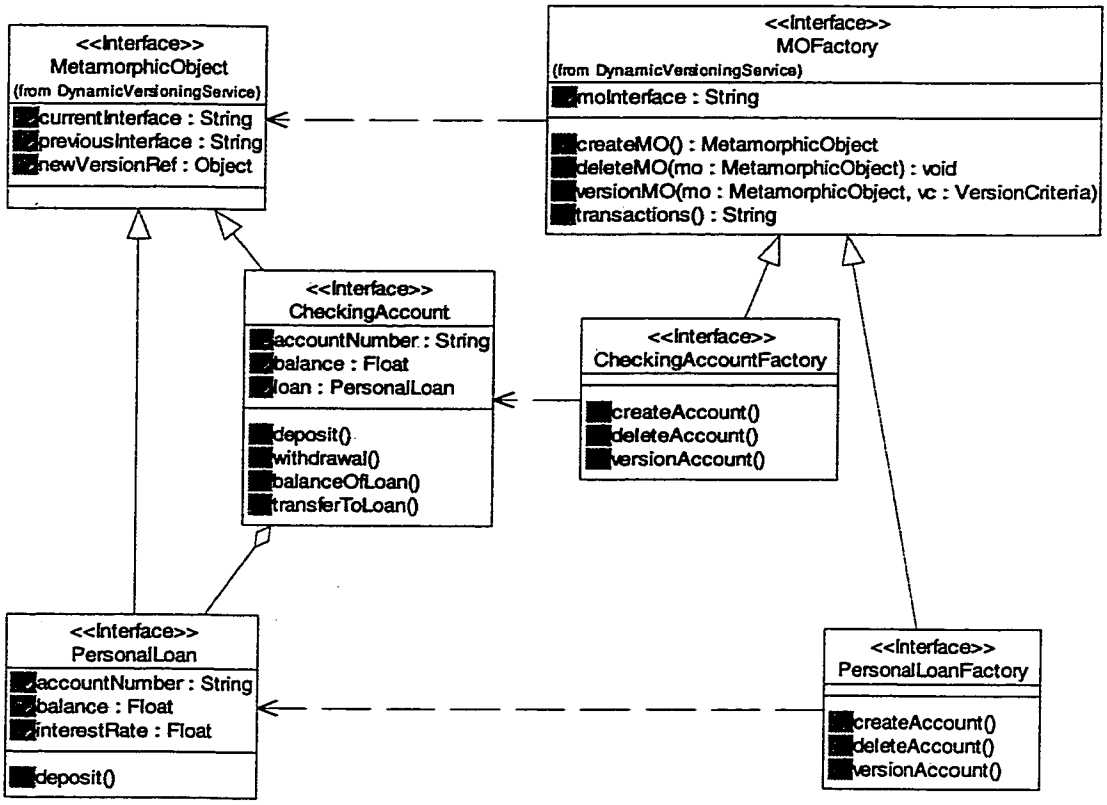

Figure 6-1 Class Diagram for the DVS Bank CheckingAccount Application

```
module DVSCA {     // Dynamic Versioning Service Checking Account

 interface PersonalLoan : DVS::MetamorphicObject {
      exception InvalidAmount { float amount; };

      readonly attribute string accountNumber;
      readonly attribute float balance;
      readonly attribute float interestRate;

      void deposit (in float amount)
            raises (InvalidAmount);
 };

 interface PersonalLoanFactory : DVS::MOFactory {
      PersonalLoan createLoan (in string accountNumber,
                        in float amount,
                        in float interestRate)
            raises (CanNotCreate);
      void deleteLoan (in PersonalLoan loan)
            raises (CanNotDelete);
      void versionLoan(in PersonalLoan loan,
                        in DVS::VersionCriteria vc)
            raises (AlreadyVersioned, CanNotVersion);
  };

 interface CheckingAccount : DVS::MetamorphicObject {
      exception InvalidAmount { float amount; };
      exception LoanUnavailable { string reason; };

      readonly attribute string accountNumber;
      readonly attribute float balance;
            attribute PersonalLoan loan;

      void deposit (in float amount)
            raises (InvalidAmount);
      float withdrawal (in float amount)
            raises (InvalidAmount);
      float balanceOfLoan()
            raises (LoanUnavailable);
      void transferToLoan(in float amount)
            raises (LoanUnavailable, InvalidAmount);
};

 interface CheckingAccountFactory : DVS::MOFactory {
      CheckingAccount createAccount (in string accountNumber)
            raises (CanNotCreate);
      void deleteAccount (in CheckingAccount account)
            raises (CanNotDelete);
      void versionAccount(in CheckingAccount account,
                        in DVS::VersionCriteria vc)
            raises (AlreadyVersioned, CanNotVersion);
  };
};
```

Figure 6-2 IDL Definition for CheckingAccount Application.

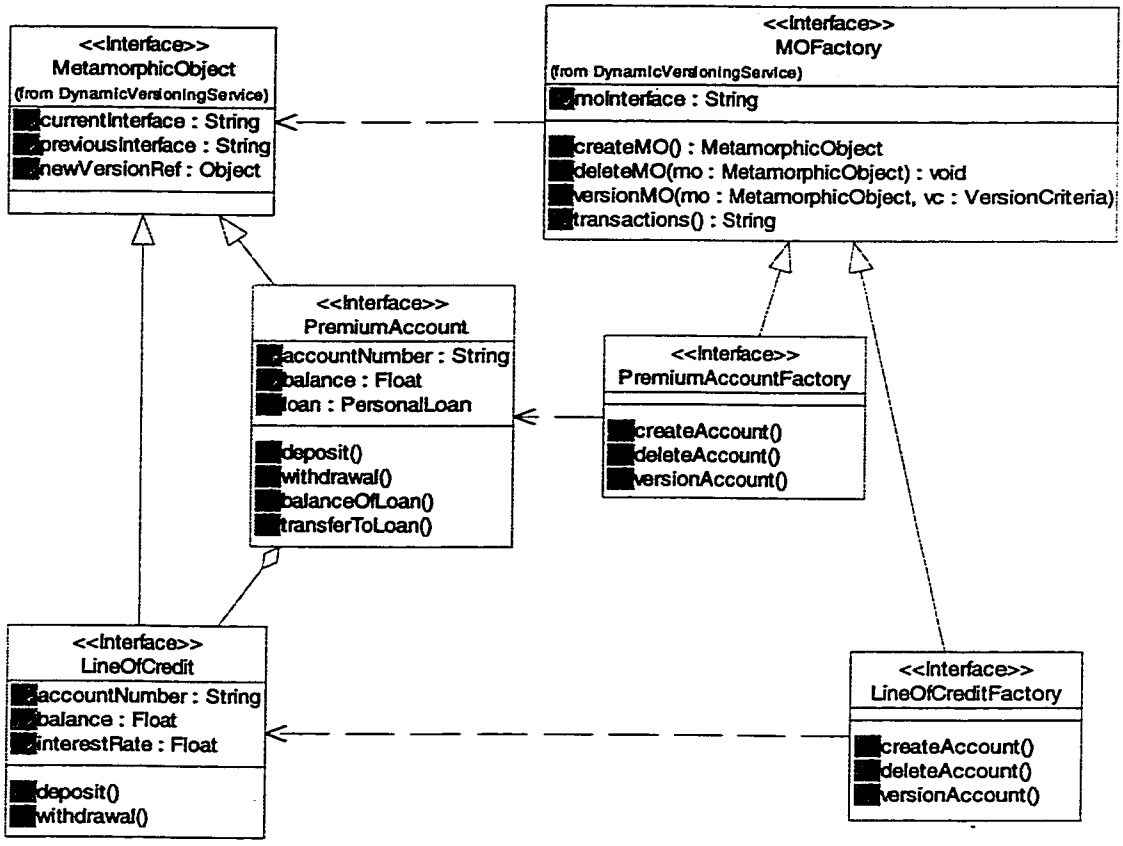

Figure 6-3 Class Diagram for the DVS Bank PremiumAccount Application.

```
module DVSPA {     // Dynamic Versioning Service PremiumAccount
 interface LineOfCredit : DVS::MetamorphicObject {
       exception InvalidAmount { float amount; };
       readonly attribute string accountNumber;
       readonly attribute float balance;
       readonly attribute float interestRate;
       void deposit (in float amount)
             raises (InvalidAmount);
       float withdrawal (in float amount)
             raises (InvalidAmount);
};
 interface LineOfCreditFactory : DVS::MOFactory {
       LineOfCredit createLoan (in string accountNumber,
                           in float amount,
                           in float interestRate)
             raises (CanNotCreate);
       void deleteLoan (in LineOfCredit loan)
             raises (CanNotDelete);
       void versionLoan(in LineOfCredit loan,
                         in DVS::VersionCriteria vc)
             raises (AlreadyVersioned, CanNotVersion);
};
 interface PremiumAccount : DVS::MetamorphicObject {
       exception InvalidAmount { float amount; };
       exception LoanUnavailable { string reason; };
       readonly attribute string accountNumber;
       readonly attribute float balance;
       readonly attribute float interestRate;
       readonly attribute float serviceCharge;
                attribute LineOfCredit loan;
       void deposit (in float amount)
             raises (InvalidAmount);
       float withdrawal (in float amount)
             raises (InvalidAmount);
       float balanceOfLoan()
             raises (LoanUnavailable);
       void transferToLoan (in float amount)
             raises (LoanUnavailable, InvalidAmount);
       void transferFromLoan (in float amount)
             raises (LoanUnavailable, InvalidAmount);
 };
 interface PremiumAccountFactory : DVS::MOFactory {
       PremiumAccount createAccount (in string accountNumber,
                                in float interestRate,
                                in float serviceCharge)
                    raises (CanNotCreate);
       void deleteAccount (in PremiumAccount account)
                    raises (CanNotDelete);
       void versionAccount(in PremiumAccount account,
                           in DVS::VersionCriteria vc)
             raises (AlreadyVersioned, CanNotVersion);
 };
};
```

Figure 6-4 IDL Definition for PremiumAccount Application.

6.2.1 The CheckingAccount and PremiumAccount Applications

The PremiumAccount application is completely independent of the CheckingAccount application. The only commonality between the two applications is that they both inherit from MetamorphicObject and MOFactory. As was discussed in Chapter 4, the criterion for classes to be compatible is determined by the degree of polymorphism that is supported between the classes, not on inheritance. As can be seen by comparing the IDL definitions for the CheckingAccount and PremiumAccount applications in Figures 6-2 and 6-4, PremiumAccount supports all of the operations of CheckingAccount and is therefore fully compatible with it. On the other hand, CheckingAccount does not support the PremiumAccount operation `transferFromLoan()` or the attributes `interestRate` and `serviceCharge`, which makes it only partially compatible with PremiumAccount. Similarily, LineOfCredit is fully compatible with PersonalLoan but PersonalLoan does not support the operation `withdrawal()`, making it only partially compatible with LineOfCredit. Metamorphic objects are required to throw and/or catch an exception when an invalid operation is invoked on an MO that has been versioned. The incompatibility between PremiumAccount and CheckingAccount and between LineOfCredit and PersonalLoan is used to verify that an exception is thrown when the operations `transferFromLoan()` and `withdrawal()` are invoked on instances of PremiumAccount and LineOfCredit after they have been versioned.

6.2.2 MOStore Database Schema

The database schema for the MOStore was produced manually from the IDL definitions for CheckingAccount, PremiumAccount, PersonalLoan, and LineOfCredit. Since the schema for an MO subclass can be produced directly from its IDL definition, there could be added an operation such as `createSchema(in string moInterface)` to the MOStore component of the DVS. Whether the database schema was created manually or

automatically, the result would be the same set of database tables. Figure 6-5 shows the database schema for CheckingAccount.

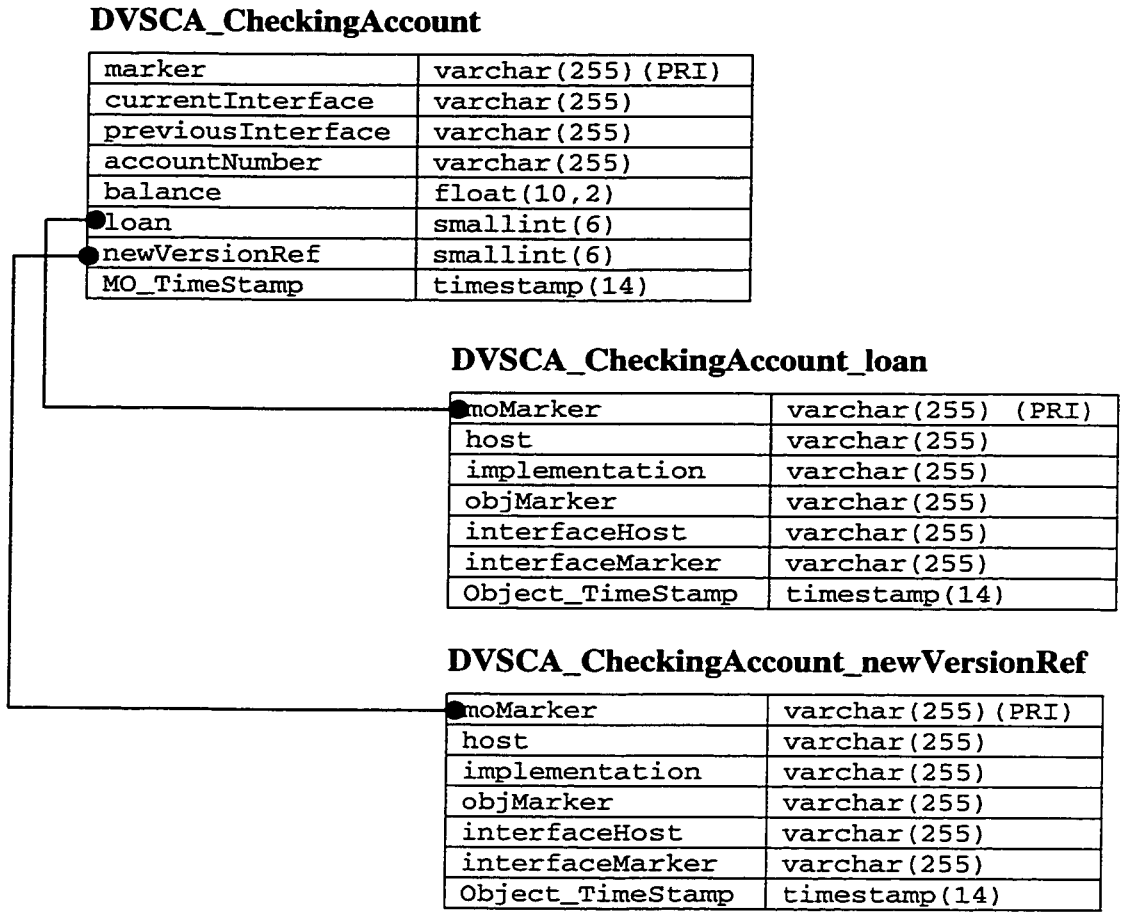

Figure 6-5 MOStore Schema for CheckingAccount.

As can be seen in Figure 6-5, the prime table for CheckingAccount contains columns for the attributes for CheckingAccount and those inherited from MetamorphicObject. Also, the two object attributes, `loan` and `newVersionRef`, are stored as an integer to indicate if an object reference for the attribute exists of not. If these columns are non-zero (representing a boolean true), it indicates to the MOStore that an IOR must be constructed from an entry in a table named by <Interface>_<attribute>. The MOStore uses the marker of the MO as the key to the object reference table. The MOStore then constructs the IOR using the `stringToObject()` operation. Conversely, when storing

an object reference, the MOStore marks the column in the MO class table (i.e. DVSCA_CheckingAccount) for the object attribute name with a value of 1, and then saves the appropriate strings in the matching database table. The TimeStamp fields were included for testing purposes to verify database operations and are not required for normal MOStore schema.

6.2.3 Server Implementations

Four application specific classes must be implemented for each of the four servers required for the Bank application. These include the MOFactory subclass, the Implementer and Forwarder servant classes for the MO subclass (i.e. CheckingAccount), and the corresponding MOManager subclass (i.e. `CAManager`). The MOFactory subclass implementation involves extending the interface for the MO subclass specific factory (i.e. `CheckingAccountFactory`). The implementation of the MOFactory subclass is straightforward. The Implementer servant, Forwarder servant, and Manager classes must be implemented according to the DVS framework.

The Implementer servant is written to implement the desired functionality of the corresponding Interface. For example, `CheckingAccount_i` is written to support the IDL interface CheckingAccount and maintain a `balance` using `deposit()` and `withdrawal()` operations. The Forwarder servant is written to support the same IDL interface as the Implementer. However, instead of carrying out the desired functionality such as adding a deposit amount to a balance, the Forwarder creates a DII request message (`m_request`) and sends it to whatever object it references with its `newVersionRef` pointer. A comparison of the `withdrawal()` operation in the Implementer and Forwarder servants for the implementation of CheckingAccount in C++ using Orbix 2.3 is shown in Figure 6-6.

```
CORBA::Float CheckingAccount_i::withdrawal (CORBA::Float f,
                                CORBA::Environment& IT_env)
                    throw (CORBA::SystemException,
                           DVSCA::CheckingAccount::InvalidAmount){
      float amountReturned;

      if (f < 0.0)
            throw DVSCA::CheckingAccount::InvalidAmount(f);
      if (f > m_balance){
            amountReturned = m_balance;
            m_balance = 0.0;
      }
      else {
            m_balance -= f;
            amountReturned = f;
      }
      this->CORBA::Object::_save();   //MOLoader calls MOStore
      return amountReturned;
}
```

```
CORBA::Float CheckingAccount_f::withdrawal (CORBA::Float f,
                                CORBA::Environment& IT_env)
                    throw (CORBA::SystemException,
                           DVS::MetamorphicObject::InvalidOperation)
      CORBA::Float returnValue;
      m_request.reset();
      m_request.setTarget(m_newVersionRef);
      m_request.setOperation("withdrawal");
      m_request.set_return_type(CORBA::_tc_float);
      m_request << CORBA::inMode << f;
      try {
            m_request.invoke();
      } catch (...){
            throw DVS::MetamorphicObject::InvalidOperation(
                        CORBA::string_dup("No Longer Supported"));
      }
      m_request >> returnValue;
      return(returnValue);
}
```

Figure 6-6 Implementer & Forwarder Servants for CheckingAccount withdrawal().

Because the object reference `m_newVersionRef` for the Forwarder servant is of type `CORBA::Object`, it can be invoked on any CORBA object, even if its class was defined after the servant was compiled. If the target object does not support the operation in the DII request, an InvalidOperation exception is returned to the Forwarder servant by the ORB, which checks the Interface Repository of the target object before it invokes the DII request [7]. The Forwarder catches this exception and returns a MetamorphicObject

exception indicating that the operation is no longer valid. The use of DII enables the Forwarder to make an ad hoc polymorphic operation invocation on an instance of a class that was unknown at compile time. This is the basis of Dynamic Versioning in CORBA.

The subclass of MOManager to be implemented for the server is dependent on the Implementer servant because it must access its private members to provide persistence using MOStore. This requires that the MOManager subclass (i.e. `CAManager` for `CheckingAccount_i`) be declared a friend class in C++ [9]. Since the Forwarder servant has no state member to change, it is requires no additional persistent storage. The MOManager subclass must have its load, save, and version operations over-ridden for the specific Implementer Servant. The template is included in the MOManager base-class for each attribute type supported by the MOStore to aid in development of these operations.

The components for each DVS Bank class, CheckingAccount, PersonalLoan, PremiumAccount, and LineOfCredit, are combined with the DVS components MOLoader and MOStore to create each server according to the template described in Chapter 5. These four servers were launched as individual processes for the functional and performance testing of the DVS Bank application. The UML diagram for the CheckingAccount server is shown in Figure 6-7.

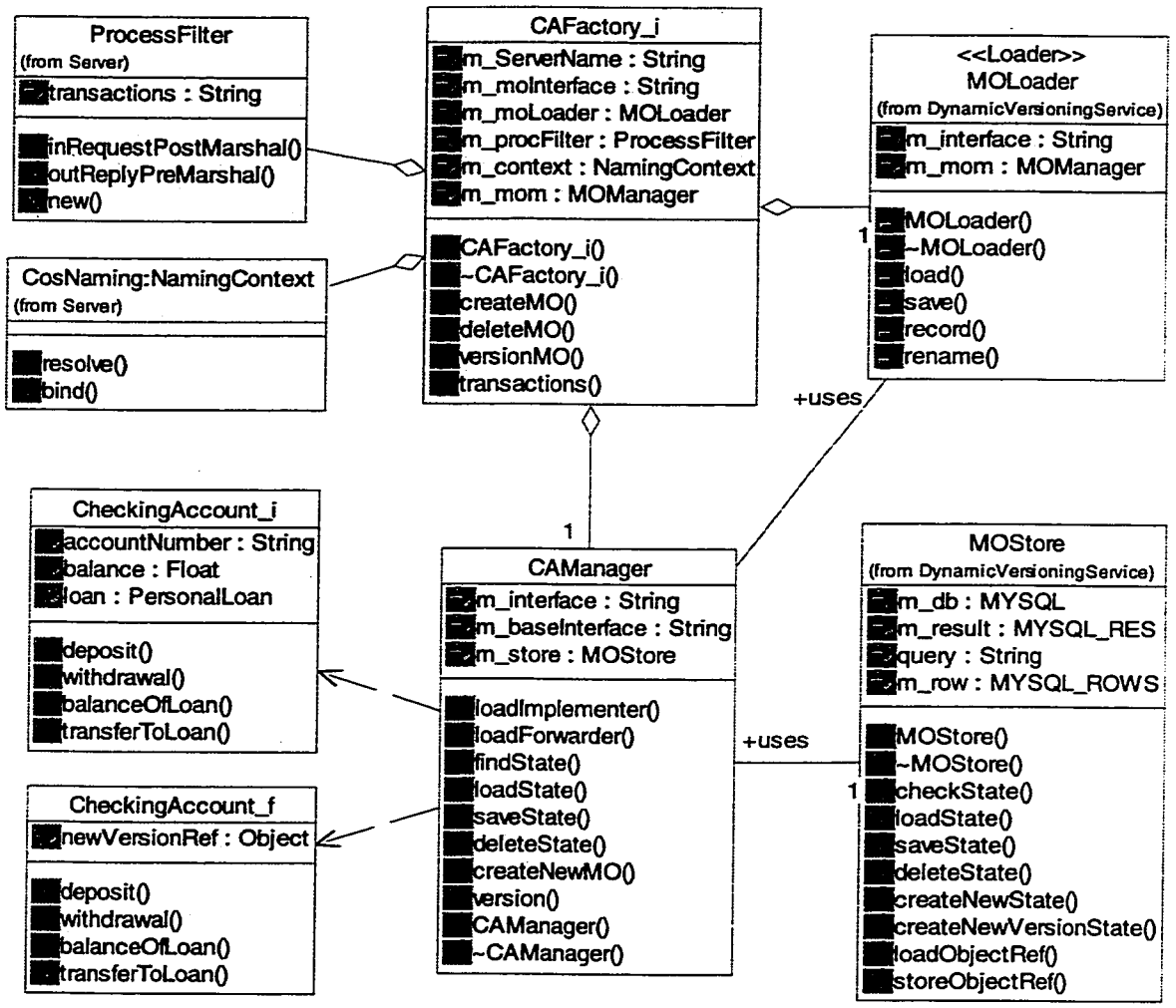

Figure 6-7 UML Class diagram for CheckingAccount server.

6.3 Functional Testing of the DVS Bank Application

Two sets of functional tests were performed on the DVS Bank application: one set for the CheckingAccount operations and one set for the PremiumAccount operations. The CheckingAccount tests were performed using a special client manager application that could perform all CheckingAccount operations and `CheckingAccountFactory` and `PersonalLoanFactory` operations. A similar client application was used to test the PremiumAccount. The aim of these two sets of tests was to verify the correct operation of

the Metamorphic Object subclasses (CheckingAccount, PersonalLoan, PremiumAccount, and LineOfCredit) before and after versioning. These tests were performed using the `CheckingAccountServer`, `PersonalLoanServer`, `PremiumAccountServer`, and `LineOfCreditServer` described earlier.

6.3.1 Functional Testing of CheckingAccount

There are six different scenarios for the versioning of a CheckingAccount object, with and without a PersonalLoan, to a PremiumAccount object. Each case scenario for CheckingAccount and the `accountNumber` assigned to it are shown below.

- Case 1: (caftest1) CheckingAccount, no PersonalLoan.
- Case 2: (caftest2) CheckingAccount, with a PersonalLoan.
- Case 3: (caftest3) CheckingAccount versioned to a PremiumAccount, no PersonalLoan.
- Case 4: (caftest4) CheckingAccount versioned to a PremiumAccount with a PersonalLoan.
- Case 5: (caftest5) CheckingAccount with a PersonalLoan versioned to a LineOfCredit.
- Case 6: (caftest6) CheckingAccount versioned to a PremiumAccount, with a PersonalLoan versioned to a LineOfCredit.

The process for verifying correct operation for each test case was as follows. Note that once the object reference to a CheckingAccount is obtained from the Naming Service, it is used for all invocations, even after the CheckingAccount object has been versioned to a PremiumAccount or a PersonalLoan has been versioned to a LineOfCredit. This proves that the client reference remains valid throughout the versioning process.

1. Create the CheckingAccount with the `accountNumber` indicated using the `createAccount()` operation on the `CheckingAccountFactory`. This also registers the account name with the Naming Service.
2. Resolve the name (`accountNumber`) in the Naming Service to obtain the registered object reference to the CheckingAccount.
3. Perform the operations `balance()`, `deposit()`, `withdrawal()`, `balanceOfLoan()`, and `transferToLoan()`. Verify that the expected values were stored in the MOStore database and returned to the client, or that an exception occurred only where expected.
4. For cases 2, 4, 5, and 6, a PersonalLoan is created using the `PersonalLoanFactory` and assigned to the CheckingAccount. CheckingAccount operations are verified.
5. For cases 5 and 6, the PersonalLoan is versioned to a LineOfCredit object on the `LineOfCreditServer`. The MOStore is checked to verify a new state exists for the object as a LineOfCredit. CheckingAccount operations are verified.
6. For cases 3, 4, and 6, the CheckingAccount is versioned to a PremiumAccount on the `PremiumAccountServer`. The MOStore is checked to verify a new state exists for the object as a PremiumAccount. CheckingAccount operations are verified.

Since the class PremiumAccount is fully compatible with CheckingAccount and LineOfCredit is fully compatible with PersonalLoan, no exceptions should be raised except for when the CheckingAccount attempts to access a PersonalLoan that has not been assigned to it (cases 1 and 3). This was indeed observed during the functional tests. The effect of versioning a CheckingAccount to a PremiumAccount was observable by the results of the CheckingAccount operations. CheckingAccount was implemented to have no service charges for withdrawals and to pay no interest on the balance when a deposit is made. PremiumAccount was defined to levee a service charge (default set to $0.50) for each withdrawal and pay interest (default set to 1%) on the balance when a deposit is made. This made it possible to verify the versioning of CheckingAccount to

PremiumAccount (and visa versa) by observing the object's behavior. The state of MOStore was also examined to verify that the object was versioned correctly. Versioning of PersonalLoan to LineOfCredit cannot be verified by its behavior because the LineOfCredit version of the object has the same interest rate as the original PersonalLoan. Even though it is possible to withdraw funds from the LineOfCredit, the CheckingAccount does not support this operation. It is possible to verify the versioning of a LineOfCredit to a PersonalLoan by its behavior since `transferFromLoan()` for PremiumAccount will raise an exception as the operation is no longer valid.

The `CheckingAccountManager` client application is used to create a CheckingAccount (caftest6) on the CheckingAccount server and a PersonalLoan on the PersonalLoan server in Figures 6-8 and 6-9, respectively. The `CheckingAccountManager` then assigns the loan to the CheckingAccount "caftest6". The test results for case 6, which verify the operation of versioning CheckingAccount "caftest6" to a PremiumAccount and a PersonalLoan to a LineOfCredit, are shown in Figures 6-10 and 6-11 respectively. All versioning operations were verified by inspecting the MOStore database. In all cases, the tests results were as expected.

```
Create new CheckingAccount
ENTER new account Number: caftest6

Time To Create CheckingAccount: 31659 us at Wed Dec  1 18:37:50 1999
New CheckingAccount is:  DVSCA_CheckingAccount :
DVSCA_CheckingAccount19991201183544        //Object Interface : marker

Account number: caftest6
Time :      5224 us : Wed Dec  1 18:37:50 1999
Balance  = 0
Time :      4667 us : Wed Dec  1 18:37:50 1999
Deposit amount  = 500
Time :     15972 us : Wed Dec  1 18:37:55 1999
Balance  = 500                              // No interest
Time :      4896 us : Wed Dec  1 18:37:59 1999
Deposit amount  = 200
Time :     16135 us : Wed Dec  1 18:38:02 1999
Withdrawal amount  = 100   Received = 100
Time :     16305 us : Wed Dec  1 18:38:05 1999
Balance  = 600                              // No service charge
Time :      4959 us : Wed Dec  1 18:38:06 1999
```

Figure 6-8 Creation of CheckingAccount 'caftest6'.

```
Create new PersonalLoan for CheckingAccount
ENTER new account Number: caftest6
Loan Amount (e.g. -500): -500

Time To Create PersonalLoan: 21903 us at Wed Dec  1 18:38:22 1999

New PersonalLoan is:  DVSCA_PersonalLoan :
DVSCA_PersonalLoan19991201183615          // object Interface:marker
host      = banff.genie.uottawa.ca
server    = PersonalLoanServer

Account number: caftest6
Time :     23018 us : Wed Dec  1 18:38:22 1999
BalanceOfLoan  = -500
Time :     10281 us : Wed Dec  1 18:38:22 1999
BalanceOfLoan  = -500
Time :      9490 us : Wed Dec  1 18:38:24 1999
 Enter amount to transfer to Loan: 100
TransferToLoan amount  = 100
Time :     31966 us : Wed Dec  1 18:38:30 1999
BalanceOfLoan  = -440                       // 10% interest on loan
Time :      9429 us : Wed Dec  1 18:38:32 1999
Balance  = 500                        // no service charge on account
Time :      4105 us : Wed Dec  1 18:38:34 1999
```

Figure 6-9 Creation of a PersonalLoan for 'caftest6'.

```
Versioning PersonLoan for Account Number caftest6

Versioning:  DVSCA_PersonalLoan : DVSCA_PersonalLoan19991201183615
 to Interface: DVSPA_LineOfCredit
 on Server:    LineOfCreditServer
 on Host:      banff
Compatible attributes:
currentInterface
previousInterface
newVersionRef
accountNumber
balance
interestRate

Time To version PersonaLoan: 49730 us at Wed Dec  1 18:38:36 1999

New Version Ref :  DVSPA_LineOfCredit :
DVSCA_PersonalLoan19991201183615     // new interface, same marker
host       = banff.genie.uottawa.ca
server     = LineOfCreditServer
BalanceOfLoan  = -440
Time :     27511 us : Wed Dec  1 18:38:41 1999

Balance  = 500
Time :      4295 us : Wed Dec  1 18:38:44 1999

 Enter amount to transfer to Loan: 100
TransferToLoan amount  = 100
Time :     37245 us : Wed Dec  1 18:38:51 1999

BalanceOfLoan  = -374                // maintained 10% interest rate
Time :     14501 us : Wed Dec  1 18:38:53 1999

Balance  = 400                       // CheckingAccount unchanged
Time :      4302 us : Wed Dec  1 18:38:55 1999
```

Figure 6-10 Versioning of PersonalLoan for 'caftest6' to LineOfCredit.

```
Versioning: DVSCA_CheckingAccount:DVSCA_CheckingAccount19991201183544
 to Interface: DVSPA_PremiumAccount
 on Server:    PremiumAccountServer
 on Host:      banff
Compatible attributes:
     currentInterface
     previousInterface
     newVersionRef
     accountNumber
     balance
     loan

Time To version CheckingAccount: 34044 us at Wed Dec  1 18:38:59 1999

New Version: DVSPA_PremiumAccount:DVSCA_CheckingAccount19991201183544
host = banff.genie.uottawa.ca        // new Interface, same marker
server = PremiumAccountServer
Balance  = 400
Time :    28841 us : Wed Dec  1 18:39:02 1999

Deposit amount  = 100
Time :    23873 us : Wed Dec  1 18:39:08 1999

Balance  = 505                       // now gets 1% interest
Time :     9393 us : Wed Dec  1 18:39:10 1999

Withdrawal amount  = 200   Received = 200
Time :    23256 us : Wed Dec  1 18:39:14 1999

Balance  = 304.5                     // now has $0.50 service charge
Time :     9764 us : Wed Dec  1 18:39:16 1999

BalanceOfLoan  = -374
Time :    20193 us : Wed Dec  1 18:39:21 1999

 Enter amount to transfer to Loan: 100
TransferToLoan amount  = 100
Time :    43524 us : Wed Dec  1 18:39:26 1999

BalanceOfLoan  = -301.4              // LineOfCredit with 10% interest
Time :    19968 us : Wed Dec  1 18:39:29 1999

Balance  = 204                       // $0.50 service charge
Time :     9713 us : Wed Dec  1 18:39:31 1999
```

Figure 6-11 Versioning of CheckingAccount 'caftest6' to PremiumAccount.

6.3.2 Functional Testing of PremiumAccount

There were also six different case scenarios for the versioning of a PremiumAccount object, with and without a LineOfCredit assigned, to a CheckingAccount object. These are shown below with their assigned `accountNumbers`.

- Case 1: (paftest1) PremiumAccount, no LineOfCredit.
- Case 2: (paftest2) PremiumAccount, with a LineOfCredit.
- Case 3: (paftest3) PremiumAccount versioned to a CheckingAccount, no LineOfCredit.
- Case 4: (paftest4) PremiumAccount versioned to a CheckingAccount with a LineOfCredit.
- Case 5: (paftest5) PremiumAccount with a LineOfCredit versioned to a PersonalLoan.
- Case 6: (paftest6) PremiumAccount versioned to a CheckingAccount, with a LineOfCredit versioned to a PersonalLoan.

The tests performed for each case were similar to the tests for CheckingAccount. Again, all tests were performed using the initial object reference to a PremiumAccount that was obtained from the Name Service. This proved that the client's object reference remained valid before and after versioning.

The process for verifying correct operation for each test was as follows:

1. Create the PremiumAccount with the `accountNumber` indicated using the `createAccount()` operation on the `PremiumAccountFactory`. This also registers the account name with the name service.
2. Resolve the name (`accountNumber`) in the Naming Service to obtain the registered object reference to the PremiumAccount.
3. Perform the operations `balance()`, `deposit()`, `withdrawal()`, `balanceOfLoan()`, `transferToLoan()`, and `transferFromLoan()`.

Verify that the expected values were stored in the MOStore database and returned to the client or that an exception occurred only where expected.

4. For cases 2, 4, 5, and 6 a LineOfCredit is created using the `LineOfCreditFactory` and assigned to the PremiumAccount. PremiumAccount operations are verified.
5. For cases 5 and 6 the LineOfCredit is versioned to a PersonalLoan object on the `PersonalLoanServer`. The MOStore is checked to verify a new state exists for the object as a PersonalLoan. PremiumAccount operations are verified.
6. For cases 3, 4, and 6 the PremiumAccount is versioned to a CheckingAccount on the `CheckingAccountServer`. The MOStore is checked to verify a new state exists for the object as a CheckingAccount. The operations of the PremiumAccount are verified.

In all cases, the results of the functional tests for PremiumAccount were as expected. This includes the results for cases 3, 4, and 6 where the PremiumAccount object was versioned to a CheckingAccount object and an exception was raised when the client attempted to invoke the unsupported `transferFromLoan()` operation. Likewise, an exception was raised when the operation `withdrawal()` was invoked on a LineOfCredit object that had been versioned to a PersonalLoan. As the `withdrawal()` operation is invoked by a PremiumAccount object, this demonstrates the ability of the DVS to support versioning of nested objects, even to a class that is not fully compatible with the original class. The result of versioning the LineOfCredit for test case "paftest6" to a PersonalLoan is shown in Figure 6-12. An exception is raised when the PremiumAccount object attempts to withdraw an amount of 100.0 from the object that is now a PersonalLoan.

As was seen for versioning of CheckingAccount and PersonalLoan, when a PremiumAccount or LineOfCredit is versioned, its marker, which uniquely identifies the object on a server, remains the same even though the object now exists as a different class with a different implementation. This was verified by observing the MOStore database before and after versioning. These tests demonstrated the concept and operation

of versioning an object from one class definition and implementation to another using the DVS.

```
Versioning:  DVSPA_LineOfCredit : DVSPA_LineOfCredit19991201185647
 to Interface: DVSCA_PersonalLoan
 on Server:    PersonalLoanServer
 on Host:      banff
Compatible attributes:
currentInterface
previousInterface
newVersionRef
accountNumber
balance
interestRate

Time To version LineOfCredit: 52023 us at Wed Dec  1 18:59:24 1999
New Version: DVSCA_PersonalLoan : DVSPA_LineOfCredit19991201185647
host       = banff.genie.uottawa.ca  // new Interface, same marker
server     = PersonalLoanServer
BalanceOfLoan  = -651                // balance from LineOfCredit
Time :     28245 us : Wed Dec  1 18:59:27 1999

Balance  = 717.145                   // PremiumAccount balance
Time :      4331 us : Wed Dec  1 18:59:29 1999

 Enter amount to transfer to Loan: 200
TransferToLoan amount  = 200
Time :     40124 us : Wed Dec  1 18:59:39 1999

BalanceOfLoan  = -473.55             // PersonalLoan has 5% interest
Time :     16798 us : Wed Dec  1 18:59:41 1999

Balance  = 516.645                   // $0.50 service charge
Time :      4366 us : Wed Dec  1 18:59:44 1999

 Enter amount to transfer from Loan: 100
Error calling PremiumAccount->transferFromLoan()
Reason: CanNot Access Loan           // Exception, invalid for P.L.
Error, Could Not TransferFromLoan amount  = 100
Time :     21456 us : Wed Dec  1 18:59:49 1999

BalanceOfLoan  = -473.55          // transfer from PersonlLoan failed
Time :     15697 us : Wed Dec  1 18:59:54 1999

Balance  = 516.645
Time :      4487 us : Wed Dec  1 18:59:56 1999

 Enter amount to transfer to Loan: 100
TransferToLoan amount  = 100
Time :     41446 us : Wed Dec  1 19:00:02 1999

BalanceOfLoan  = -392.227            // transfer to PersonalLoan okay
Time :     15348 us : Wed Dec  1 19:00:04 1999
```

Figure 6-12 Versioning of the LineOfCredit for "paftest6" to a PersonalLoan.

6.4 Performance Testing of the DVS Bank Application.

Four sets of performance tests were performed on the DVS Bank application. These included local and remote clients for CheckingAccount and PremiumAccount. For each set of tests, six case scenarios were tested. The case scenarios for the CheckingAccount client are:

- Case 1: CheckingAccount, no PersonalLoan.
- Case 2: CheckingAccount, with a PersonalLoan.
- Case 3: CheckingAccount versioned to a PremiumAccount, no PersonalLoan.
- Case 4: CheckingAccount versioned to a PremiumAccount with a PersonalLoan.
- Case 5: CheckingAccount with a PersonalLoan versioned to a LineOfCredit.
- Case 6: CheckingAccount versioned to a PremiumAccount, with a PersonalLoan versioned to a LineOfCredit.

Similarly, the test case scenarios for the PremiumAccount client are:

- Case 1: PremiumAccount, no LineOfCredit.
- Case 2: PremiumAccount, with a LineOfCredit.
- Case 3: PremiumAccount versioned to a CheckingAccount, no LineOfCredit.
- Case 4: PremiumAccount versioned to a CheckingAccount with a LineOfCredit.
- Case 5: PremiumAccount with a LineOfCredit versioned to a PersonalLoan.
- Case 6: PremiumAccount versioned to a CheckingAccount, with a LineOfCredit versioned to a PersonalLoan.

The significant difference between the CheckingAccount and PremiumAccount scenarios is that the PremiumAccount operation `transferFromLoan()` is not part of the IDL definition for CheckingAccount. This illustrates the ability of the DVS to version an object of one class to another class definition where the rules of polymorphism are not satisfied for all operations defined in the original class. The result of invoking this operation in cases 3, 4, and 6 is that an exception is returned to the PremiumAccount client. This was seen in the functional tests for PremiumAccount. In case 5, an exception

is also returned to the PremiumAccount client but this time it is caused by the PremiumAccount object invoking the operation `withdrawal()` on a PersonalLoan object. In this case, the ORB returns an invalid operation exception to the PremiumAccount object, which raises the `CanNotAccessLoan` exception to the client. The exception impacts the performance tests in that the delay caused by versioning a PremiumAccount object to a CheckingAccount or versioning a LineOfCredit object to a PersonalLoan cannot be measured for the PremiumAccount operation `transferFromLoan()`.

The test case scenarios for a local CheckingAccount client are shown in Figure 6-13 and indicate which operation calls are passed between processes. For example, in Case 1, all 5 operation calls are passed from the client process to the CheckingAccount (CA) server but only operations 2 and 3, `deposit()` and `withdrawal()`, are passed to the MOStore database on the remote server host "hyperion". The tests for a remote CheckingAccount client include the same test case scenarios but with the client process residing on the remote host "arnie". Similarly, the test cases for a local PremiumAccount client are shown in Figure 6-14. The remote client tests were performed with the PremiumAccount client residing on the remote host "arnie". The difference in performance for both remote clients is discussed later.

To measure the impact of versioning on the test objects, the functions performed for each client operation in each of the test cases must be identified. Since all test objects use MOStore for persistence, a component of each operation invocation may require a database transaction to occur. Each message between processes, whether an explicit operation call or a Forwarder message, requires an IIOP message between ORBs. Remoting the client from the server may also impact the operation invocation times. The components of each operation invocation spent because of database accesses, forwarding, and a remote client are identified and measured for all test cases for each test scenario.

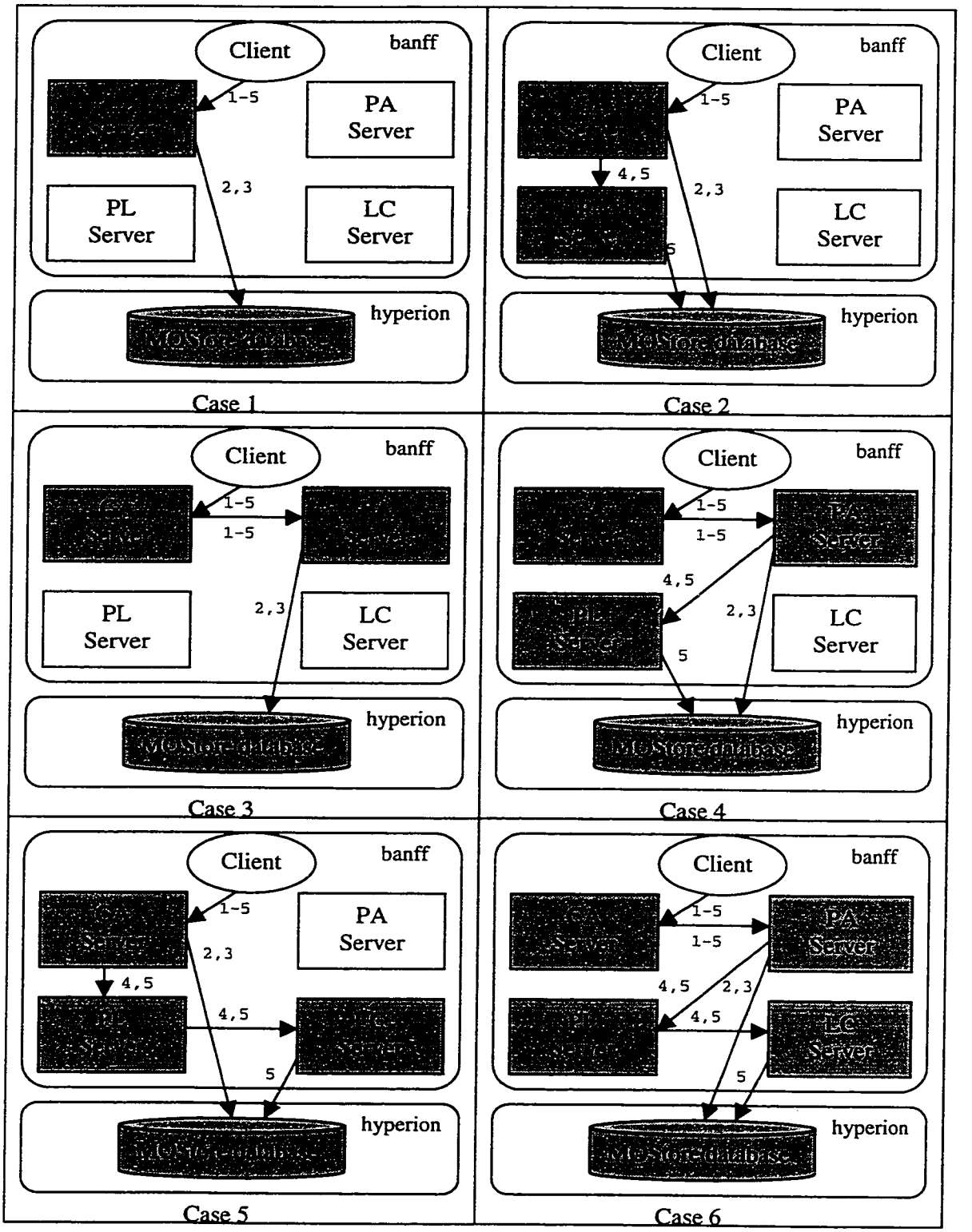

Figure 6-13 Test Case Scenarios for Local CheckingAccount Client.

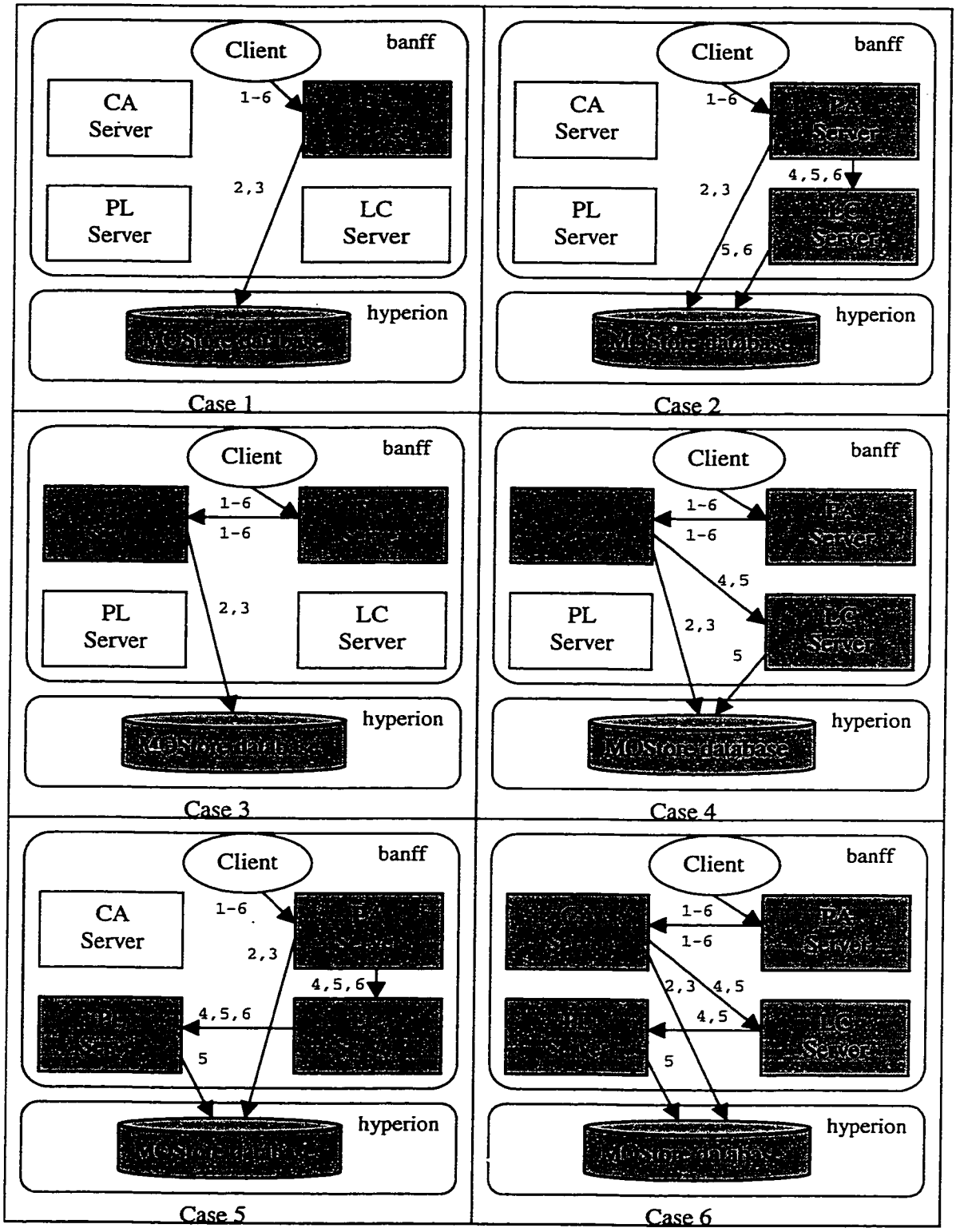

Figure 6-14 Test Case Scenarios for Local PremiumAccount Client.

Tables 6-2 and 6-3 show the number of inter-process CORBA IIOP messages and SQL database calls for each CheckingAccount and PremiumAccount client test case. There are no additional IIOP messages for remote clients, the initial IIOP message is transmitted over the LAN instead of between processes on the same host.

Table 6-2 Number of IIOP and SQL messages for CheckingAccount Client Test Cases.

	Case 1		Case 2		Case 3		Case 4		Case 5		Case 6	
Operation	IIOP	SQL	IIOP	SQL	IIOP	SQL	IIOP	SQL	IIOP	SQL	IIOP	SQL
1. balance()	1	0	1	0	2	0	2	0	1	0	2	0
2. deposit()	1	1	1	1	2	1	2	1	1	1	2	1
3. withdrawal()	1	1	1	1	2	1	2	1	1	1	2	1
4. balanceOfLoan()	*	*	2	0	*	*	3	0	3	0	4	0
5. transferToLoan()	*	*	2	1	*	*	3	1	3	1	4	1

*Operation is invalid and an exception is raised

Table 6-3 Number of IIOP and SQL messages for PremiumAccount Client Test Cases.

	Case 1		Case 2		Case 3		Case 4		Case 5		Case 6	
Operation	IIOP	SQL	IIOP	SQL	IIOP	SQL	IIOP	SQL	IIOP	SQL	IIOP	SQL
1. balance()	1	0	1	0	2	0	2	0	1	0	2	0
2. deposit()	1	1	1	1	2	1	2	1	1	1	2	1
3. withdrawal()	1	1	1	1	2	1	2	1	1	1	2	1
4. balanceOfLoan()	*	*	2	0	*	*	3	0	3	0	4	0
5. transferToLoan()	*	*	2	1	*	*	3	1	3	1	4	1
6. transferFromLoan()	*	*	2	1	*	*	*	*	*	*	*	*

* Operation is invalid and an exception is raised

The test applications were run on the network setup shown in Figure 6-15. Each host was running a large number of processes, including applications not involved with these experiments. For example, the host "hyperion", which runs the MySQL database for

MOStore, is used for development of other applications in the MIRLab. Likewise, the host "banff" for the Bank servers and local clients was also running the Orbix Naming Service used by multiple CORBA applications. Although this is a realistic environment for distributed CORBA applications, it resulted in large fluctuation in invocation times.

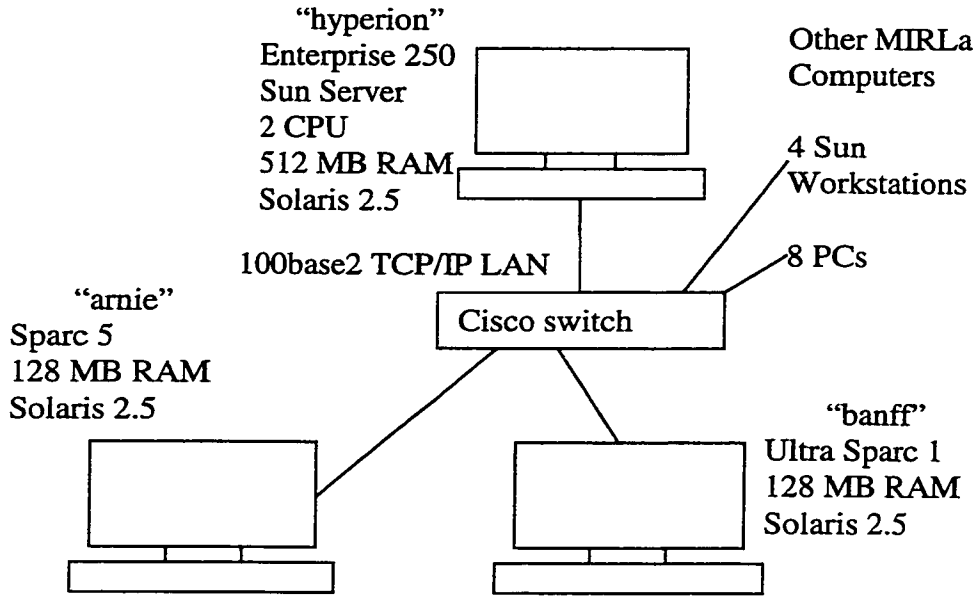

Figure 6-15 DVS Bank Test Setup in the MIR Lab.

In order to obtain statistically meaningful averages of operation invocation times, each operation in each test case was executed 100 times. This resulted in over 13,000 values for the local and remote CheckingAccount and PremiumAccount clients. Also, a large number of inter-server operation invocation times were collected using process filters on each of the servers. These inter-server times were collected to verify DII requests during the Forwarding process. All times were measured to the nearest microsecond by starting and stopping a Unix system timer immediately before and after the operation call. The filters used the same timer to measure the time from when the IIOP message was received by the server until the time the operation returned to the caller.

The result of these performance tests is a large data set that can be used to (1) calculate average invocation times for each type of operation, and (2) identify the effect of

Forwarding, database access, and remote invocation on performance. To illustrate the performance data collected, all operation invocation times for 100 iterations for local and remote CheckingAccount and PremiumAccount clients are shown in the histograms in Figure 6-16 to Figure 6-19. Each line represents the invocation time for an operation, such as `balance()` in CheckingAccount or PremiumAccount, respectively, but they are not labeled for clarity. The horizontal axis represents the iterations in each histogram.

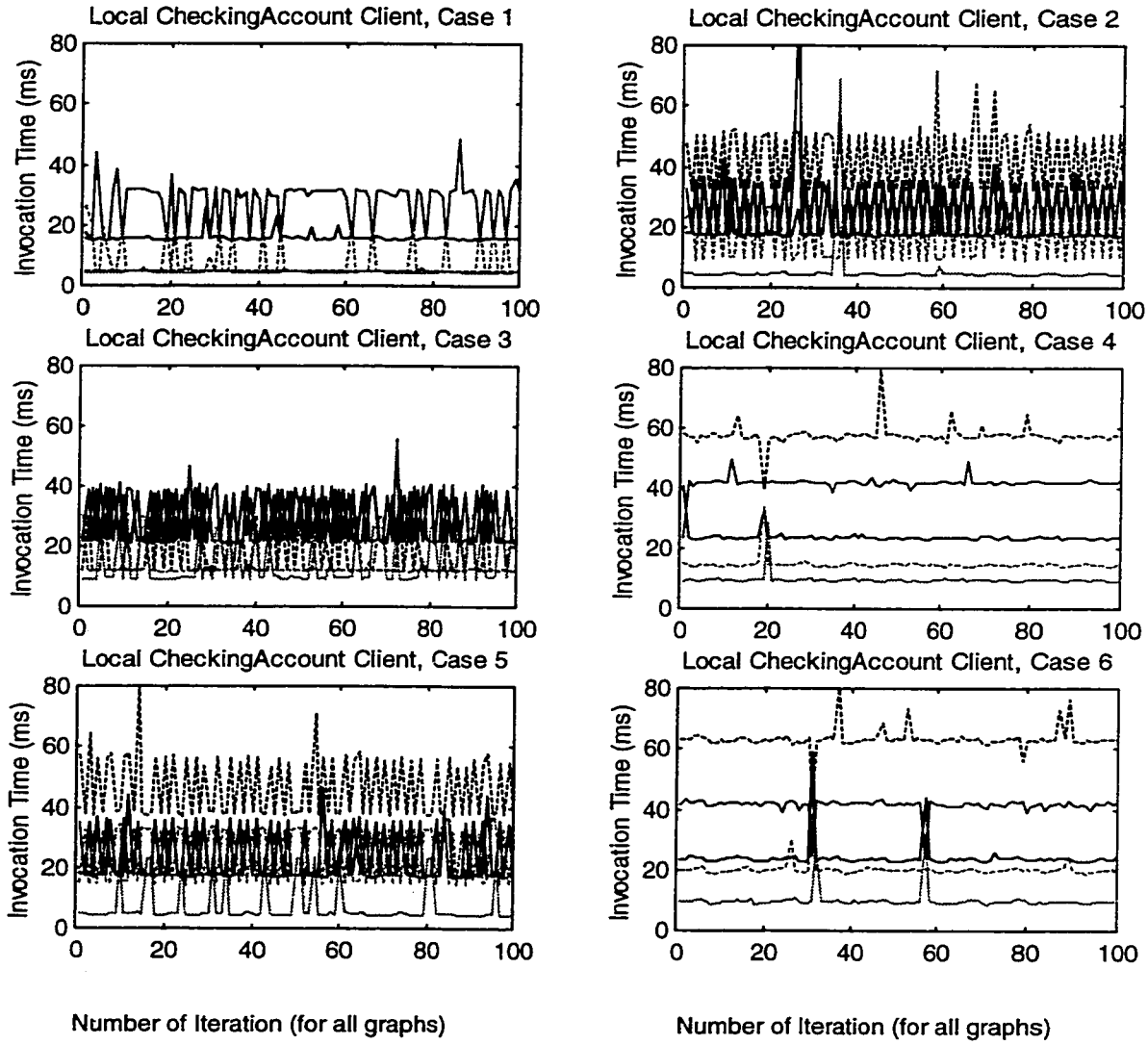

Figure 6-16 Local CheckingAccount Client Invocation Times.

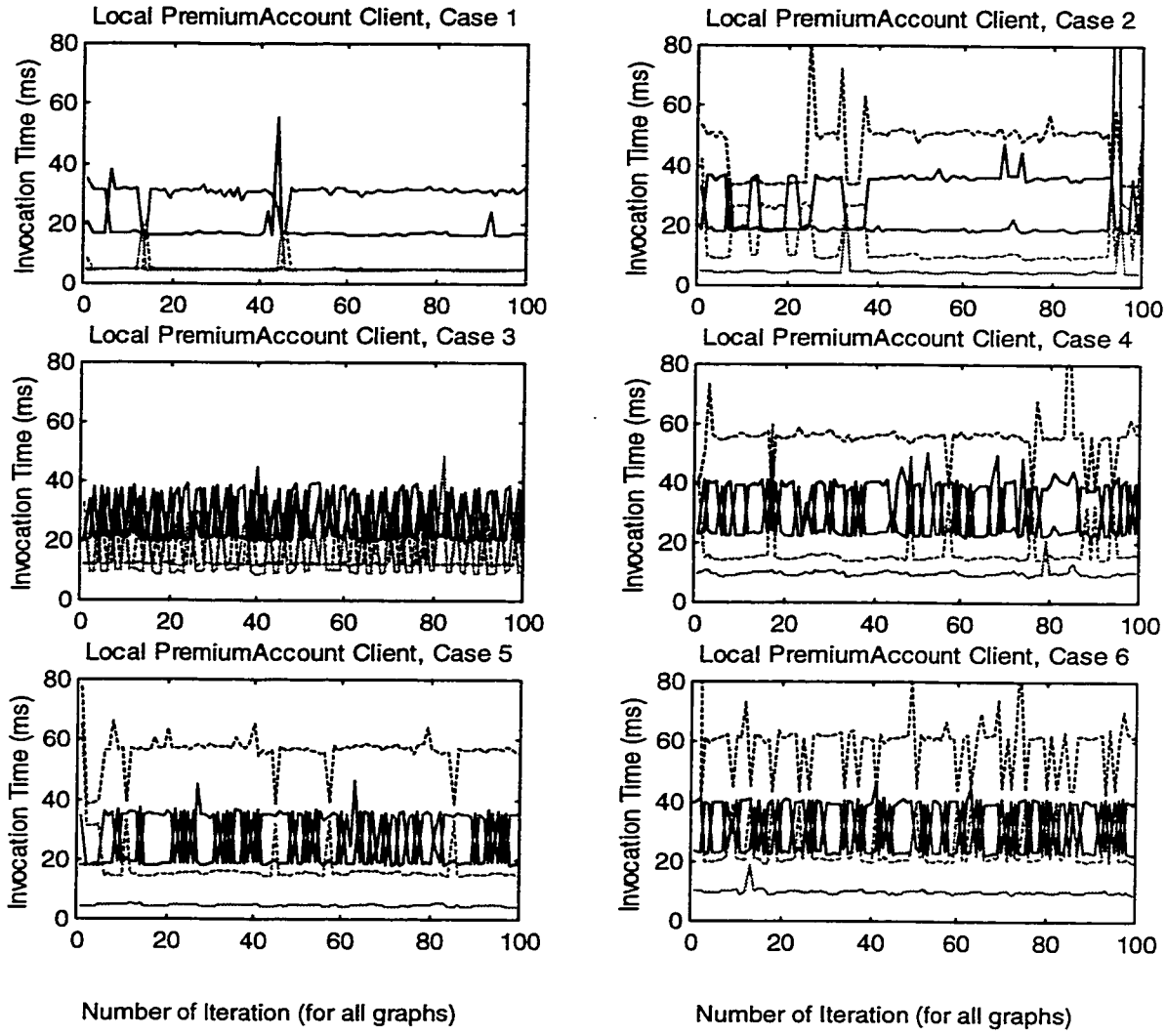

Figure 6-17 Local PremiumAccount Client Invocation Times.

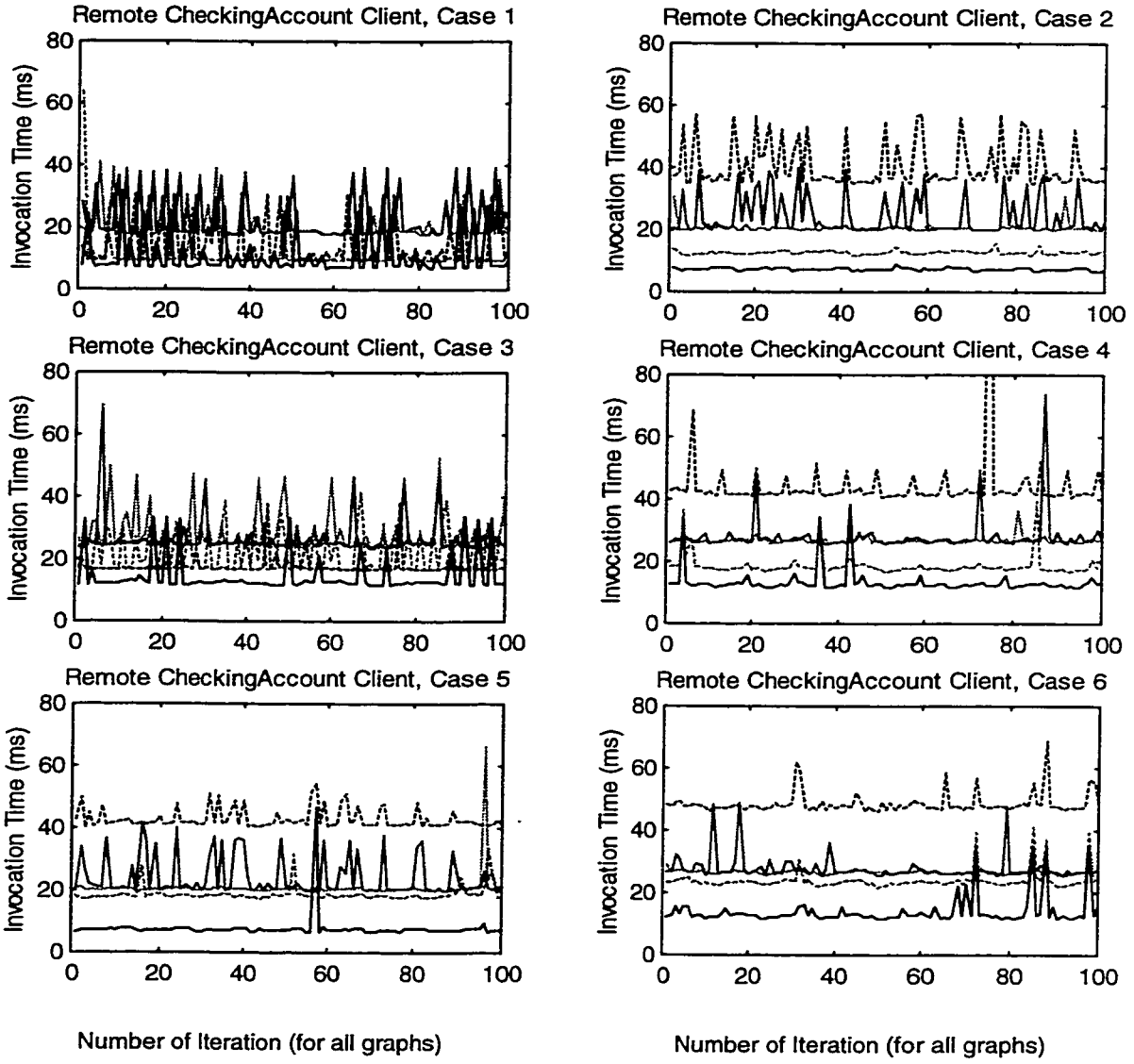

Figure 6-18 Remote CheckingAccount Client Invocation Times.

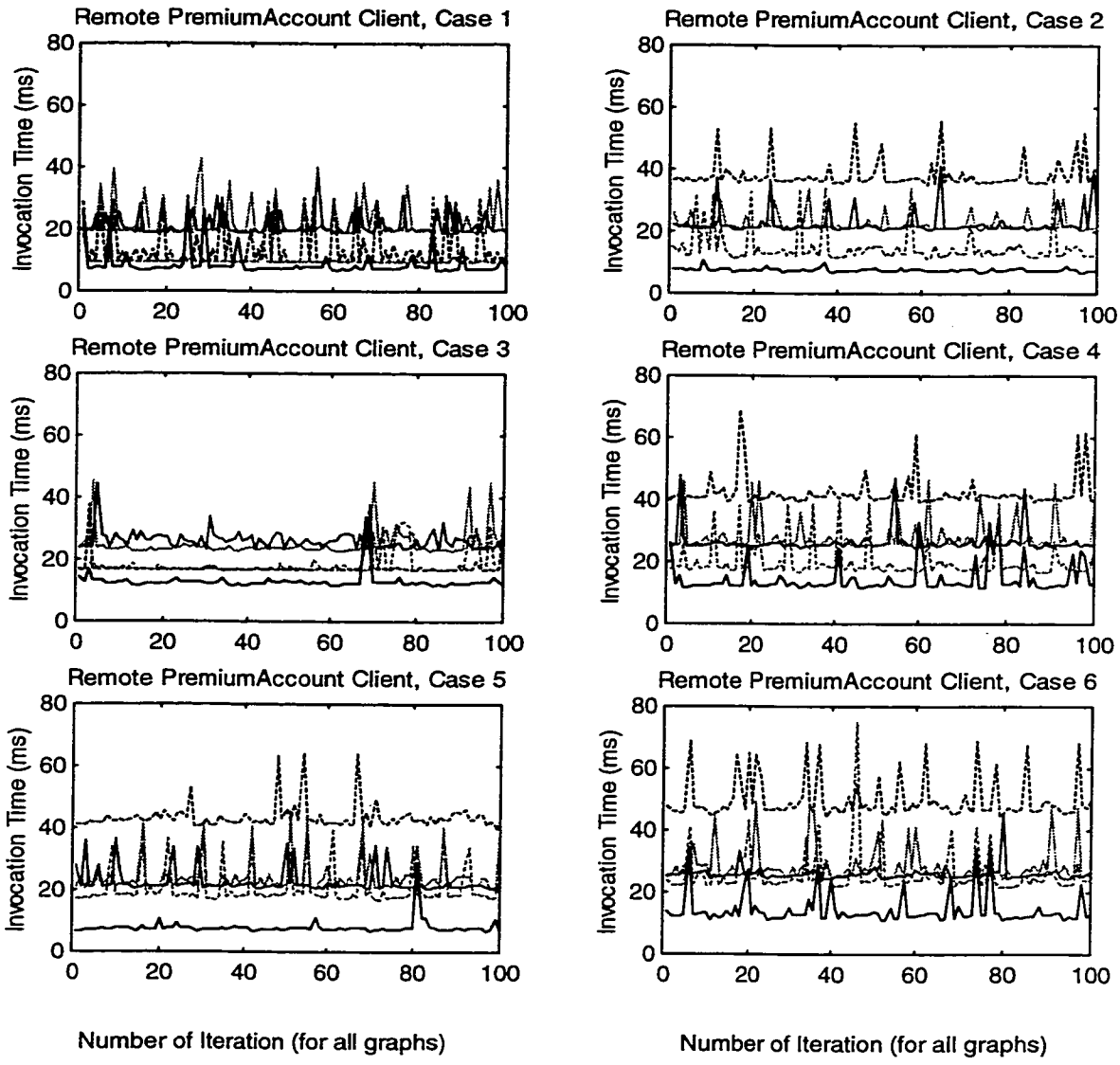

Figure 6-19 Remote PremiumAccount Client Invocation Times.

The timing data was analyzed using the MATLAB® software package. Each of the operation timings in Figures 6-16 to 6-19 was collected as a separate array for each test case. As can be seen in Figures 6-16 to 6-19, there was a great deal of fluctuation of invocation times for the various test case scenarios. These fluctuations, due mainly to the processing load on *banff*, required that a large sample of data be collected to ensure a good statistical average. These fluctuations are measured in Figures 6-20 and 6-21. Figure 6-20 shows the minimum and maximum invocation times for each CheckingAccount and PremiumAccount operation (b = `balance`, d = `deposit`, w =

`withdrawal`, `o` = `balanceOfLoan`, `t` = `transferToLoan`, and `r` = `transferFromLoan`) grouped by the six test cases. In many test cases, an unusually long invocation time occurs. These appear to be infrequent and are probably caused by other processes running on *banff*, as these excessive invocation times appear to decrease in frequency and duration when the client application is moved from *banff*. The standard deviation of invocation times for the same groupings of operation invocation times demonstrates this. The left-hand figures are for client processes residing on the same host as the server and the right-hand figure represent the cases where the client process resides on a remote host.

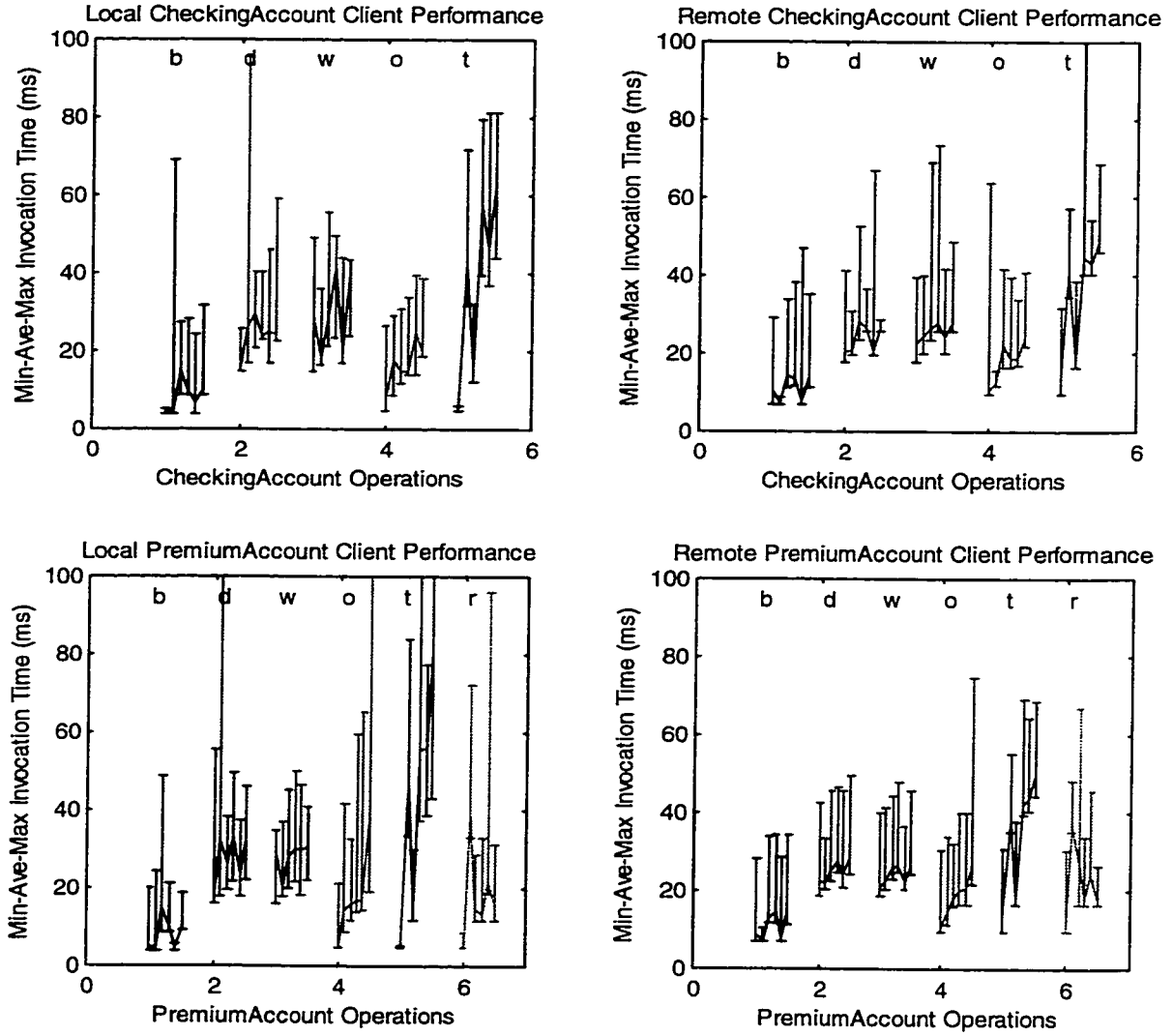

Figure 6-20 Fluctuations in Operation Invocation Times.

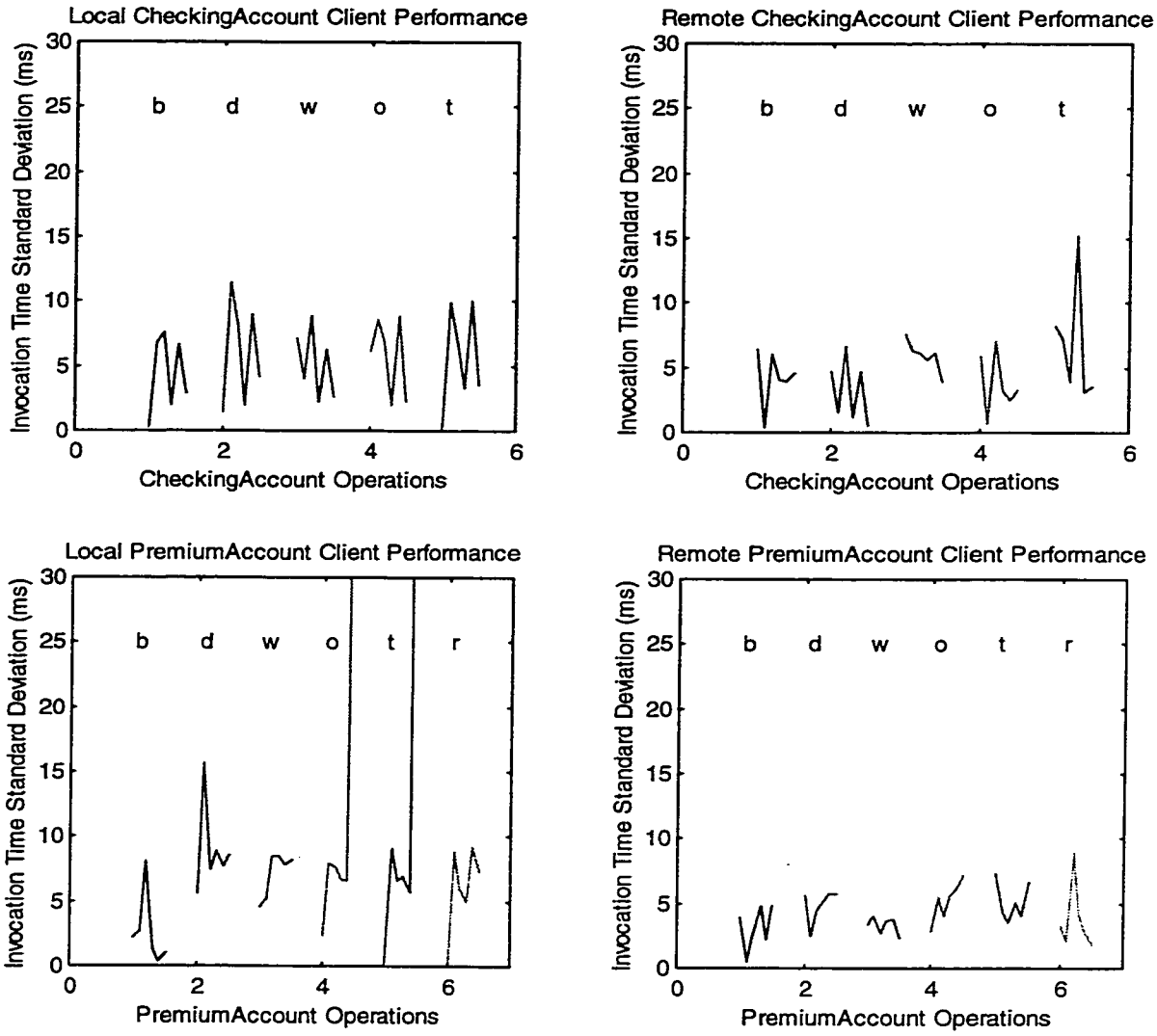

Figure 6-21 Standard Deviation of Operation Invocation Times.

To analyze the components of the time spent in each operation invocation, the operations for CheckingAccount and PremiumAccount are grouped based on each case scenario as to whether they include Forwarding or database accesses. The grouping of operations is shown in Table 6-4 and 6-5.

Table 6-4 Test Case Grouping for CheckingAccount Operations

	NO Forwarding		1 Forwarding		2 Forwarding	
Operation	0 SQL	1 SQL	0 SQL	1 SQL	0 SQL	1 SQL
1. balance()	1,2,5		3,4,6			
2. deposit()		1,2,5		3,4,6		
3. withdrawal()		1,2,5		3,4,6		
4. balanceOfLoan()	2		4,5		6	
5. transferToLoan()		2		4,5		6

Table 6-5 Test Case Grouping for PremiumAccount Operations

	NO Forwarding		1 Forwarding		2 Forwarding	
Operation	0 SQL	1 SQL	0 SQL	1 SQL	0 SQL	1 SQL
1. balance()	1,2,5		3,4,6			
2. deposit()		1,2,5		3,4,6		
3. withdrawal()		1,2,5		3,4,6		
4. balanceOfLoan()	2		4,5		6	
5. transferToLoan()		2		4,5		6
6. transferFromLoan()		2		4		

The operation times were collected from each test case array using the groupings in Table 6-4 and 6-5. The average invocation time from the local and remote clients for CheckingAccount and PremiumAccount was calculated for each operation/test case grouping. For example, the average invocation time for the local CheckingAccount client operation `balance()` before versioning is an average of the results for that operation from test cases 1, 2, and 5. Because the groupings are based on the number and types of functions performed for each operation invocation, multiple test cases can be averaged together to reduce the quantity of data and to increase the number of samples for the average invocation times. This results in four plots of the average invocation times for local and remote CheckingAccount and PremiumAccount clients. These are shown in Figure 6-22 with the left and right bars indicating the average invocation time before and after versioning, respectively.

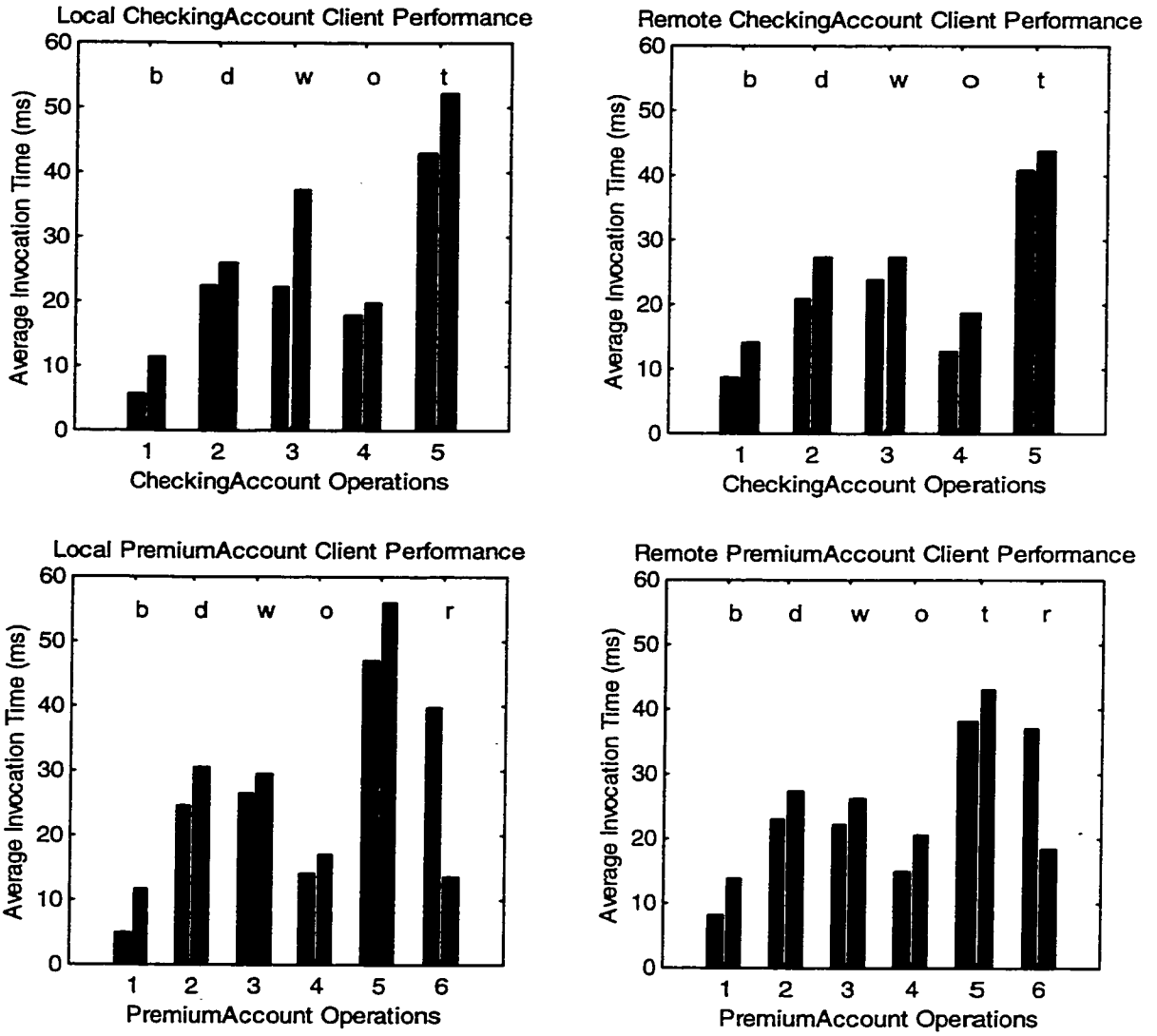

Figure 6-22 Average Operation Invocation Times before and after Versioning.

The results in Figure 6-22 illustrate the effect of versioning on operation invocation time. The additional IIOP message that is sent from the Forwarder servant for the original object to its new implementation adds an average of about 6 milliseconds to the client invocation time. The Forwarder IIOP message, consisting of the operation invocation and result return, is roughly equivalent to the invocation time for local client operation on an non-versioned server object, such as `balance()` for a local CheckingAccount client. This is as expected since the Forwarder acts as a client to the new version of the object,

sending a single DII operation request to a server on the same host. For all four test scenarios, the times for `balance()` can be taken as the base invocation time for objects before and after versioning as it requires a single IIOP message and no database operations. The effect of versioning adds one IIOP message to be sent, roughly doubling the invocation time as seen in Figure 6-22.

There are some fluctuations in the results such as the unexpected increase in invocation time for the `withdrawal()` operation for the local CheckingAccount client. Operations 2 and 3, `deposit()` and `withdrawal()`, are expected to have equivalent times for all test scenarios as they send the same number of IIOP and SQL messages. This may have been caused by a high level of processor activity on *banff* during testing. A comparison of local and remote client invocations shows a slight decrease in invocation times for the remote client. This also indicates that additional delays are occurring due to processor activity on *banff*.

Another apparent discrepancy is the times for versioning a PremiumAccount based on operation 6, `transferFromLoan()`, which shows a dramatic decrease in invocation times after versioning. However, as mentioned earlier, these timings are the result of the operation throwing an exception. The invocation times for operation 6 are presented to illustrate the effect of invoking an operation that is no longer valid for the new class definition for the versioned object. The invocation times for operation 6 for the PremiumAccount and all other cases where an exception is thrown are not used in calculating the average delays for Forwarding, database access, and remote clients presented below.

The operation timings shown in Figure 6-22 were grouped by three criteria: whether they include Forwarding or not (Versioned/Not Versioned), whether they include one or zero databases accesses, and whether the client is local or remote. The average differences in invocation times for each criterion were then calculated for the other four groupings. For example, the measure of the delay caused by versioning is calculated for a local and remote client when there is zero and one database access. The average operation

invocation delay caused by versioning is shown in Table 6-6, along with a measure of the relative increase in invocation times. The same results are illustrated in Figure 6-23, with the left and right bars indicating the average invocation time before and after versioning, respectively.

Table 6-6 Effect of Versioning on Operation Invocation Time

Grouping Criteria	Average Delay (ms)	Relative increase in invocation time
Local client, 0 database access	6.3070	2.2095
Remote client, 0 database access	5.6013	1.6756
Local client, 1 database access	6.9362	1.2903
Remote client, 1 database access	4.5831	1.2047
Overall average	5.8569	1.5950

The value of 5.8569 milliseconds represents the overall average delay added by the versioning process to operation invocation times. This average value is consistent with the versioning delays for individual operations shown in Figure 6-20. This value is also close to the invocation time for a local client operation invocation that does not require a database access, which was found to be 5.214 milliseconds for the test environment.

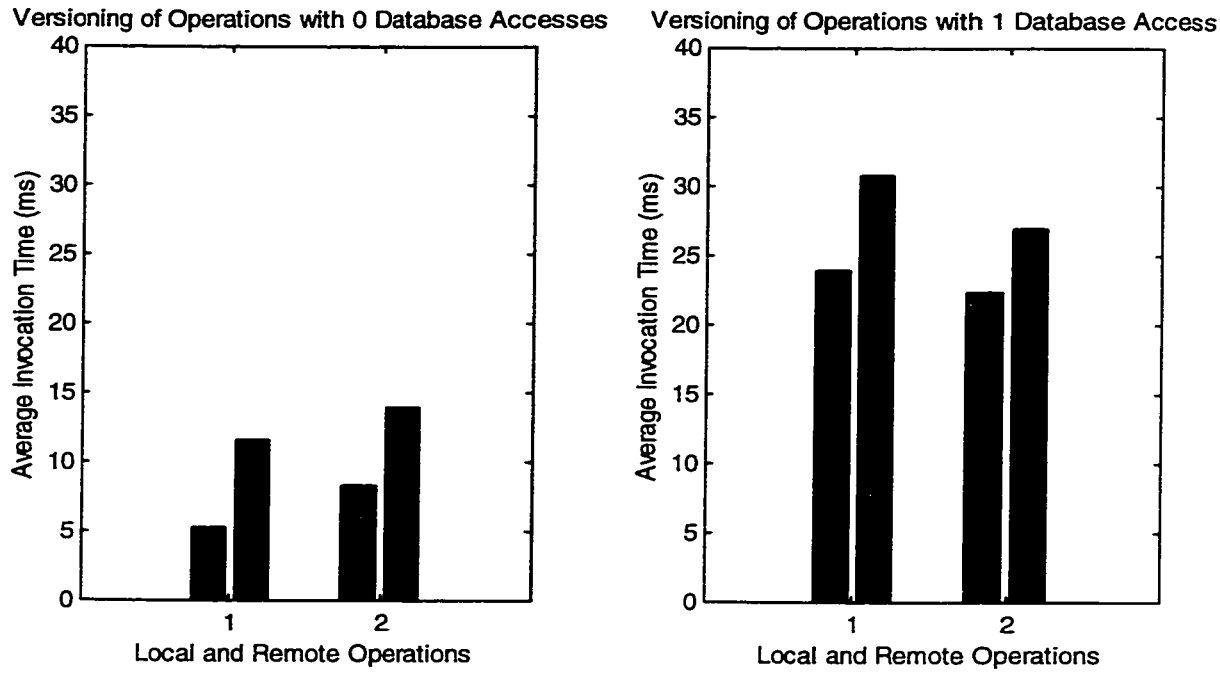

Figure 6-23 Comparisons of Invocation Times before and after Versioning.

As can be seen in Figure 6-23, the time spent by an operation making a database access is far greater than the delay added by the Forwarding operation for versioned objects. The effect of database access on invocations times is shown in Table 6-7 and is found to be approximately three times the duration of a Forwarder operation. Since all Metamorphic Objects must be persistent, the effect of database performance is more important than the effect of Forwarding.

Table 6-7 Effect of Database Access on Operation Invocation Time

Grouping Criteria	Average Delay (ms)	Relative increase in invocation time
Local client, Not Versioned	18.6753	4.5814
Remote client, Not Versioned	14.0956	2.7002
Local client, Versioned	19.3044	2.6755
Remote client, Versioned	13.0774	1.9414
Overall average	16.2882	2.9746

The last delay calculated is the difference between the client residing on the same host as the server or on a remote host. One of the main aspects of CORBA is its ability to hide message passing between client and server processes, whether they are on the same machine or separate hosts. This was tested by running the same CheckingAccount and PremiumAccount client applications on the same host as the servers (*banff*) and then on a separate host (*arnie*). The effects of running the client processes on the remote host are shown in Table 6-8.

Table 6-8 Remote Client Operation Invocation Times relative to Local Client.

Grouping Criteria	Local Client Average Invocation Time (ms)	Remote Client Average Invocation Time (ms)	Relative increase in invocation time
0 Database Access, Not Versioned	5.2146	8.2907	1.5899
0 Database Access, Versioned	11.5236	13.8940	1.2057
1 Database Access, Not Versioned	23.9030	22.3995	0.9371
1 Database Access, Versioned	30.8281	26.9715	0.8749
Overall average	17.8673	17.8889	1.0012

It is interesting to note that the invocation times were better for a remote client than a local client when the operation required a database access. This is because the processing load on the server host *banff* was reduced by moving the client applications to *arnie*. The overall execution time of these operations was decreased even though it required the client's IIOP message to be transmitted over the TCP/IP LAN instead of between processes. This is an example of the benefits of distributed computing using CORBA.

7 Conclusions and Future Research

Distributed computing will play an important role in Information Technology of the 21st century. An evolving standard for distributed applications is the Common Object Request Broker Architecture defined by the Object Management Group. Many service and facilities are defined for CORBA to aid in the design and development of large, enterprise-scaled applications. However, no existing CORBA mechanism is specified to support the dynamic evolution of these applications. The need to incrementally modify the functionality and implementation of large distributed applications, without an interruption in service, has led to the research in this thesis.

7.1 Concepts Addressed in this Thesis

The concept of modifying objects in a distributed CORBA-based application without interruption in service has been termed "dynamic reconfiguration" [21]. The concept of dynamic reconfiguration of a CORBA object is extended in this thesis to include changing the OMG IDL interface, implementation language class definition, and server implementation. The result is the novel concept of a Metamorphic Object that can dynamically change its interface, functionality, implementation, and location without invalidating any client object references to it. In theory, the use of metamorphic objects to realize a distributed application could enable the application to be continuously evolved in terms of functionality and technology, without loss of service, indefinitely.

This thesis presents a prototype of a Dynamic Versioning Service to support the evolution of metamorphic objects in a CORBA environment. This service uses the concept of Forwarding operation invocations between versions of a metamorphic object using the Dynamic Invocation Interface in CORBA. The approach provides a means of mapping the functionality and state of a metamorphic object from one version to another.

The specifications for a CORBA service for versioning metamorphic objects are presented. A test application was developed that demonstrated the concept of versioning and measured the performance cost of Forwarding. Results are presented on the impact to performance caused by the dynamic versioning of Metamorphic Objects.

7.2 Contributions of this Thesis

Overall, this thesis examines the problem of object evolution and dynamic configuration in a distributed environment, resulting in the new concept of a metamorphic object. A mechanism is proposed for dynamic versioning of metamorphic objects, and the results of implementing a Dynamic Versioning Service as part of a distributed system using CORBA are presented.

More specifically, the need for and the problems of object evolution in a distributed environment were addressed. This includes a review of related research in the field of dynamic reconfiguration of distributed systems. A new concept in object-oriented programming, a Metamorphic Object, is presented to address these requirements. Forwarding is proposed as mechanism to realize Metamorphic Objects in a distributed environment. The concept of dynamically versioning Metamorphic Objects is described and the criteria for mapping a Metamorphic Object between class definitions is presented. A framework is presented for a Dynamic Versioning Service for CORBA. The Forwarder Pattern for metamorphic objects is presented as part of the design of this framework. Other components of the Dynamic Versioning Service such as the persistent store for metamorphic objects are included in the design. A specification for a complete Dynamic Versioning Service is presented using the Interface Definition Language and a detailed design of the core components are presented using the Unified Modeling language. A test application using the DVS is constructed and results are presented on the impact to performance caused by the dynamic versioning of metamorphic objects. The impact of Forwarding on operation invocation times is measured along with the times for accessing the persistent store and remote invocations. This test application demonstrated that

objects could be transformed to a new IDL interface, implementation class definition, and server implementation without invalidating any existing object references or causing an interruption in service. Forwarding was found to increase the communication time of an operation invocation by the duration of a single IIOP message consisting of an operation invocation and returned value.

This thesis made several significant research contributions, which can be summarized as follows:

1. A new concept in object-oriented programming, Metamorphism, is presented.

2. A description of Metamorphic Objects in a distributed environment is presented in chapter 4. A method for transforming object instances between different interface and class definitions is presented. The semantics and precept for invoking operations on a new version of a metamorphic object using the original object reference and interface is also presented.

3. A framework for a Dynamic Versioning Service is described in chapter 5. This service is fully compliant with the CORBA 2.0 specification and is implemented at the application level, without modification to the ORB or IDL compiler.

4. The Forwarder Pattern for realizing Metamorphic Objects is presented. This pattern can be used to dynamically change the interface and implementation of a server object, from the perspective of a client process.

5. The requirements of a persistent store and configuration management for the Dynamic Versioning Service are also discussed.

6. A functional prototype of a Dynamic Versioning Service for CORBA is constructed. A complete metamorphic object application is developed in Chapter 6. The impact of dynamic versioning of Metamorphic Objects on performance is

measured. The performance overhead of Forwarding was found to be small and acceptable to most applications.

7.3 Future Research

The implementation of a complete Dynamic Versioning System to realize metamorphic objects and provide dynamic reconfiguration of real-time distributed systems is beyond the scope of this thesis. Work in this field should be continued in order to validate the concept of metamorphic objects in a more stringent environment. This would include the following.

- Development of a Version Manager Service as part of a complete Dynamic Versioning Service for CORBA. This would include defining the configuration management functions required to manage (1) objects that can exist across multiple versions of an IDL interface, (2) implementation class definitions, and (3) server implementations.
- Investigate the use of an Object-Oriented Database Management System to implement the persistent store for the Dynamic Versioning Service. This could allow development of metamorphic objects with little or no constraints on their attributes, operations, or aggregates.
- Investigate the new version 3.0 of the CORBA specification and ORB implementations by different vendors. Determine the use of the new Portable Object Adapter (POA) in replacing the Loader or MOManager presented in this thesis.
- Investigate the implementation of the DVS in different programming languages, including Java, and operating systems, including MS-WindowsNT.
- Investigate techniques for updating client-held object references after a metamorphic object has been versioned to a new interface, class, and implementation. This should include new features available in CORBA 3.0.

- Investigate the theory of metamorphic objects. Expand on the basic concepts presented here and develop a formal theory for metamorphic objects using the theory of Abstract Data Types.
- Investigate other applications of the Forwarder Pattern including gateways and server security applications.

8 Bibliography

[1] D. C. Schmidt and S. Vinoski, "Object Adapters: Concepts and Terminology", C++ *Report*, vol. 11, November/December 1997.

[2] D. C. Schmidt and S. Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects", C++ *Report*, vol. 12, April 1998.

[3] D. C. Schmidt and S. Vinoski, "Developing C++ Servant Classes Using the Portable Object Adapter", C++ *Report*, vol. 13, June 1998.

[4] D. C. Schmidt and S. Vinoski, "C++ Servant Managers for the Portable Object Adapter", C++ *Report*, vol. 14, September 1998.

[5] J. Klefndfenst, F. Plasfl, and P. Tuma, "Lessons Learned from Implementing the CORBA Persistence Object Service", in Proceedings of OOPSLA'96, (San Jose, CA), ACM, October 1996.

[6] S. Vinoski and M. Henning, *Advanced CORBA Programming With C++*. Addison-Wesley Longman, 1998.

[7] S. Baker, *CORBA Distributed Objects Using Orbix*, Addison-Wesley Longman, 1997.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[9] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[10] T. Quatrani, *Visual Modeling With Rational Rose And UML*, Addison-Wesley Longman, 1998.

[11] D. Ionescu, V. Groza, R. Ocica, V. Cimpu, M. Trifan, M. Cimpu, V. Vieru, "A Distributed Network Management Environment", Proceedings of CONTI'98, The International Conference on Technical Informatics, third edition, Timisoara, Romania, vol. 3, pp 49-60, 1998.

[12] V. Cimpu, D. Ionescu, M. Cimpu, "Dynamic Managed Objects for Network Management", Machine Intelligence Laboratory, University of Ottawa, Canada, 1998.

[13] D. Ionescu, V. Groza, M. Trifan, R. Ocica, C. Lambiri, "Report No: 1 BaseLayer Functional Specification", Machine Intelligence Laboratory, University of Ottawa, Canada, 1997.

[14] D. Ionescu, V. Groza, M. Trifan, R. Ocica, C. Lambiri, "Report No: 2 BaseLayer General Architecture", Machine Intelligence Laboratory, University of Ottawa, Canada, 1997.

[15] D. Ionescu, R. Ocica, "Report No: 1 Interaction Translation", Machine Intelligence Laboratory, University of Ottawa, Canada, 1999.

[16] Active Network Working Group, *Architectural Framework for Active Networks*, Version 0.9, August 1998.

[17] "Life Cycle Service Specification", Object Management Group, November 1996.

[18] "Persistent Object Service Specification" Object Management Group, March 1995.

[19] "The Common Object Request Broker: Architecture and Specification (Revision 2.0)", Object Management Group, July 1995.

[20] Senivongse T., "Enabling flexible cross-version interoperability for distributed services", Proceedings of the International Symposium on Distributed Objects and Applications. IEEE Computer Society, 1999, pp.201-10. Los Alamitos, CA, USA.

[21] Pellegrini, N-C, "Dynamic reconfiguration of Corba-based applications", Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 IEEE Computer Society, 1999, pp.329-40. Los Alamitos, CA, USA.

[22] Barbacci, M., Weinstock, C., Doubleday, D., Gardner, M., and Lichota, R., "Durra: A structure description language for developing distributed applications.", IEEE Software Engineering Journal, pp. 83-94, March 1993.

[23] Bloom, T., and Day, M., "Reconfiguration and module replacement in Argus: theory and practice.", IEEE Software Engineering Journal, pp. 102-108, March 1993.

[24] Magee, J., and Kramer, J., "Dynamic structure in software architectures", Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM Press, pp. 3-14, 1996.

[25] Purtilo, J.M., and Hofmeister, C., "Dynamic reconfiguration of distributed programs", Proceedings of the 11th International Conference on Distributed Computing Systems, pp. 560-571, 1991.

[26] Erwig, M., "Categorical Programming with Abstract Data Types", Algebraic Methodology and Software Technology. 7th International Conference, AMAST'98. Proceedings. Springer-Verlag. 1999, pp. 406-21. Berlin, Germany.

[27] Schmidt RW, "Dynamically extensible objects in a class-based language.", Proceedings. Technology of Object-Oriented Languages and Systems, TOOLS-23, IEEE Computer Society, Los Alamitos, CA, USA, pp. 294-305, 1998.

[28] Tucker, J.V., and Zucker, J.I., *Program Correctness over Abstract Data types, with Error-State Semantics*, CWI Monographs, Amsterdam, 1988.

[29] Lambiri, C., "Temporal Logic Model for Distributed Systems", Master's Thesis, Department of Electrical Engineering, University of Ottawa, Canada, 1995.

[30] Breu, R., *Algebraic Specification Techniques on Object Oriented Programming Environments*, Springer-Verlag, Berlin, Germany, 1991.

[31] Bjorkander, M., "Using SDL to develop CORBA object implementations", in *Formal Methods for Open Object-based Distributed Systems*, Chapman & Hall, pp. 177-192, London, UK, July 1997.

[32] Lerner, B., and Habermann, N., "Beyond Schema Evolution to Database Reorganization", in Proceedings of OOPLSA, ECOOP '90, ACM press, pp. 67-88, Reading, MA, USA, 1990.

[33] Kordale, R., Ahamad, M., and Devarakonda, M., "Object Caching in a CORBA Compliant System", in Proceedings of The Second Conference on Object-Oriented Technologies and Systems, USENIX Association, pp. 65-81, Toronto, Canada, June 1996.

[34] Mowbray, T., "CORBA Design Patterns", in Proceedings of Patterns '98, Object Management Group, London, U.K., February 1998.

[35] Mowbray, T. and Malveau, R., *CORBA Design Patterns*, John Wiley & Sons, Inc., New York, 1997.

[36] Blaha, M., Premerlani, W., and Shen, H., "Converting OO Models into RDBMS Schema", IEEE Software, pp. 28-39, May 1994.

[37] Axmark, D., Widenius, M., DuBois, P., and Aldale K., *MySQL reference manual (version 3.22.22),* http://www.mysql.com, 1998.

[38] Using MATLAB Graphics (Version 5), The Math Works Inc., Natick, MA, USA, December 1996.

[39] Webster's New World Dictionary and Thesaurus, Macmillan Publishing, New York, 1996.

[40] Webster's New World Dictionary of Computer Terms, 7th Edition, Macmillan Publishing, New York, 1999.

[41] Orbix Programmer's Guide (Version 2.3), IONA Technologies PLC, Dublin, Ireland, 1997.

[42] Orbix Programmer's Reference (Version 2.3), IONA Technologies PLC, Dublin, Ireland, 1997.

[43] Goldberg, A. and Robinson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, MA, 1983.

[44] Seigel, J. *CORBA Fundamentals and Programming*, John Wiley & Sons, New York, 1996.

[45] Stallings, W., The Open Systems Interconnection (OSI) Model and OSI related Standards, Macmillan Publishing, New York, 1987.

[46] LaLonde, W., *Discovering Smalltalk. Benjamin Cummings*, Redwood City, CA, 1994.

[47] *ObjectStore C++ API User Guide (Release 5.1),* Object Design, Inc., Burlington, MA, April 1998.

[48] *ISO/IEC* 10746-1 ODP Reference Model Part 1: Overview, ISO/IEC, 1995.

[49] *The Common Object Request Broker: Architecture and Specifications Revision 2.2.* Object Management Group, 1998.

[50] Shirley, J., Hu, W., and Magid, D., *Guide to Writing DCE Applications, 2nd Edtion*, O'Reilly & Associates, 1994.

[51] Sessions, R., *COM and DCOM*, John Wiley & Sons, 1998.

[52] Crawley, S.C. and Duddy, K.R., "Improving Type Safety in CORBA", Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, UK, September 1993.

[53] Skarra, A.H., and Zdonik, S.B., "Type Evolution in an Object-Oriented Database", Research Directions in Object-Oriented Programming, MIT Press, 1988.

[54] Monk, S.R., and Sommerville, I., "Schema Evolution in OODBs Using Class Versioning", SIGMOD RECORD, 22 (3), September 1993.

[55] Evans, H., and Dickman, P., "DRASTIC: A Run-Time Architecture for Evolving, Distributed, Persistent Systems", ECOOP'97 Proceedings, 1997.

[56] *ANSAware 4.1 Application Programming in ANSAware*, APM Document RM.102.02, UK, 1993.

[57] Koscielny, G., and Sadou, S., "Dynamic Reuse of Services in Distributed Systems", Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 29 IEEE Computer Society, 1999, pp.305-318. Los Alamitos, CA, USA.