Improving Polymorphism and Concurrency in Common Object Models

by

Siva Prasadarao Challa

Dissertation submitted to the faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

APPROVED:

Dennis G. Kafura, Chairman

Marc Abrams

James D. Arthur

Lenwood S. Heath

Scott F. Midkiff

January, 1998

Blacksburg, Virginia

Improving Polymorphism and Concurrency in Common Object Models

by

Siva Prasadarao Challa

Committee Chairman: Dennis G. Kafura

Computer Science and Applications

(ABSTRACT)

Most common object models of distributed object systems have a limited set of object-oriented features, lacking the advanced features of 'polymorphism' (an abstraction mechanism that represents a quality or state of being able to assume different forms) and 'concurrency' (the ability to have more than one thread of execution in an object simultaneously). The lack of support for advanced features is a serious limitation because it restricts the development of new components and limits reuse of existing of components that use these advanced features. As a result, wrappers must be used that hide the advanced features or components must be re-implemented using only the features of the common object model.

In this dissertation, a new direction of research centered on a subset of object-oriented languages, specifically statically typed languages, is considered. One of the major drawbacks of existing distributed object systems is that they cater to a broad domain of programming languages including both object-oriented as well as non object-oriented languages. Mapping an object model into a non object-oriented language is a complex task and it does not appear natural to a native language user.

The interoperable common object model (ICOM) proposed in this dissertation is an attempt to elevate common object models (with the advanced features of polymor-

phism and concurrency) closer to the object models of statically typed object-oriented languages. Specific features of the ICOM object model include: remote inheritance, method overloading, parameterized types, and guard methods. The *actor* model and *reflection* techniques are used to develop a uniform implementation framework for the ICOM object model in C++ and Modula-3. Prototype applications were implemented to demonstrate the utility of the advanced features of the ICOM object model.

The main contributions of this dissertation are: design and implementation of a powerful common object model, an architecture for distributed compilation, and an implementation of a distributed object model using the actor model.

ACKNOWLEDGEMENTS

This research would not have been possible without the support and encouragement from several people. First of all I would like to thank Dr. Dennis Kafura for his constant encouragement, guidance, and abundant patience. He always surprised me with his ability to motivate me to continue work on new research issues. I don't think I can ever thank him enough for all his support throughout my association with him.

I would like to thank Dr. Lenwood Heath for dropping a little flyer about a distributed object model in my mailbox several years ago. He laid the stepping stone for my research. I would like to thank him for all the lunches/dinners. He showed a lot of patience to listen to my problems when I was in trouble.

Many thanks to Dr. Marc Abrams, Dr. James Arthur, Dr. Lenwood Heath, and Dr. Scott Midkiff for readily agreeing to serve on my Ph.D. committee when I approached them. I got really good feedback from all of them. I also want to thank Dr. Edward Fox and Dr. Nick Stone for offering research assistantship and part-time programming jobs. I would like to thank Dr. K.V.S.V.N. Raju who was my bachelor's degree advisor and Dr. K.C. Reddy who was my master's degree advisor for their encouragement.

Several friends both at Virginia Tech and other places have helped me to be a normal person. I want to thank Chandra Reddy and Giri Tiruvuri for encouraging me to go for a Ph.D. I also want to thank Sriram Pemmaraju and Praveen Paripati for

their emotional support. I would like to thank all of my bachelors and masters degree classmates and seniors for their friendship. To name a few, Sriram Popuri, Sudhir Machireddy, Krishna Nareddy, Srinivas Kotla, ALN Reddy, and Chandra Kota come to my mind right away.

Throughout my stay at Virginia Tech, I made a number of good friends. I am sure I cannot list all of them here, but I don't think I can live with myself without naming a few. I cannot thank enough the 6500A roommates (Chandra, Gopi, Murali, and Sekhar) who always stuck with me even after leaving Blacksburg. My other friends who deserve a special mention are Prashant, Modan, Nagendra, Karunanidhi, Praharaj, Varma, Anil, Mahesh, Pankaj, and Gurjit.

Several people in the Computer Science department deserve a mention. I would like to thank Randy Ribler, Xiangdong Liu, Jay Wang, Ben Keller, Todd Stevens, Mark Lattanzi, Juvvadi Ramana, Greg Lavender, Krishna Ganugapati, and Ashish Shah for their friendship. The Computer Science department secretaries, Jessie, Tammi, and Jo-Anne deserve a special thanks for their support. Thanks to Dr. Verna Schuetz who has always been very friendly with me.

I would like to thank my parents and relatives for constantly encouraging me to complete the Ph.D. as soon as possible. I want to thank my brother, Venkat Challa, and sisters, Shantha and Sharada, for their support.

Last but not the least, I would like to thank my sweetheart, Madhavi, for tolerating the torture of staying home alone all the time while I was working on the research in school. She listened to my CS blabbering very carefully, even though most of it did not make sense to her. Her proof-reading helped the writing process. Surprisingly, I never felt that I was working hard, even though I spent several hours a day at school. Thank you, Madhavi, for making me feel comfortable and energetic all the time.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Many organizations are moving away from software systems written in a single programming language on monolithic hardware architectures (legacy systems); they are moving towards distributed systems running on a network of heterogeneous computer workstations. These changes are making the application development for distributed and heterogeneous environments inevitable. A challenging problem that arises is mastering the complexity involved in integrating software components written in multiple programming languages. At the same time, integration of software components from legacy systems to work with other components is a serious problem facing software developers.

Object-oriented technology with its encapsulated object structure and the basic communication mechanism of message passing is a congruous solution for the development of applications using software components in distributed and heterogeneous environments. The abstraction and encapsulation facilities provided in object-oriented languages are useful in hiding implementation details and focus on the external behavior of software components. Another aspect of object-oriented programming with reference to the legacy systems is that object-oriented *wrappers* can be built around

existing legacy system components. The wrappers allow the legacy components to be integrated with newly developed components or the legacy components in an application [55]. Thus, object-orientation is proving to be a useful programming paradigm for current software development and for legacy systems.

Integration of software components written in several popular object-oriented languages is a recurring problem in the software development process. Because there is no single universally accepted object-oriented language, software is developed in different languages that have vastly different object models. These differences force the existing code to be re-written in multiple languages. This duplication impedes software reusability and maintainability. So, there is a strong necessity for integration of software components written in multiple languages.

Even though the Java [12] language is treated by some as a single universal language that can be used for software development, it is not a solution for the problems facing the software development in distributed and heterogeneous environments. Java supports communication between objects distributed in different address spaces, using Remote Method Invocation (RMI) [124]. But, RMI is specifically designed to operate only in the Java environment. It cannot be used effectively with other languages [22]. Java is still an evolving language, undergoing changes in the facilities it provides. It is not available on all platforms, and moreover, it is an enormous task to translate software developed in several popular languages into Java. So, integration of legacy components existing in other languages to work with Java components is currently under investigation [21, 60]. In summary, in order to reuse the existing software effectively, software modules built in multiple object-oriented languages need to interoperate with one another.

1.1 Interoperability

Interoperability is the ability of the components of a software system to coexist, communicate, and cooperate with one another to achieve a common goal in a distributed and heterogeneous environment involving disparate implementation languages, operating systems, and hardware platforms. Integrating existing software components written in different programming languages is becoming a common practice in the software development process because it reduces code duplication and results in better reuse and maintenance of software.

Many common object models [20, 60, 70, 110, 117] have been proposed to support interoperability of software components developed in multiple languages. The systems based on these models try to encapsulate the communication mechanisms used for the transfer of messages between objects in a distributed environment. They provide translations of object descriptions described in Interface Definition Language (IDL), into different languages. Most of the systems provide language transparency (an application developer can act as if all application components are available in the native implementation language) and distribution transparency (an application developer can act as if all application components are available locally). Several systems have mappings supporting both object-oriented and non object-oriented languages.

1.2 Problem statement

Most of the common object models, including the popular industry standard models — the Common Object Request Broker Architecture (CORBA) [60] and the Component Object Model (COM) [20]— do not support certain useful and advanced features of 'polymorphism' and 'concurrency'. These models are limited in the object-oriented

features supported by them. Although the features of polymorphism and concurrency are present in several object-oriented languages and have been proven to be very useful in the application development process, they cannot be used for interoperability of software components because they are not part of the existing common object models. Many popular object-oriented languages distinguish themselves with the support for advanced object-oriented features. The lack of support for these features in the existing common object models is a serious limitation because, it restricts the reuse of software components developed using these advanced features.

Polymorphism is an abstraction mechanism that represents a quality or state of being able to assume different forms. Several categories of polymorphism exist in object-oriented programming languages. But not all of them are present in common object models. The categories of polymorphism not present in the existing common object models are as follows:

Inheritance of abstract methods in distributed environments

This feature supports inheriting and overriding in a derived class, of the abstract methods defined at a base class. An invocation of an overridden abstract method at the base class results in the execution of the overriding method in the derived class. The base class and the derived class may exist on different nodes in a distributed environment, possibly in different languages. This type of polymorphism is also referred to as *inclusion polymorphism*.

Method overloading

Method overloading involves the existence of more than one method with the same name but different signatures[1] in the interface of an object. The object can be a distributed object with its components existing in multiple places. A

[1] A signature of a method is its argument types and the result type.

method can be overloaded among the distributed components of the object.

Parameterized types

Parameterized types involve the abstraction of a generic type based on the common features of existing types. They allow a general concept to be instantiated with one or more types.

Concurrency has different interpretations in different contexts. In this dissertation, *Concurrency* in an object-oriented system is its ability to have multiple threads of control. It can be achieved by interleaving or simultaneous processing of method invocations. Concurrently executing threads can cooperate with one another through *synchronization*, to achieve a common task. The advanced features of concurrency not present in the existing models are[2]:

Atomicity of method executions

The atomicity feature requires completion of the processing of a method invocation on an object, before that object can start the processing of another method invocation.

Synchronization of the method invocations

Synchronization features support accepting, rejecting, and deferring of the method invocations on an object, based on the object's state information and the arguments of the method.

Thus, the existing common object models lack advanced object-oriented features and need to be elevated closer to the object models of statically typed object-oriented languages.

[2]CORBA supports concurrency via CORBA services, but it is not part of the CORBA object model, and is more oriented towards transaction processing, not general applications. COM also supports concurrency via its message filtering mechanism, but it is more oriented towards compound documents, and is not part of the object model.

1.3 Goal

The major goal of this dissertation is to investigate aspects of a powerful common object model which supports the advanced features of polymorphism and concurrency.

The categories of polymorphism considered are 'inheritance of abstract methods in distributed environments', 'method overloading' and 'parameterized types'. Support for inheritance of abstract methods ensures that appropriate overriding methods of derived classes are bound to the method invocations on the corresponding overridden methods of the base class, even in distributed environments involving multiple languages. The method overloading feature allows multiple methods with the same name but different signatures to exist in the same class. It also allows some methods in the superclass and some methods in the derived class to have the same name but different signatures. The parameterized types supported are 'bounded parameterized types' which require the interfaces of parameters to be specified explicitly along with the interfaces of the parameterized types.

The features of concurrency to be considered are 'atomicity' of method execution and 'synchronization' of method invocations. Atomicity allows any number of external invocations on an object, but it requires the processing of only one method invocation at a time. In the case of distributed objects, atomicity allows external invocations on any distributed component of the object, but it guarantees the completion of the processing of an external invocation before starting the processing of another external method invocation. The synchronization feature allows for delaying of message processing, based on the state of the object and message contexts. The problem that arises by supporting inheritance of classes that use synchronization is known as the 'inheritance anomaly' [113]. The inheritance anomaly problem is addressed in this dissertation.

The feasibility of the model is shown by the development of an implementation framework. The framework supports the polymorphism and concurrency features considered above. It provides a uniform design for translating the common object model features into the languages C++ and Modula-3. The utility of the model can be demonstrated by the implementation of prototype applications in a distributed and heterogeneous environment involving software components developed in multiple object-oriented languages.

1.4 Approach

The motivation for selecting the domain of languages used for implementing the proposed common object model is described in section 1.4.1. The new concepts introduced and software tools used are briefly discussed in section 1.4.2.

1.4.1 Motivation

A new direction of research centered on a subset of the object-oriented languages is chosen in this dissertation. One of the main reasons why the existing common object models do not include the advanced features of polymorphism and concurrency is the fact that current object-oriented systems for interoperability and component-based reuse employ object models that are weakened by the attempt to provide accessibility from too broad a domain of programming languages [27]. These common denominator models are limited in their support for object-oriented features. They support only a subset of object oriented features because mapping of a more complete object model into a non-object oriented language is not manageable. For example, method overloading is not provided in Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [60] object model, because dealing with

overloaded names is too difficult or very awkward to handle in typical non-object oriented languages. Powerful features missing from this and many other models include: parametric polymorphism and concurrency. The limitation on the interoperability and component reuse implied by such weakened object models restricts the availability of services to end-users. It restrains software productivity as it forces the rebuilding of existing software components.

Statically typed object-oriented languages are frequently used in building systems for interoperability. Table 1.1 shows a number of popular statically typed object-oriented languages and some important features supported by them. The method overloading feature is not directly supported in Modula-3 and BETA. But, the other features of these languages such as 'runtime type checking' are used to emulate the method overloading feature. The generic types feature is proposed in the Java language as an extension citeagesen:java, myers:java, thorup:java. The concurrency feature is not directly supported in C++ and Eiffel, but standard libraries exist for these languages to use concurrency.

Table 1.1: Statically Typed Object-Oriented Languages Versus Important Features Supported

Language	*Features*		
	Method Overloading	Generic Types	Concurrency
Ada 95	√	√	√
BETA	*weak*	√	√
C++	√	√	*library*
Eiffel	√	√	*library*
Java	√	*proposed*	√
Modula-3	*weak*	√	√

Static typing offers both a degree of safety to applications and is critical in defining the interfaces of remote objects and the data that is exchanged between client

and server objects. This dissertation concentrates on statically typed object-oriented languages. The languages selected are C++ [166] and Modula-3 [133]. Though these languages fall within the same family of statically typed languages, they are similar enough to yield a powerful object model, yet different enough to make the problem interesting, and widely enough used to create interesting opportunities for reuse (e.g., a Modula-3 animated user interface to a C++ digital library).

1.4.2 New Concepts

In this section the new concepts introduced in the ICOM object model and its framework are described.

To provide inheritance of abstract methods in a distributed environment, a *truly distributed object* is defined. A truly distributed object is a distributed object whose state and method information can exist in multiple processes implemented in different languages on multiple machines, with a constraint that the various components of the object are related with one another using a 'superclass' and 'subclass' relationship. Truly distributed objects form a subset of a more generic form of distributed objects, called *split objects* [187]. A split object is a distributed object whose components are arbitrarily placed in different processes.

A new concept of *distributed compilation* is introduced to support 'truly distributed objects'. A distributed compiler, when invoked on an IDL description involving truly distributed objects, transfers information among translators at different sites. The existence of distributed objects with abstract methods requires communication between interacting components before the actual invocation of methods, so that each component is aware of the existence of the other component. In the case of statically typed languages, the communication among translators must take place

before execution time. So, a distributed compiler makes this interaction possible by transferring IDL descriptions, and invoking IDL-to-native language translations at remote sites.

1.4.3 Software Tools and Design Criterion

The software tools used to implement the ICOM object model, and the design criterion chosen are described in this section.

The actor model [5], with its asynchronous message passing, is an excellent tool for communication in distributed environments. It can be used as a basic framework for implementing the common object model. In this dissertation, the basic actor model is extended to support the concept of a 'distributed actor'. The actor integrity as a distributed entity is preserved by ensuring the completion of execution of a method invocation before the next message intended for that actor is processed.

A uniform framework for the proposed common object model, based on the actor model of communication is developed using reflection-like techniques [102]. This framework allows the addition of the advanced features mentioned in previous sections in a systematic manner. This framework can be translated into any statically typed object-oriented language that supports the common object model features directly. The framework can also be translated into any statically typed language that supports the set of features identified in this dissertation.

A major design criterion observed in this dissertation is that no changes to the existing operating systems and compilers should be required. Solutions involving translations to some intermediate low level language are not followed. Because of the mechanical nature of the translation only hand coded translations are used for translations, although an implementation of such a translator is technically feasible.

1.5 Outline

This section describes the outline of each chapter of the dissertation. In Chapter 2, the existing interoperable systems are described. A taxonomy of existing interoperable systems is provided. Distributed object systems that have object models similar to the CORBA object model are described in detail. The drawbacks of these systems are identified.

Two applications demonstrate the utility of the framework. A digital library application is developed that uses the common object model framework. A system for a financial institution that owns and monitors a set of national level *grocery* stores and *clothes* stores is implemented. This sytem uses the full power of the proposed common object model features. For each example discussed throughout the dissertation, an IDL description, the translation of the IDL description into a native language, and the diagrammatic sequence of method invocations are provided.

In Chapter 3, a basic framework of the proposed interoperable common object model (ICOM) is illustrated. The example problem of 'Department Stores' is used to explain the concepts of the framework that supports the ICOM model. This problem is used throughout the dissertation. The object model discussed in this chapter is the basic object model on top of which the advanced object-oriented features can be added.

In Chapter 4, the basic polymorphism in the ICOM object model is explained in detail. This polymorphism is also referred to as the inclusion polymorphism. An example from the 'Department Stores' problem that uses truly distributed objects is explained.

In Chapter 5, the advanced polymorphism of the ICOM object model is described. This polymorphism includes method overloading and parameterized types. Examples

from the 'Department Stores' are used to explain the various concepts.

In Chapter 6, the concurrency features of the ICOM object model are illustrated. The features of atomicity of processing of a method invocation and guard methods are described in detail with the examples from the 'Department Stores' problem.

In Chapter 7, a summary of the research is provided along with future research directions.

In Appendix A, syntax and semantics of the ICOM object model are described.

A Glossary of the terms used in this dissertation follows Appendix A. It is followed by a References section and an Index.

1.6 Summary

This chapter introduced the problem addressed in this dissertation, The 'problem statement' section identified the lack of support for certain advanced object-oriented features in the existing common object models. The 'goal' section described the major goal of developing a powerful common object model to help better interoperability of different object-oriented languages in a distributed and heterogeneous environment. The 'approach' section explained the motivation for selecting statically typed object-oriented languages for implementing the proposed common object model (ICOM) framework, the new concepts defined, and the software tools used for the implementation of the ICOM framework and the design constraint observed to achieve the goal. Finally, the 'outline' section briefly described the contents of each chapter.

Chapter 2

Interoperable Systems

2.1 Introduction

Providing interoperability among programming languages has been a challenging problem for several years [42]. Earlier attempts to address interoperability were directed towards non-object-oriented languages [57, 65, 153, 185]. Several solutions exist for direct mapping between languages when only two languages are involved [46, 115]. Though these solutions are adequate for two languages, they become very complex when the number of languages used is more than two. When N languages are involved, they require N*(N-1) translators, with every language requiring translators to every other language. An addition of an (N+1)st language requires N translators to be written. Also, the differences in the models of the languages makes the translator writing very difficult.

A classification of existing interoperable systems is illustrated in Figure 2.1. Some of the example systems are listed under each category in the figure.

There are two major approaches to address the interoperability problem - the

common runtime approach and the *protocol based* approach. The common runtime approach has been applied in a shared memory and distributed shared memory environment whereas the protocol based approach has been applied in a distributed environment.

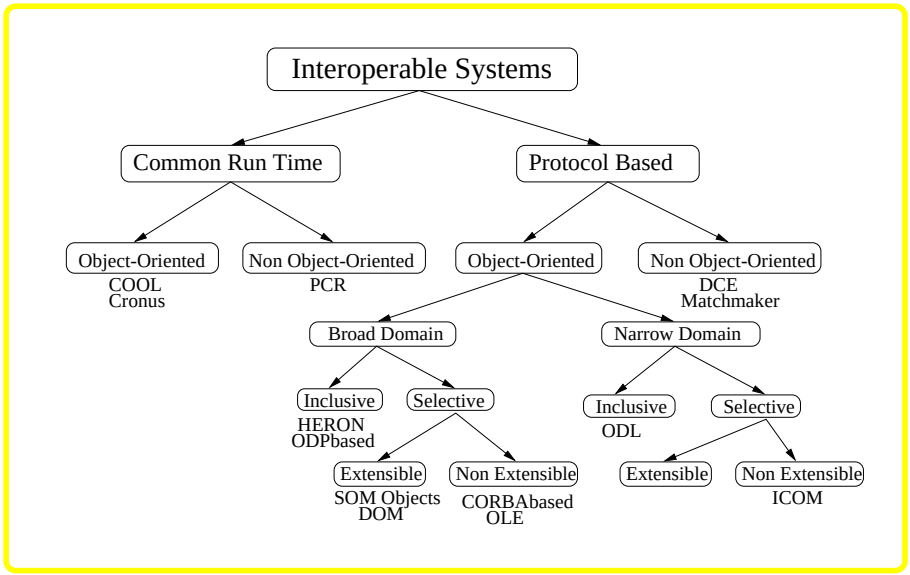

Figure 2.1: Interoperable Systems Classification

The common runtime approach provides a software layer over existing operating systems that acts as an intermediary between the application components and the operating systems and provides services similar to the operating system services. The interaction between software components takes place through 'upcalls' and 'downcalls'. These systems are briefly discussed in section 2.2.

The protocol based approach employs an intermediate language typically called the *interface definition language* (IDL) that is used to describe different software components. There are several ways to define an IDL. It can be a common denominator of all features (of relevant languages) or a union of all features or a variation of both.

Systems based on the protocol based approach provide the necessary communication mechanisms required for communication between various software components. In this dissertation we concentrate on the protocol based approach.

Table 2.1: Existing Systems Versus Languages Supported by Them

Language	*Existing Models*									
	PCR	COOL	Cronus	ILU	CORBA	SOM	COM	RISC	ODL	HERON
C++		√		√	√	√	√	√	√	
Java				√	√					
Modula-3				√				√		
Smalltalk					√				√	
Ada95					√					
CLOS									√	
Eiffel										√
ObjectiveC									√	
C	√		√	√	√	√	√	√		
LISP	√		√	√						
Python				√						
Ada			√							
Fortran			√	√						
COBOL					√					
Cedar	√									
Scheme	√									

A tabular representation displaying various existing interoperable systems and the languages supported by them is shown in Table 2.1. The top portion of this table displays object-oriented languages and the bottom portion shows non-object-oriented languages.

A second tabular representation showing various existing interoperable systems and the architectures supported by them is displayed in Table 2.2. Among these systems, SOM is a CORBA compliant system. Several other systems exist that are CORBA compliant [144, 122, 174], but they are not shown here.

Table 2.2: Architectures (Common Run Time and Protocol Based) Versus Existing Models

Architecture	*Existing Models*									
	PCR	COOL	Cronus	ILU	CORBA	SOM	COM	RISC	ODL	HERON
Common Run Time	√	√	√							
Protocol Based				√	√	√	√	√	√	√

2.2 Common Run Time Systems

Common Run Time systems [91, 185] have a run time layer over existing operating systems. The run time layer provides services that can be used like operating system services. It acts as an intermediary between the interacting entities using *down-calls* and *up-calls*. A down-call is a request from the client to the common run time and an up-call is a request from the common run time to the server. Both shared memory implementations [185] and distributed shared memory implementations [91] of the common run time systems are available. These systems support both object-oriented as well as non-object-oriented models.

Portable Common Runtime (PCR) [185] from Xerox is an example of a non-object-oriented common runtime system. The PCR system provides *closely coupled interoperability* between programs written in different languages in a shared memory environment. Closely coupled interoperability is the ability to share data structures, memory, and threads of control between program components of an application. Facilities provided in the PCR system are: a shared address space, symbol binding, light weight threads, and garbage collection. The PCR system implementation supports the languages C, Cedar, Scheme, and Common Lisp.

The Chorus Object-Oriented Layer (COOL) [91] is an example of the common run time approach. COOL is an extension of the facilities provided by the Chorus

distributed operating system. It provides a set of generic services that co-exist with an operating system's services. These generic services are divided into three functionally separate layers, the *COOL base layer*, the *COOL generic run time layer* and the *COOL language-specific run time layer*. The COOL base layer provides memory abstractions where objects are stored. The COOL generic run time layer implements a variety of object services including object creation, dynamic linking and loading, and fully transparent invocation of an object's methods. The language-specific run time maps a particular language object model to the generic run time model. Currently mappings to C++ are supported in COOL. Recently, there have been attempts to make COOL a CORBA compliant framework [72].

Though the common run time approach seems promising, it is not popular because of the existence of an additional run time layer and primitive (i.e., many object-oriented features are not supported) object models. The common runtime systems are not discussed further.

2.3 Protocol Based Systems

The models supported by the protocol based systems are classified into object-oriented and non-object-oriented models. The early protocol based systems use non-object-oriented models. Most of the recent systems use object-oriented models.

2.3.1 Protocol Based Non-Object-Oriented Systems

Protocol based non-object-oriented systems use remote procedure call abstraction. Systems supporting non-object-oriented models [57, 65, 76] provide a common interface definition language to describe a server's interface. The description is compiled to produce stubs for *Marshalling* and *Unmarshalling* needed during a remote call

from a client and return from the server. Marshalling is the process of building up a sequence of bytes that contains the values of the arguments or results in a proper format. Unmarshalling is the reverse process that recovers values from the sequence of bytes. When a remote call is made, its arguments are marshalled, sent as a part of a message over the network, and unmarshalled at the other side before passing them to the called procedure. Similarly the results are marshalled, sent, and subsequently unmarshalled. These protocol based non-object-oriented models were limited to using remote procedure calls. Higher abstractions over remote procedure calls are not possible in these models. Protocol based non-object-oriented systems are not discussed further.

2.3.2 Protocol Based Object-Oriented Systems

Recently, several systems [20, 60, 70] following the protocol based object-oriented approach have been developed. Such systems are also called *Distributed Object Systems*. An interoperable system following a protocol based architecture is responsible for interactions between client and server objects. It provides mechanisms required for finding the object implementation for a request, preparing the object implementation to receive the request, and communicating the data making up the request. The interface that the client sees is completely independent of where the object implementation is located, and in what programming language it is implemented.

The protocol based systems achieve interoperability using an intermediate language such as an Interface Definition Language (IDL). An IDL defines the types of objects by specifying their interfaces. An interface consists of a set of named operations and the parameters to those operations. IDL is the means by which a particular object implementation informs its potential users what operations are available and

how they should be invoked. From the IDL definitions it is possible to map object interfaces to particular programming languages. Different object-oriented languages may prefer to access interoperable objects in different ways. The language mapping includes definition of the language specific data types and interfaces to access objects.

In a distributed object system, both client and server object developers need to understand the common object model along with their own native language model. The distributed object system provides IDL translators that can translate IDL descriptions of server objects into a native programming language. The IDL translations use a *proxy class* approach to represent a server object at a client site. A proxy class is a class that can be used to instantiate a *proxy object* at a client site. A proxy object is a surrogate object to a server object. It supports the functionality of the server object, but it simply delegates all method invocations on it to the server object. For a client object, the proxy object appears as a real server object.

To develop a distributed application, the client and server development can be done independent of each other. While implementing a server, a developer needs to:

- describe the server functionality in IDL,
- use an IDL-to-native language translator to generate code,
- implement the server functionality in the native language,
- link the native language code with the translator generated code, and
- start the server process to accept requests from the client objects.

To use a remote server object, a client developer needs to:

- use an IDL-to-native language translator on the server description written in IDL,

- implement the client functionality in the native language using the translator generated code and the server proxy classes,
- link the native language code with the translator generated code, and
- run the application.

The architecture of a typical distributed object system is shown in Figure 2.2. The figure illustrates a distributed application that has a client object at a Local node and a server object at a Remote node[1]. This architecture has three major layers: the *translation layer*, the *intermediate layer*, and the *communication layer*. The translation layer represents the compilation phase of the IDL descriptions into native languages. The IDL translators generate different code for a client and a server for the same IDL description. The translators at the client side generate interface files, and a *proxy* class for each server object; they are shown as solid arrows from the translation layer into the intermediate layer. The IDL translations are represented by thin solid arrows that start from the IDL descriptions shown in the white region of the layer and come out of language specific translators, which are represented as inverted domes. The server specific code includes interface files that can be used by a server application code and a *message handler* class implementation whose instance handles all external invocations on the server object.

The intermediate layer exists between the communication layer and the translation layer. The client code, the translator generated code, and the server code are present at this layer. The client specific code generated by the translator contains the native language representations of the interface of the server object and implementations of

[1]In the context of this dissertation a node represents an abstract computer in the distributed system. A single computer as a whole can represent a node. Multiple nodes can possibly exist on a single machine but run in different processes that have their own address spaces.

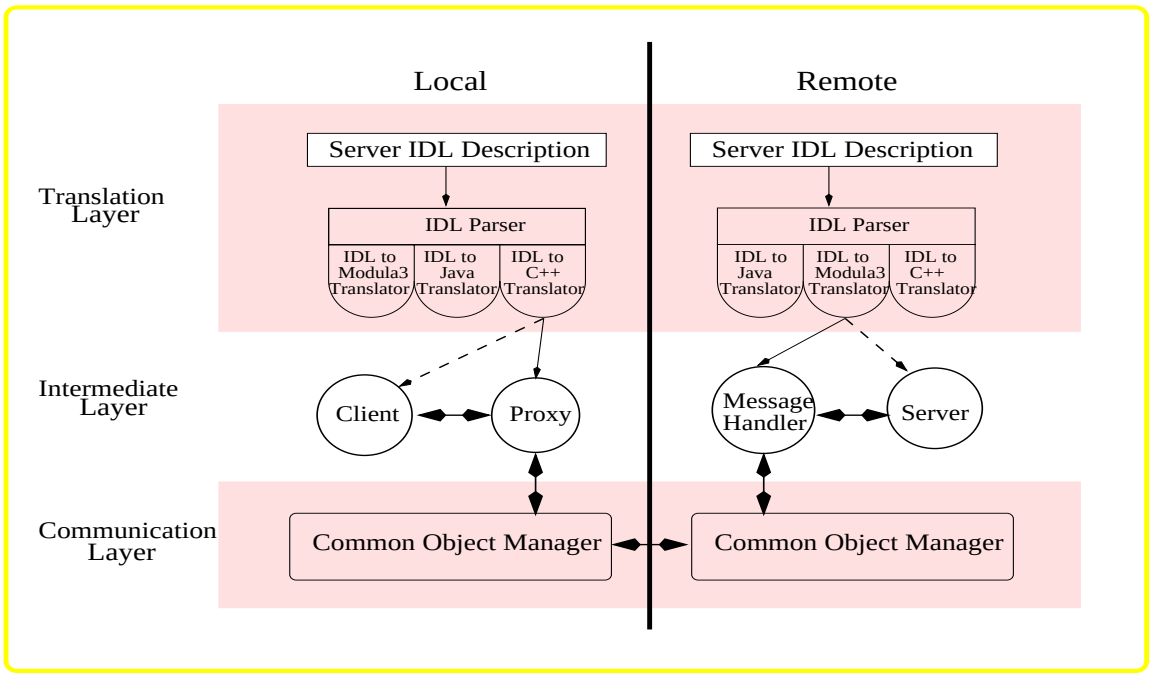

Figure 2.2: Architecture of a Typical Distributed Object System

the proxy objects. The interface files used by the client and the server at the local and remote nodes is shown with the dashed arrows. A client application knows only the logical structure of a server object according to its interface and experiences the behavior of the object through invocations. Although a client is considered to be a program or process initiating requests to an object, it is called a client with respect to to a particular server object. A client of an object may be a server of other objects. At the server side, the server specific translation includes the interface files that can be used by the server implementor and message handler class implementations that handle external invocations meant for the server objects.

The communication layer is responsible for all communication required between a client and a server. The 'Common Object Manager' components in the diagram

represent the communication layer. The communication between client objects and server objects is a peer to peer communication, i.e., a server can be a client of another server. The communication path between a client and a server is shown using thick and solid bi-directional arrows. It is the communication layer's job to prepare a server for communication, deliver an invocation message from a client to a server and return results. Typically, encoding and decoding of method invocations and results is automatically handled by the communication layer. In some systems, the encoding and decoding is done at the intermediate layer.

When an invocation occurs on a proxy at the client side, the proxy object representing the server object hands the call to the underlying common object manager. The common object manager delegates that call to the appropriate method of the implementation at the server side. When the method completes execution, the common object manager at the server side returns results or exceptions back to the client via the common object manager at the client side.

An example of an IDL description similar to CORBA IDL is given in Figure 2.3. This declaration defines an interface to a clothes store object that can be any retail clothes store. It has two methods monthlySales and quarterlySales. The method monthlySales accepts two input parameters month and year, and returns an output parameter amount representing the monthly sales at a particular retail store. Similarly, the method quarterlySales accepts two input parameters quarter and year, and returns an output parameter amount representing the quarterly sales at a retail store.

In the figure, the keywords of IDL are shown in **bold face**. There are two types of parameter passing modes shown, in and out. They represent 'input' and 'output' parameter passing modes respectively. Note that the figure does not show any return values for the methods. The parameters of the methods can be either 'basic' types or 'user defined' types. The user defined types are passed by reference only.

```
interface ClothesStore {

   // returns quarterly sales at a clothes store through
   // the output parameter 'amount'
   quarterlySales(in int quarterNumber;
                  in int yearNumber;
                  out double amount);

   // returns monthly sales at a clothes store through
   // the output parameter 'amount'
   monthlySales(in int monthNumber;
                in int yearNumber;
                out double amount);

   ⋮
}
```

Figure 2.3: IDL Description of a Clothes Store

This interface declaration when translated to a specific language becomes a type specific to that language. For example, in C++, this interface is usually translated to a class. An application program can use the newly generated class to create instances of retail stores and invoke monthlySales and quarterlySales methods on them. The implementation of the generated class corresponding to the ClothesStore interface can exist either locally or remotely. The underlying system communicates the invocation messages to the place where the implementation exists.

2.3.3 Classification of Protocol Based Systems

Systems using protocol based architectures follow one of two approaches: *broad domain* or *narrow domain*.

In the broad domain approach, a language neutral object model is formed across languages from different programming paradigms such as procedural, functional, and logical paradigms. This set of languages may include both object-oriented languages and non-object-oriented languages. This approach defines a language neutral object

model which is typically supported by the object-oriented languages. The model is "mapped" into the non-object-oriented languages (every feature of the model is translated into the target non-object-oriented languages).

The object models of the systems following the broad domain approach can be sub-divided into two categories: *Inclusive Models* and *Selective Models*.

In inclusive models, an extensive set of object-oriented features (usually a union of the object-oriented features of all languages) is included in the model. The HERON system [188] and the Open Distributed Processing (ODP) [141] from International Standards Organization (ISO) and Consultative Committee on International Telephony and Telegraphy (CCITT) fall into this category.

In selective models, a common denominator of all object-oriented features of the participating object-oriented programming languages is taken as the object model. This model is mapped into the non-object-oriented programming languages supported by the system. It is supported directly by the features of the object-oriented languages as the language model is a superset of the common object model. Communication between the *client*, the object requesting a service, and the server, the object providing the service, takes place through a common object model framework.

The models defined under the selective models can be partitioned into *extensible models* and *non-extensible models*. In non-extensible selective models, a common denominator object model is introduced and all interacting languages support this model. Though these models were good initial solutions to the interoperability problem, they do not provide a satisfactory long-term solution, because the features that are initially not part of the model cannot be included in the model. Example systems include COM [20], CORBA [60], and ILU [39]. The selective models that are extensible, consist of a "core object model" whose few basic features are specialized or combined using reflection techniques (or special translations) to incorporate other

object-oriented features. This core object model is mapped into non-object-oriented programming languages. Example systems include SOM [70] from IBM, and RISC [110] from GTE laboratories. Reflection techniques are discussed in the next chapter.

The narrow domain approach considers only a subset of languages, in particular, object-oriented languages. A common object model is derived from the object models of these languages and all interoperating languages support this model. Object models of the systems following this approach can also be divided into *Ielective Models* and *Inclusive Models*. There has not been any research done in the Selective Models so far. This dissertation aims at developing a powerful common object model using this approach. The Object Description Language (ODL) from the University of Utah falls into the category of Inclusive Models.

2.3.4 Examples of Protocol Based Models/Systems

Several common object models following the protocol based approach have been defined and systems supporting the models are developed. In the following subsections, a few relevant systems are briefly described. These systems are listed in Figure 2.1

2.3.4.1 HERON

The HERON system [187, 188] from the Free University of Berlin focuses on the Eiffel programming language's object model as the common object model. In this system, the object model of Eiffel is treated as the most versatile object model that encompasses all object-oriented features.

A notable exception to the classification of the HERON system under Protocol based systems is that this system does not use an IDL as a specification language for the interfaces of the server objects. Instead, HERON uses a configuration manage-

ment approach to distributing the interfaces in which every participating node in the system has a configuration file that keeps track of the implementation locations of the server objects. When processed, these configuration files are translated to proxy classes that can be used by the application developers.

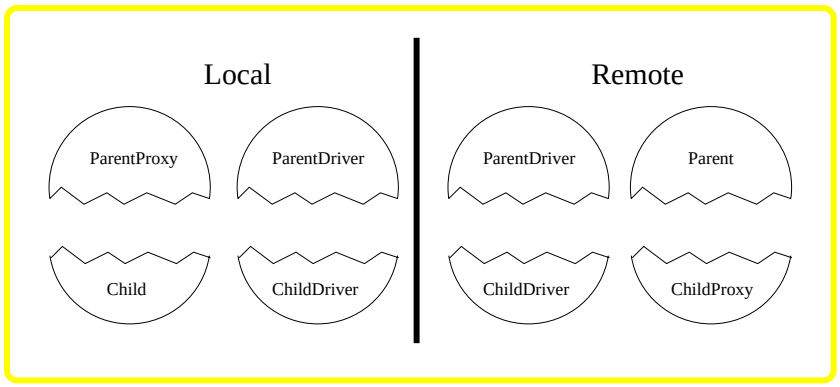

Figure 2.4: Remote Inheritance in HERON

Among all existing systems, HERON is the only system that supports remote inheritance of abstract methods in distributed objects. For example, consider a distributed object with superclass part Parent and a subclass part Child. If Child is implemented on a local site and parent is implemented on a remote site, the HERON system generates *proxy* and *driver* classes on both sides as shown in Figure 2.4. The inheritance structure is preserved both in the *proxy* and *driver* classes. The Child at the local site inherits from the ParentProxy class, the ChildDriver class inherits from the ParentDriver class both locally and remotely. The ChildProxy class on the remote site inherits from the Parent class. Any method invocations that refer to the Parent's methods, on an instance of the Child are delegated to the parent site via the ParentDriver instance. On the Parent site they will be received by the ParentDriver instance and eventually reach the Parent implementation. This scheme also works

fine for overridden virtual methods at the Child class. An invocation of an overridden method at the Parent site is delegated to the Child site via the ChildDriver instances at the Parent and Child sites.

In the HERON system, method invocations are synchronous. Any client object after invoking a method of a remote server object blocks and waits for the results. This is a limitation because the system does not take advantage of the latencies involved in message transmission. There is no support for concurrency.

HERON system was intended to interoperate with any language. A 'Turbo Pascal' implementation was attempted to show the heterogeneity of the system. But, it revealed the drawbacks of trying to map the model to a non-object-oriented language. It was concluded that it is best to use only the features common to the Eiffel object model and the participating language's object model [50].

One of the drawbacks of the HERON system is that it does not support concurrency and parameterized types. Also, an application developer needs to be aware of all participating distributed nodes for the application and maintain appropriate configuration files information.

2.3.4.2 Inter-Language Unification

The Inter-Language Unification (ILU) system [39] from Xerox is an example of a common object model that follows an selective approach under broad domain, object-oriented protocol categories. The object model in the ILU system is non-extensible. The ILU system, working in a distributed, heterogeneous environment aims at smoothly integrating modules written in different languages. An object's interface is described in ILU system's Interface Specification Language (ISL). For each of the programming language supported by the ILU system, a version of the interface in that language can be generated. The ISL compiler generates stubs for method

binding and calling.

An ILU interface can declare types, exceptions, and constants. With respect to a particular ILU object, a module is called the server if it implements the methods of that object, or a client if it calls, but does not implement, the methods of that object. Each ILU object has addressing information called a *string binding handle*. The ILU system uses a name server called the *name-and-maintenance server*, which allows servers to register instances on a net-wide basis. The ILU system's object model provides single inheritance. This system provides the notion of sibling objects: two instances are siblings if their methods are handled by the same server. A simple form of garbage collection is provided for ILU objects. Exceptions are also a part of the ILU object model. The ILU system supports C, C++, Modula-3, Python, Perl5, Java, and Common LISP.

2.3.4.3 Component Object Model

Microsoft's Object Linking and Embedding 2 (OLE 2) [20] is a second example of selective models similar to the ILU system. It defines the Component Object Model (COM) which specifies a programming language independent binary standard for object implementations. Any object conforming to this standard is a legitimate 'Microsoft Windows Object' (hereafter referred to as 'Windows object'), independent of the language used to implement it.

An *interface* is a named set of semantically related functions implemented for an object. An *implementation* of an interface is an array of pointers to functions. Any code that has a pointer to an interface can call the functions in that interface. A Windows object implements one or more interfaces, i.e., provides pointers to function tables for each supported interface.

Users of objects always obtain and act through the pointers to object interfaces;

users never obtain pointers to the underlying object itself. OLE 2 defines a standard function, called QueryInterface, through which the user of one interface of an object can obtain a pointer to another interface of the same object. QueryInterface is part of an interface called IUnknown, which defines a group of fundamental functions that all windows-objects support. All other interfaces in OLE 2 are derived from IUnknown, so all interfaces contain the QueryInterface function. Thus, navigation is always possible between the interfaces of a given object. The OLE 2 object structure resembles the C++ object structure.

Polymorphism is supported in the COM model in the sense that the same message can be sent to any interface that supports the requested operation. Every interface has a unique identifier called the interface identifier.

Support for interface inheritance is present in the COM model. The COM model does not have inheritance at the implementation level, *containment* and *aggregation* mechanisms are used instead. Containment means that a "subclass", sub, of a class super is a simple user of super, and super need not know about its use within sub. Aggregation means that a "subclass" sub of a class super directly exposes super's interface. Aggregation requires super to "know" that its interface is exposed for something other than itself, such that the QueryInterface function behaves as a user expects.

2.3.4.4 Common Object Request Broker Architecture Object Model

The Common Object Request Broker Architecture (CORBA) [60] from the Object Management Group (OMG) is another example of common object models following the selective approach under broad domain, object-oriented protocol based categories. The CORBA object model is non-extensible. It addresses two key issues: (1) a way to define the interfaces between (possibly remote) objects and (2) a mechanism for

conveying requests and responses between these objects. In its current form CORBA specifies a minimal architecture for distributed object management. It includes five interfaces dependent on the object request broker (ORB), offering applications and objects access to the architecture's functions.

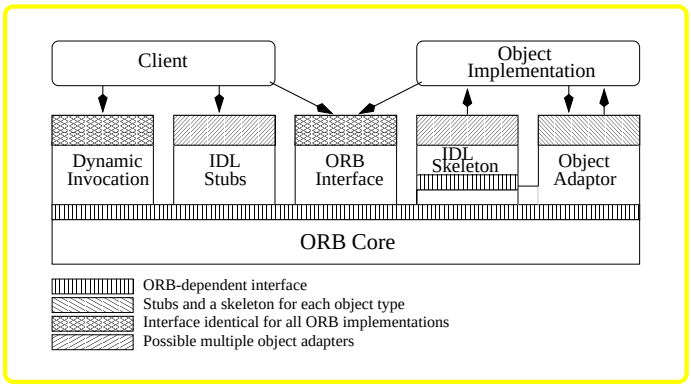

Figure 2.5: CORBA Architecture

A *client* is an entity that wishes to invoke an operation on a target object via the ORB. The object *implementation* comprises the code and data that realize the target object's behavior. The ORB *core* locates an object implementation for a request, ensures that the object implementation is ready to receive the request, and transmits the request, data and results between the client and the target object. The CORBA interface definition language (IDL) describes the operations and associated attributes of an object interface in language and system independent terms. While IDL is independent of any programming language it is intended to support different programming languages through convenient language mappings to CORBA objects. The *static invocation interface* is supported by the stub routines compiled from the IDL interface specifications. Using the *dynamic invocation interface*, a client can name the request's target object and call on the ORB core services to add the

required arguments to the request. An *interface repository* supports the dynamic invocation interface. It stores objects representing IDL information in a form used at run time. CORBA *object adapters* provide two services. First, they provide the main interface through which object implementations invoke the ORB core services. Second, they augment the basic CORBA object model by implementing support for richer object features. CORBA has mappings to C, C++, Java, COBOL, Smalltalk, and Ada 95 languages.

2.3.4.5 System Object Model

System Object Model (SOM) [70] from IBM is an example for extensible systems under the selective approach following the broad domain, object-oriented protocol based categories. It allows languages to share class libraries independent of the language used to implement the class libraries. The SOM object model can co-exist with the language's object model in which it is used. This model consists of an IDL (with compiler), a run time environment with procedure calls and a set of enabling frameworks.

The SOM object model consists of: an *object* that is an object-oriented programming entity having behavior and state (behavior refers to the object's methods and state refers to its data values), an object's *implementation* which is determined by the procedures that execute its methods and by the instance variables, an *object type* that represents the interface through which an object can be communicated with, and a *class* that defines the implementation of objects. Each object of a given class is called an instance or instantiation of the class. SOM supports multiple inheritance. In SOM, a class (also called a *class object*) is treated as an object that has its own methods, interfaces and is defined by another class. A class that defines the implementation of class objects is called a *metaclass*. A metaclass defines methods that

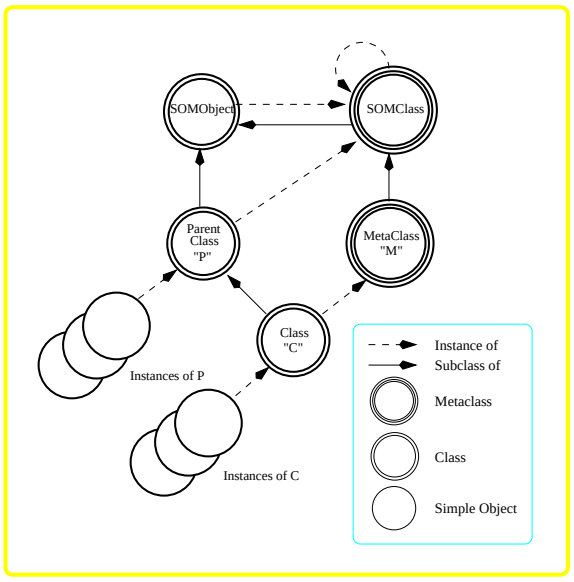

Figure 2.6: SOM Class and Instance Relationships

a class object responds to. Metaclasses can be derived from parent metaclasses, in order to define new functionality for classes.

In SOM, there are three types of *method dispatching* schemes, namely, offset resolution, name-lookup resolution, and dispatch-function resolution. Offset resolution is the fastest and it uses a *method token* (obtained from a global data structure) as an *index* into the receiver's *method table*. Name-lookup resolution is slower than offset resolution and it uses a name-based search in the method table. Dispatch-function resolution is the slowest of the three dispatching schemes and it allows the user to define an arbitrary method resolution mechanism. The offset resolution scheme is done at compile time whereas the other two schemes can be used at run time.

There are three primitive classes in SOM. They are the basis of all subsequent classes. They are: SOMObject which is the root ancestor for all SOM classes, SOMClass which is the root ancestor of all SOM metaclasses and SOMClassMgr which

is the class of the SOMClassMgrObject, an object created automatically during SOM initialization, to maintain a registry of existing classes and to assist in dynamic class loading and unloading. SOMClass is a subclass of SOMObject and SOMObject is an instance of SOMClass. SOMClass is an instance of itself. The various relationships between the classes, metaclasses and instances are shown in Figure 2.6.

The SOM system provides various class libraries including the ones for persistence and distribution. SOM supports mappings to C and C++.

2.3.4.6 Reduced Instruction Set Code Object Model

The Reduced Instruction Set Code (RISC) object model [110] from GTE Laboratories is a second example under the extensible object models category of broad domain, object-oriented protocol based systems that follow the selective approach. It is an adaptable object model. The various features of the RISC model are given below.

An object *state* is the object's private memory, which maintains information between the execution of the object's operations. An object's state is encapsulated. Object *methods* are the programs which implement the operations that the object can perform when a message is sent to it. An object *interface* defines the object boundary that separates the "inside" of the object from the "outside". The interface specification determines which aspects of the object are visible to clients of the object, and which aspects are invisible. The external interface of the object serves as a contract between the object and its clients. An object *identity* is the property of an object that distinguishes it from all other objects. Normally these identifiers are unique.

The *execution model*, in addition to modeling the structural aspects of different object models, provides a framework for realizing the behavioral aspects of different object models. This is achieved by defining an appropriate set of operations for each

of the types of primitive objects, together with an overall flow of control that ties these operations together. The combination of operations of the individual primitives, together with the steps in the overall flow of control, can then be used to describe the execution models of various object models. The primitive objects defined by the RISC object model constitute *metaobjects* and the operations defined for these objects, together with the overall flow of control, constitute a *metaobject protocol*. In order to be able to describe the execution models of various object models, messages and any aspects of objects that have to do with message handling are represented explicitly.

An *object's type* or *class* typically defines the internal form of the object, as well as defining the external interface associated with all objects of the class. In RISC, object types or classes are distinct objects of type "type". The basic behavior supported by type objects includes an operation that returns the signature of the type and a type-checking operation that determines whether a given object satisfies the type.

Object construction in the RISC model covers the construction of objects visible in a given object model and also the construction of the RISC primitives making up those objects.

The RISC model is planned to be implemented by the Distributed Object Management (DOM) 3.0 prototype.

2.3.4.7 Object Description Language

The Object Description Language (ODL) [117], developed at the University of Utah, is an example of a system that has a non-extensible object model under the narrow, object-oriented protocol based category. It consists of a common object model with features selected from the object-oriented languages C++, Objective-C, Smalltalk and CLOS. A noteworthy aspect of ODL is that languages from different program-

ming paradigms are considered for the common object model. This model could not be implemented successfully because of the vast differences among the features of the languages. The implementation is further complicated as the remote procedure call facility offered by the underlying operating system is not made use of for inter-language procedure calls.

2.3.5 Drawbacks of Existing Protocol Based Systems

Most of the existing systems following the protocol based approach do not support advanced polymorphism features. With the exception of the HERON system, these systems do not support remote inheritance of abstract methods. SOM does not include some of the object-oriented features such as concurrency and templates/generics. The RISC model is still under development and the identification of the "right" set of core model features is still under consideration. The Object models of existing non-extensible models are primitive. The COM model provides an *introspection* capability by allowing a client to query the available interfaces at run time. This model requires any language used for implementing a Windows object to be able to pass functions as parameters, because of the binary standard. Most of the models do not support advanced features such as method overloading, concurrency, and parameterized types.

2.4 Summary

Several existing interoperable systems have been introduced in this chapter. Some of their drawbacks are identified. A classification of existing system is provided. Some of the existing systems closely related to this dissertation are discussed in a greater detail than the other systems. This dissertation is classified under the narrow domain approach in the protocol based object models.

Chapter 3

Interoperable Common Object Model (ICOM)

3.1 Introduction

The proliferation of powerful computer workstations and faster networks facilitated the distribution of applications over multiple machines. As the technology improved with time, the complexity of distributed applications has increased. Typical distributed applications currently range from tens of thousands of lines of code to hundreds of thousands of lines of code. To develop such applications quickly and efficiently, reuse of existing software components must be done. Since software is developed in several programming languages, integrating software components developed in multiple programming languages has become a challenging problem. The heterogeneity of hardware architectures on which distributed applications are to be developed, added more complexity to the interoperability issues of existing software components.

Several common object models have been proposed to support application development in distributed and heterogeneous environments. As mentioned in Chapter 2, existing distributed models can be broadly classified into two categories: Common Runtime and Protocol Based. The Common Runtime models use shared memory and distributed shared memory whereas the Protocol Based models use separate address spaces for interacting distributed objects. In this dissertation the protocol based object models are considered.

The organization of this chapter is as follows: the proposed interoperable common object model (ICOM) is introduced in Section 3.2. The basic framework of the ICOM model that addresses the issues involved in incorporating advanced object-oriented features into a common object model, is explained in Section 3.2.1. An example application of 'Department Stores' is described in Section 3.2.2. This application is used throughout the dissertation to demonstrate the utility of the ICOM model. The functionality of the basic ICOM framework is illustrated through an example in Section 3.3. The various stages of application development process in the distributed object system supporting the ICOM object model are explained in detail. The programming techniques (including the actor model and reflection techniques) used in the implementation of the ICOM framework are described in detail. In this chapter, only the implementation of the basic framework that provides a way of binding methods dynamically when invoked on distributed objects is shown. An example of a digital library that uses the basic functionality of the ICOM object model is explained in Section 3.4. The basic framework is used to extend the ICOM object model to include the powerful features that are missing from the existing distributed object models. The extensions include the features of inheritance, parameterized types, and concurrency. These extensions are described in detail in chapters 4, 5 and 6. A summary of this chapter is provided in Section 3.5.

3.2 Interoperable Common Object Model

The Interoperable Common Object Model (ICOM) proposed in this dissertation focuses on object-oriented programming languages. The specific languages selected for implementing the framework are C++ and Modula-3.

3.2.1 Basic ICOM Object Model

To form a basis for developing a powerful object model, a simple distributed object model is chosen. This simple object model is termed as the basic Interoperable Common Object Model (ICOM). Only objects and method invocations of local and remote objects are supported in the object model. Other advanced object-oriented features can be added to the basic object model using the framework chosen for implementing the basic ICOM model.

3.2.2 Basic ICOM Framework

The ICOM framework follows the typical architecture of a distributed object system described in Section 2.3.2 of Chapter 2. The ICOM framework provides a uniform design for translation of the ICOM object model into statically typed object-oriented languages. The framework uses the *actor model* of concurrent computation and *reflection techniques*. The reflection techniques and the actor model are explained in detail in section 3.4. Reflection techniques are used to map the common object model supported by the ICOM framework into an object-oriented language. Here, the behavior of the model or the language is not changed. That is, a feature which is present in the common object model but not in an implementation language is interpreted in terms of existing features of the language. The intent is not to introduce a new

feature to the language. Any application in that language should be able to interoperate with the component written in a different language that has the same feature. This is termed as *one-way interoperability*.

The ICOM framework extends the Distributed Act++ model (discussed in Section 3.4) to support a common object model. Any distributed object generated as a result of IDL translation, is an *active* object in the ICOM framework. An active object is a first class entity. Every active object has a message queue. All messages intended for the object are enqueued in the message queue and processed one at a time. Any other object is a *passive* object in ICOM framework. A passive object cannot service method invocations in its own thread of execution. The ICOM framework allows the coexistence of both active and passive objects in an application. A client application treats all objects as passive in the basic ICOM framework. Some of these passive objects representing the remote objects are implemented as active objects. The framework tries to hide the differences between the active and passive objects. For an application developer there is minimal difference between implementing an active object and a passive object. The only requirement for implementing an active object is to describe the object's behavior in IDL, use an IDL translator to generate native language representations, and link the application code with the IDL translator generated code. Location transparency of the object implementations is provided in the ICOM framework.

3.3 Department Stores Problem

The application of 'Department Stores' is used to demonstrate the functionality of a typical distributed object system. The example presented in this section is used throughout this dissertation and is derived from the problem defined in [158]. An

illustration of communications between different objects of this example is shown in Figure 3.1. The problem description is given below.

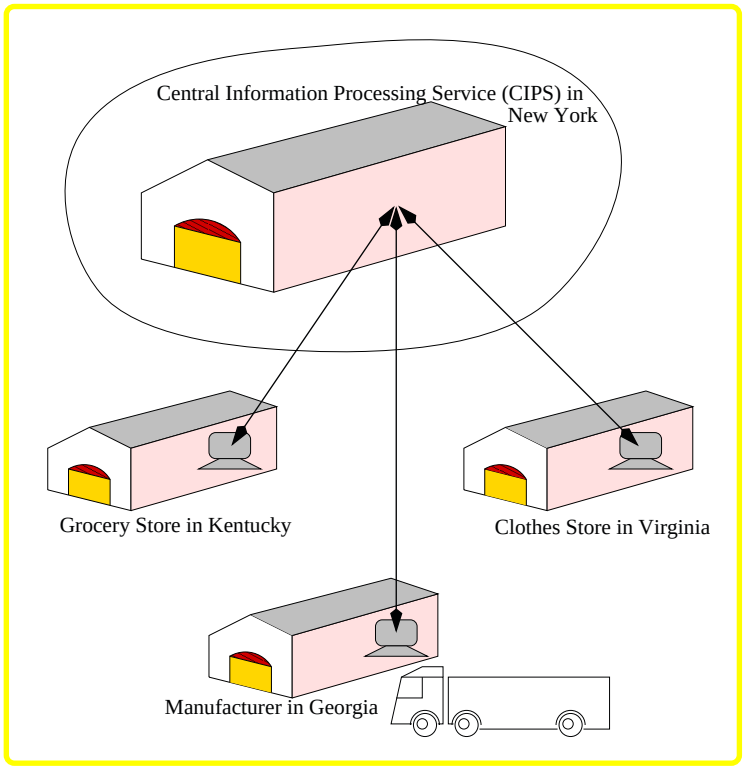

Figure 3.1: Department Stores Problem

“

The problem domain is a national level financial agency that owns a nationwide chain of stores. The types of stores owned by the agency are ‘grocery stores’ and ‘clothes stores’. The agency maintains a central information processing service (CIPS) that is used by the individual stores and manufacturers who supply products to the stores. The manufacturers and stores can be located in different geographical locations in different states of the United States.

> Individual stores supply information about their sales and losses. This information is processed by CIPS to update inventories, gross sales, and net profit for each store. The stores also provide local tax details that are used by CIPS in deciding the price of an item at a particular store. From the CIPS center, stores can get information such as the price of an item, whether a product can be put on sale or not.
>
> Apart from stores, product manufacturers can also interact with the CIPS center before delivering shipments to individual stores. Individual manufacturers can register with the CIPS center about an available product. When the inventory of a product at a particular store is low, the CIPS center can request a manufacturer to supply the product. The manufacturer can start shipping the product if it is readily available. Otherwise, the shipment gets delayed until the product is ready for shipping.
>
> ”

The various interfaces identified in this problem are StoreType, ClothesStoreBase, ClothesStoreDerived, GroceryStoreBase, GroceryStoreDerived, GeneralTax, VirginiaTax, KentuckyTax, and Manufacturer. Some of these interfaces are used in examples to illustrate the object-oriented features of a distributed object system. The StoreType interface describes the general functionality of a store that can be a grocery store or a clothes store. The interfaces ClothesStoreBase and GroceryStoreBase define the characteristics of a clothes store and a grocery store from the perspective of the CIPS center. The interfaces ClothesStoreDerived and GroceryStoreDerived define the localized views of the stores at remote places. The interface GeneralTax defines general characteristics of tax rules at the CIPS center. The interfaces VirginiaTax and KentuckyTax are the specialized interfaces that represent the characteristics of tax rules

at Virginia and Kentucky respectively. Finally, the Manufacturer interface defines the properties of a manufacturer who supplies products to a store.

Figure 3.2 shows the ClothesStore interface with one method. As mentioned in Chapter 2, the keywords of IDL are shown in **bold face**. There are two types of parameter passing modes shown, in and out. They represent 'input' and 'output' parameter passing modes respectively. Note that the figure does not show any return values for the methods. It is assumed that the return type is a 'void' type. The parameters of the methods can be either 'basic' types or 'user defined' types. The user defined types are passed by reference only.

```
interface ClothesStore {

    // returns quarterly sales at a clothes store through
    // the output parameter 'amount'
    quarterlySales(in int quarterNumber;
                   in int yearNumber;
                   out double amount);

    // returns monthly sales at a clothes store through
    // the output parameter 'amount'
    monthlySales(in int monthNumber;
                 in int yearNumber;
                 out double amount);

    ⋮
}
```

Figure 3.2: IDL Description of a Clothes Store Interface

The ClothesStore interface defines two operations called monthlySales and quarterlySales. The monthlySales method takes a monthNumber, yearNumber as input parameters and amount as an output parameter. It returns the monthly sales at a clothes store through the amount parameter. Similarly, the quarterlySales method returns the sales amount for a particular quarter at a clothes store. The implementation of this interface is assumed to be at a remote node. A client application using

the ClothesStore at the local node at New York can invoke the monthlySales method of the ClothesStore instance using a proxy instance of the ClothesStore. This type of invocation can happen at the end of every month or quarter.

As mentioned earlier in chapter 2, to prepare a server to accept requests from its clients, the various steps shown at the beginning of section 2.3.2 must be followed. The IDL translator, when invoked at the server location, parses the IDL description and generates interface files containing the interface specification and the implementation code for the ClothesStore interface in the native language. The server object interface is implemented by the server developer in the native language and linked to the IDL generated code. At the client side, the IDL translator generates a proxy for the ClothesStore object and interface files for use in the client application. The interface files contain the ClothesStore's interface rendered in the client's native language.

All requests to the proxy objects are marshalled[1] and sent to the server. At the server location the results are marshalled by the server and unmarshalled by the client and sent back to the client via the communication layer. Both the server code and the client code should be linked with the code generated by the IDL translators at their respective sites, and compiled with their respective native language compilers. When an invocation on a server object is made, the client application waits for the results and continues its execution after receiving the results.

[1]Marshalling is the process of building up a sequence of bytes that contains the values of the arguments or results in a proper format. Unmarshalling is the reverse process to marshalling; recovers values from the sequence of bytes.

3.4 Example 1: Localized Object

In this section, the 'Department Stores' problem is used to explain the interaction between remote objects in the ICOM framework. The IDL representation, translation, and method invocation are explained in detail.

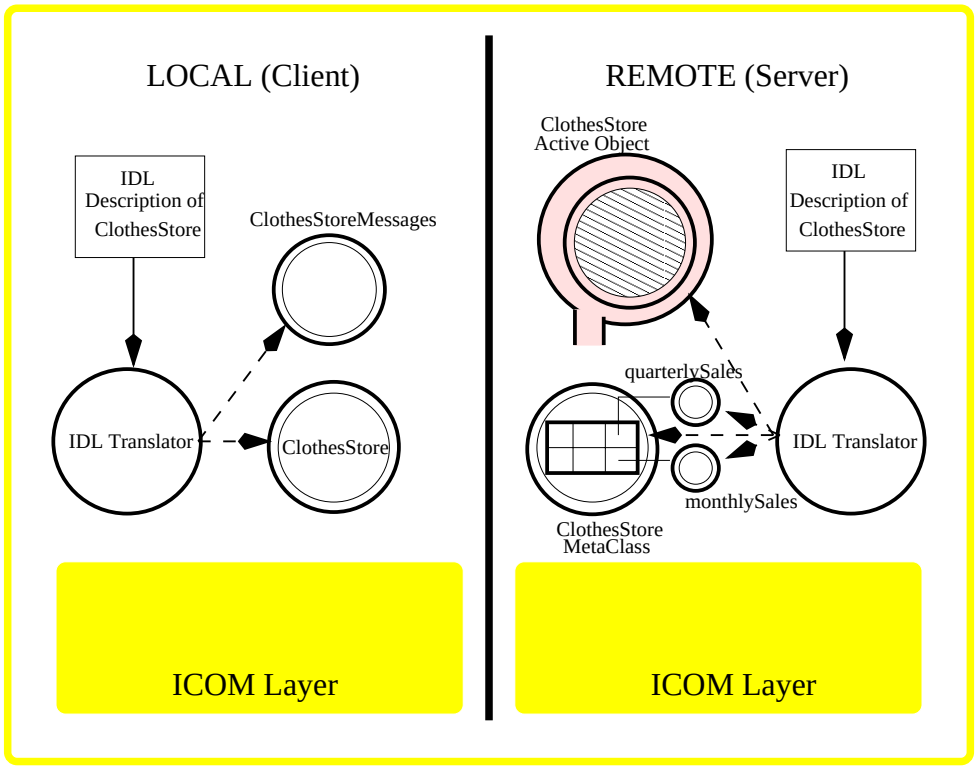

Figure 3.3: IDL Translation of Clothes Store Interface

Figure 3.3 shows the architecture of a system that supports the ICOM model, and the various classes generated as a result of invoking IDL compilers on the interface shown in Figure 3.2. In this figure and similar figures shown throughout the dissertation, the double circles represent the classes and the single circles represent objects instantiated from those classes. The dashed arrows represent the classes generated

by the IDL translator in Figure 3.3. The solid arrows represent the passage of IDL descriptions through the IDL translators. The shaded regions both at the local and remote nodes represent the ICOM communication layer whose purpose is to maintain communication links between every other node in the distributed system. The IDL descriptions are shown in rectangular boxes. In the ICOM framework, the translators are also objects. Any translator can communicate with any other translator at a different node or the same node via message passing. In the figure, the ring-like structure that wraps around a patterned circle represents an active object. It has an opening that represents the message queue of the active object. All patterned circles shown in this diagram and similar diagrams represent the classes whose implementation must be provided by an application developer. Implementations for all remaining classes are automatically generated by the IDL translators. There is a thick vertical line in the diagram that separates the local and remote nodes. This line indicates the distribution of different machines that are connected over a network.

As mentioned earlier, the IDL translator, when invoked at the remote node, parses an IDL description and generates language specific interface files and metaclasses. The interface files can be used by the client and the server to create and invoke methods on objects that support the interfaces. The translator generates a metaclass for the server. A metaclass controls the behavior of its instance, a class. Metaclasses contain method tables that have entries for the signatures[2] of the operations supported by its instance. In the current example, the translator at the remote node generates the ClothesStoreMetaClass, the server method classes, monthlySales and quarterlySales, and a skeleton class for the implementation of the class ClothesStoreImplementation. The method classes contain *future* variables for the output parameter, amount. A future

[2]A signature of a method is its argument types and the result type.

variable blocks the calling process when its value is not set. When the value is set, it unblocks the process and provides its value. The code for the ClothesStoreImplementation must be provided by the application developer. It is shown as ClothesStore Active Object in the figure.

At the client node, the IDL translator generates a proxy class, ClothesStore and a 'message generator' class, ClothesStoreMessages. The proxy class is a surrogate class for the server. The ClothesStoreMessages class can generate a message object for each method in the interface of the server. The message object generated contains a future variable for the output parameter, amount. A client application can continue with its execution after making an invocation on the proxy object, but blocks when it tries to access the value of the output argument of either of the methods, monthlySales and quarterlySales.

At the time of initialization of the server, the signatures of the operations of the server are entered into the method table along with a pointer to the operation implementation. This is shown as a table with two rows in the metaclass, in Figure 3.3. When an invocation is received by the server, the metaclass intercepts the operation invocation and sends appropriate message to the server object.

3.4.1 Dynamic Method Binding on a Localized Object

In the previous section, an introduction to the basic ICOM framework is given. A translation of an interface description into a native language for both a client and a server object is also explained in the previous section. A method invocation between a local object and a remote object is illustrated in the current section.

Assume that the client and server objects have been implemented at a local node and a remote node respectively. The server object is an active entity that has queuing

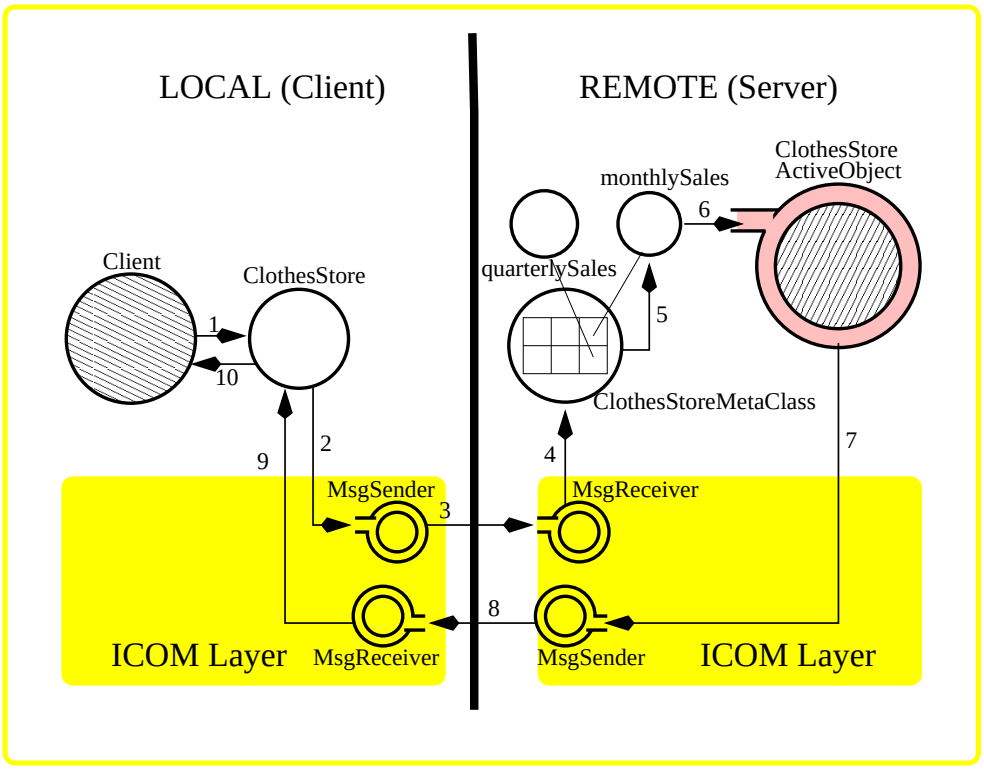

Figure 3.4: Dynamic Method Binding on an Instance of Clothes Store Interface

facility for messages. Queuing is an essential facility to have in applications that involve a large number of messages being sent to the server. The server object receives messages in its message queue and processes them one at a time.

Figure 3.4 shows an example of a remote method invocation of the method monthlySales on the ClothesStore ActiveObject. The sequence of method calls is described as follows:

At the local node the client application makes a method invocation on the local proxy object, an instance of ClothesStore (arrow numbered 1). The proxy object uses the ClothesStoreMessages instance (not shown in the figure) and creates a

message object for the message monthlySales. This message object is capable of encoding itself into the common network representation, ASN.1. The proxy object passes the message object to the message sender at the local node (arrow labeled 2). The message sender encodes the message by calling the encode method on the message object and sends it to the remote node (arrow labeled 3).

At the remote node the message receiver at the remote node partially decodes the message and uses the decoded information to find the metaclass instance for the server object. It then passes the partially decoded message to the server metaclass instance (arrow labeled 4). The metaclass instance, after doing a lookup in the method table, passes the message object to the server method class instance (arrow labeled 5). The method object, monthlySales, decodes the message and sends a message to the server's active object (arrow labeled 6). The server object executes the method corresponding to the method name, monthlySales. This method sets the value of the output argument, amount. Since, the output argument is a future variable, a message goes back to the local node, returning the value of the future variable, amount (arrows labeled 7,8, 9 and 10)

At the local node the client application if blocked on the future variable can continue with its execution after receiving a value for the future variable.

3.4.2 Implementation Details

To implement the framework that handles the method invocations as shown in the example described in the previous section, the ICOM framework uses reflection techniques. The framework uses the distributed actor model to provide the underlying

message passing between various nodes in the system. The framework is implemented with a minimal set of object-oriented features. The following sections describe the set of object-oriented features used for implementing the ICOM framework, reflection techniques, and the actor model in detail.

3.4.2.1 Core Set of Features

The ICOM framework is implemented using the object-oriented language features of 'object', 'class', 'single inheritance', 'method overloading', and 'concurrency'. In this dissertation this set of features is considered as the *core set* of object-oriented features. As mentioned in Chapter 1, all these features are either directly or indirectly supported in statically typed object-oriented languages. The languages, BETA and Modula-3, do not support method overloading feature. But, they have other features such as 'run-time type checking' which allow method overloading feature to be realized. In all statically typed object-oriented languages concurrency is either directly supported or is supported via standard libraries. The ICOM framework can be translated to any language that supports this core set of object-oriented features.

3.4.2.2 Reflection

In a typical object-oriented system, objects perform computation about an external domain. A reflective object-oriented system is said to be *causally* connected to its domain, if one of them changes, it leads to a corresponding change in the other [102].

Reflection is defined as the ability of a system to make structures representing (aspects of) itself the subject (or domain) of its computation in a causally connected way [102].

Reification is the representation of specific aspects of a program as data structures or classes within that program [49]. These aspects include method binding, inheri-

tance etc. The resulting objects are referred to as *metaobjects*. Invoking an operation on a metaobject is called *reflective computation*.

There are three approaches to reflective computation: *Metaclass, Metaobject*, and *MetaCommunication* [49]. If a reflective computation modifies a metaobject, then it is called a *reflective update* and the process is called *behavioral reflection*. If a reflective computation does not modify a metaobject, then that process is called *structural reflection*. The operations defined for the metaobjects constitute a *metaobject protocol* [83].

Metaclass Approach

In the *Metaclass* approach, the class of an object is considered as its metaobject. *Metaclasses* are metaobjects of classes. That is, a class is an instance of its metaclass. Metaclasses can have metaobjects for themselves, thus forming an infinite *reflective tower* [102]. This tower is normally terminated by defining a root metaclass which is a metaobject of itself [102].

A method invocation on an object can be described as follows: when an object receives a message, it is delegated to the object's metaobject, an object representing the class of the receiving object. The metaobject does a *lookup* operation for matching a method with the message and *apply* operation to execute that method. If the class itself has a metaobject, then the message is delegated to the metaobject which has its own lookup and apply operations. Normally, the root metaclass supplies default lookup and apply operations. If the behavior of a class needs to be changed, then the application programmer has to replace the default behavior of the root metaclass. This can be done by defining a specialized class from the root metaclass and assigning it as a metaclass of the class that required the behavior change. In this approach a

metaclass is used for both structural reflection and behavioral reflection.

Metaobject Approach

In the Metaobject approach, an object can have a metaobject for itself as opposed to a class having a metaclass in the Metaclass approach. This metaobject is an instance of a special class, typically called the 'MetaObject'. MetaObject defines the default behavior of all objects. These metaobjects are different from the class of the receiver (base-level object). They include only the operational and the control information about their base-level objects. The class of the receiver has the structural information. Classes handle structural reflection and metaobjects handle behavioral reflection. Upon receiving a message, a receiver delegates it to its metaobject, which performs *lookup* and *apply* operations. To change the default behavior of an object, the MetaObject can be specialized and assigned to the object. Since, each object has a metaobject, the number of objects in the system grows enormously. To control the number of metaobjects in the system, a *lazy* way of metaobject creation [49] can be followed where a metaobject is created only when needed. The advantage of this approach is that individual objects can be controlled in different ways even though they belong to the same class.

MetaCommunication Approach

This approach is based on the reification of the communication mechanism. Each message is treated as an object. Messages are represented as instances of a special class, typically called 'Message', that defines a method *send* for transmitting the message to the receiver. Every message object contains the address of a sender and a receiver. It is the responsibility of a message to interpret itself. That is, a message contains enough information about its method name,

class name, and the arguments for the method. This information can be used to deliver the message to the right object. The default interpretation information is provided in the 'Message' class.

In this dissertation, a combination of the MetaClass and MetaCommunication approaches is used to map the ICOM framework into the implementation languages. Since these techniques are used to map a model into a language, but not to change the semantics of a language, they are called *reflection-like* techniques.

3.4.2.3 Actor Model

The actor model is a concurrent computation model that unifies objects and concurrency. The model provides basic building blocks, called actors, for concurrent programming. An actor is a computation agent that has the ability to process messages. It is an active entity in the sense that every actor maintains the integrity of information within itself and processes its messages at its own discretion. Any computation in a system based on actor model is a result of message processing by the actors. Thus actors are autonomous and concurrently executing objects which execute asynchronously.

An actor consists of an address which represents the actor's identity and a script which represents the actor's behavior. A message queue is implicitly maintained in an actor. Any number of incoming messages can be received by the actor in its message queue. The behavior of an actor is responsible for processing the messages one at a time.

As a response to a message, an actor may send messages to its *acquaintances* (other actors that it knows about), create new actors, and specify a replacement behavior which processes the next message from the message queue of the actor, as

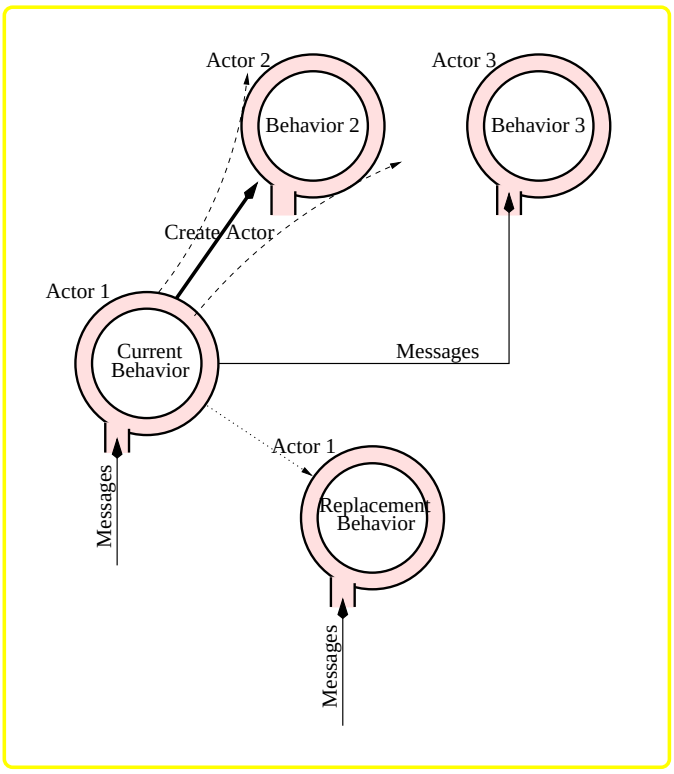

Figure 3.5: Actor Model

shown in Figure 3.5. An actor can have more than one thread of execution at a time. That is, a replacement behavior can be created while a message is being executed. The new behavior can start processing another message while the previous message is being executed. This kind of behavior replacement results in concurrent execution of messages and can potentially improve the performance of a system based on the actor model. It should be noted that concurrency also results with the simultaneous message processing by different actors.

Depending on the application, an actor can choose to accept only a certain group of messages, after processing a message. For example, if an actor represents a stack behavior, the initial behavior can accept only the push messages. After processing a

push message, the actor can accept both push and pop messages. Depending on what type of behavior an actor currently has, the messages in the message queue may get processed right away or stay in the queue until a suitable replacement behavior is provided by an existing behavior.

Concurrent object-oriented systems can be implemented using the actor model. Act++ [79] is a shared memory object-oriented programming environment based on actor model. It is implemented in C++. To facilitate synchronous communication, Act++ supports the concept of a 'Cbox'. A Cbox is a *future* variable. A future variable blocks the calling process until its value is set. Since messages are the only communication mechanism in the actor model, return values required in any method invocation are passed as Cbox arguments. For example, a method that is expected to return an integer can be implemented to take an 'integer Cbox' as its argument. The method implementation can set the value of its argument which is sent to the Cbox variable declared at the invoking process. The invoking process can continue its execution after the invocation. When it needs the value of the integer Cbox, it tries to access the value of the integer Cbox and is blocked if a value is not available. When the called method sets the value of its Cbox argument, a message comes back to the invoking process setting the value of the Cbox variable. At this point the Cbox value is available to the invoking process and it can continue with its computation.

3.4.2.4 Distributed Actor model

The Act++ system is extended to support distributed actors in Distributed Act++ [82] programming environment. In a Distributed Act++ environment every actor has a network-wide unique address. This address contains the node number which is a unique number assigned for a node in the system and a natural number that is unique for the actor at that node. A node in the Distributed Act++ system is an

abstract computer. An abstract computer consists of communication channels that can be used to communicate with every other abstract computer in the system. It also contains an application manager that manages the applications running in the system. A computer in a network can represent a node or multiple nodes can exist on one computer. A Distributed Act++ environment provides dedicated actors for communication between the nodes in the system. Each node in the system is connected to every other node via two dedicated communication actors called MsgReceiver and MsgSender. The MsgSender actor sends messages and the MsgReceiver actor receives messages meant for that node. If there are N nodes in the system, each node has (N-1) MsgSender actors and (N-1) MsgReceiver actors in the system. So the number of dedicated actors used for communication between nodes is (N-1)*(N-1) MsgSender actors and (N-1)*(N-1) MsgReceiver actors.

A distributed actor system with two nodes on two different machines is shown in Figure 3.6. Each machine can support one or more nodes in the distributed actor system. One of the nodes is treated as the main node that acts as a centralized manager. Before joining the system, every other node has to establish communications with the main node. The main node helps the new node to establish communication channels with every other node in the system. The communication channels are represented in the figure as MsgSender and MsgReceiver. These channels are managed by dedicated actors whose sole purpose is to send and receive information respectively, from the nodes they represent. There is a Boot actor object that creates the AppMgr actor at any node. The AppMgr actor keeps track of all applications currently executing in the system. The initialization of the system and the communication between various actors is described below.

The Distributed Act++ environment requires an initial setup that makes the communicating nodes (on possibly distributed machines) aware of the existence of

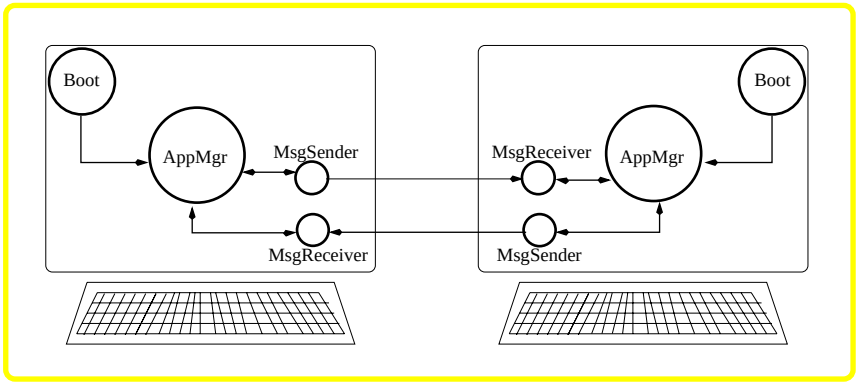

Figure 3.6: Distributed Act++ System with Two Nodes

one another. This initialization establishes the communication channels among all nodes. For every node in the system, a separate channel is established with every other node in the system. On any node of a distributed system, prior to the execution of an application, the application is registered with the application manager actors on all nodes of the system. Every application manager actor maintains a table of references to application actors in the system. After the first node is created, any number of nodes can join the system by communicating with the first node. An example of the initial setup involving three nodes on three different computers in a network is shown in Figure 3.7

To initiate an application at any node, a request message with an application name will be sent to any one of the application manager actors. The application manager actor loads the application actor, communicates with other application manager actors on the system about the new application and starts executing the application. The application actor can create on any node any number of actors required for its execution and communicate with them in achieving a common task.

To provide representation for data that can be transferred between machines with

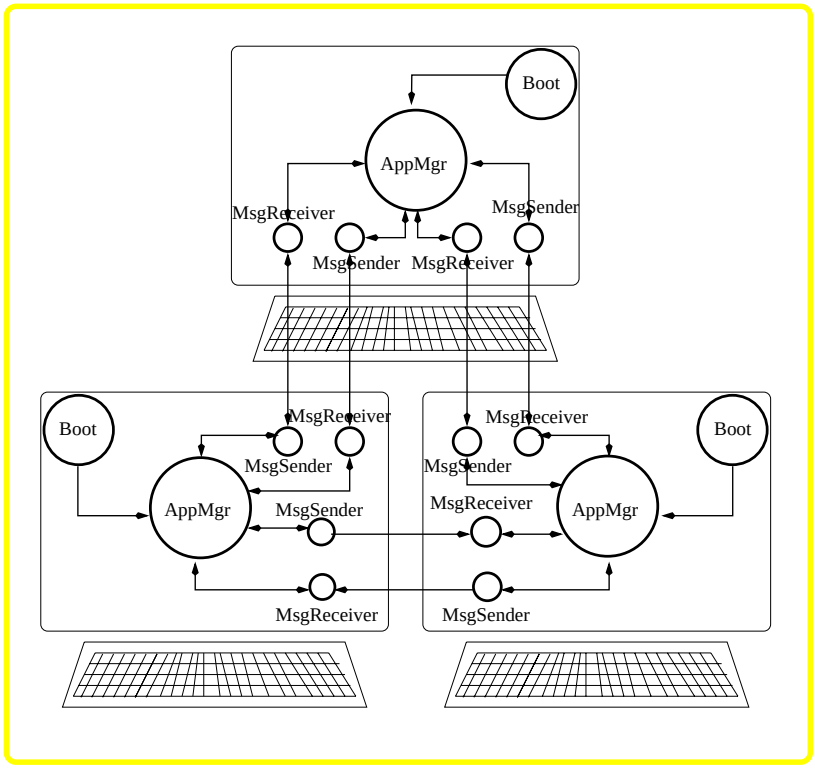

Figure 3.7: Distributed Act++ System with Three Nodes

different hardware architectures, Distributed Act++ supports a polymorphic object-oriented data specification framework (OODSF). This framework can be instantiated to support Abstract Syntax Notation 1 (ASN.1) [51] and the CORBA interface definition language (IDL) for data representation. The OODSF framework supports Basic Encoding Rules (BER) [52] and External Data Representation (XDR) [123] for transferring data between different nodes of the system.

3.4.2.5 Actor Messages used for Communication

There are two ways to implement the communication between actors. First, dedicated actors can be created that can decide whether message is a constructor message, an

invocation message or a reply message. In this approach there is only one basic message that is used for communication between actors. This approach results in many dedicated actors in the system, but the system will be uniform from the messaging point of view. The second approach extends the basic message set to include other basic messages such as the constructor message, invocation message, and a reply message. In this case, the system will have fewer actors but the messaging between actors is more complex. A sender and a receiver have to coordinate and agree upon the basic messages being sent and received. Although any approach can be followed to implement the communication, the second approach is followed in the ICOM framework implementation.

The relationship between an actor and an application object is shown in Figure 3.8. When the application invokes a method on an object that represents an actor (arrow labeled 1), the object first creates a message object using a message class corresponding to the method (arrow labeled 2), and then passes the message object to the acquaintance of the object (arrow labeled 3). For an application developer, this application object appears as a passive server object that responds to method invocations. But, in reality, this application object creates a message object and passes it to the actual server object implementation via the acquaintance.

There are three types of messages that are used for communication between actors in Distributed Act++ system: constructor, invocation, and reply messages. The communication between actors during the creation of application objects and method invocations on these objects is transparent to the application developer. The application developer views the system as a distributed actor system.

A constructor message is an instance of a class called Create. This class contains information (the arguments needed for the class constructor method in C++ for instance) required to create an instance of an application object. An application

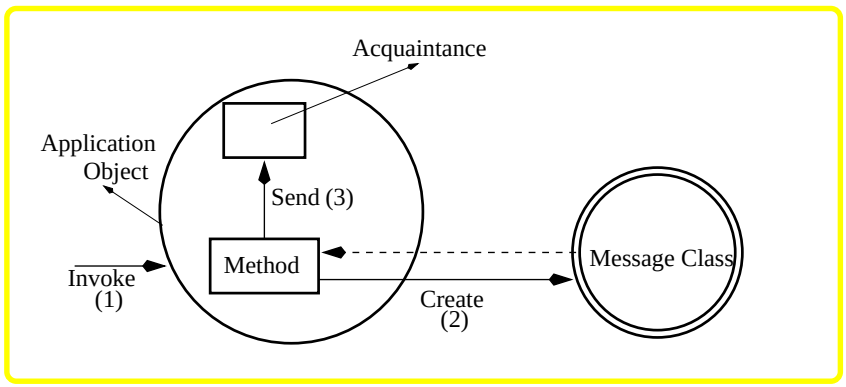

Figure 3.8: An Application Object that Represents an Actor

object can be a client object or a server object. A constructor message is passed as a method argument to create an acquaintance to a local or a remote object.

If the constructor is used to create a local application object, the acquaintance to the object builds an instance of an actor by passing the constructor as an argument. Since, the actor has access to the constructor message, it can invoke a new method on the constructor object. The constructor object then creates an instance of the application object which becomes the actor's behavior. This application object is controlled by the actor that created it.

If the constructor is used to create a remote application object, the acquaintance encodes the constructor object into a common representation that is understood by all nodes in a distributed Act++ system. The encoded message contains the class name and method name of the application object. It will then be sent to the node that has the application object implementation. The invoking process blocks and waits for the address of the remote actor. When the remote node receives the encoded message it decodes it to find the class name and method name of the server object. The remote node then creates a constructor object based on the class and method information.

At this point, the remote node creates an actor that is local to the remote node. The creation process is similar to the one explained in the previous paragraph.

A reply message is used to transfer addresses of actors between nodes. After a remote actor is created, its address will be sent back to the local node that originally sent a constructor message to the remote node. A reply message contains the address of an actor which is unique with regard to the distributed Act++ system. The remote node encodes and sends the reply message. The encoded message contains the class name and method name information and the address of the application actor. At the local node, the blocked process will be released after receiving the address of the remote actor.

An invocation message is an instance of a method class generated by the IDL translator. There is one method class for each operation defined in the interface of the application object. The invocation message object holds as its state information the arguments for the operation of the corresponding method of the application object. To invoke a method on an active application object, the application hands the invocation message to the acquaintance of the active object. This can be seen at the arrow labeled 1, in Figure 3.4. If the invocation is on a local actor, the message object will be handed to the actor that either delegates the message to its behavior for execution or enqueues it in its message queue for later processing. If the invocation is on a remote actor, the invocation message will be encoded and sent to the remote site by the acquaintance. The encoded message contains the method and class information along with the arguments required for that method. At the receiving side, the message will be decoded and reconstructed to form a message object again. This message object will be handed to the actor of the application object. If there are any output parameters in the method, their values will be sent back to the local node using reply messages. This invocation sequence can be seen in Figure 3.4.

3.5 Example 2: AGILE - Animated 3-D Graphical Interface

In this section, a second example is described which uses the basic ICOM frame work. This is a concrete example that demonstrates the functionality of the ICOM object model with a working library system.

The Animated Graphical Interactive Library Engine (AGILE) is a graphical user interface to the MARIAN library system being developed at Virginia Tech. The MARIAN library system is a search system for library catalog data from the Virginia Tech library collection.

A sample session with the AGILE interface is shown in Figure 3.9. A sphere attached with a torus is shown at the right corner. This sphere represents the MARIAN library system. The object with the title 'MailMan' represents a message object that takes queries from a user to the library object and returns the results back to the user. The object with the title 'MailBox' represents a place holder for the queries and results. The cylinder object at the bottom of the figure with the title 'System Response' displays the AGILE system responses to user actions. The cylinder object titled 'Enter Query' represents the field that takes user queries. The objects with the title 'Short Descriptions' represent the area where short descriptions of the results of a user query are displayed. There are two buttons titled, 'Previous' and 'Next' to browse through the results of a query. The rectangular block titled 'Detailed Description Record' displays a longer description of a result.

The AGILE interface is built on top of the Interoperable Common Object Model (ICOM) framework. It is implemented in C++, Modula-3, and C languages and is developed in a distributed, heterogeneous environment consisting of DEC Alpha-3000, Sun Sparc-10, and NextStep machines.

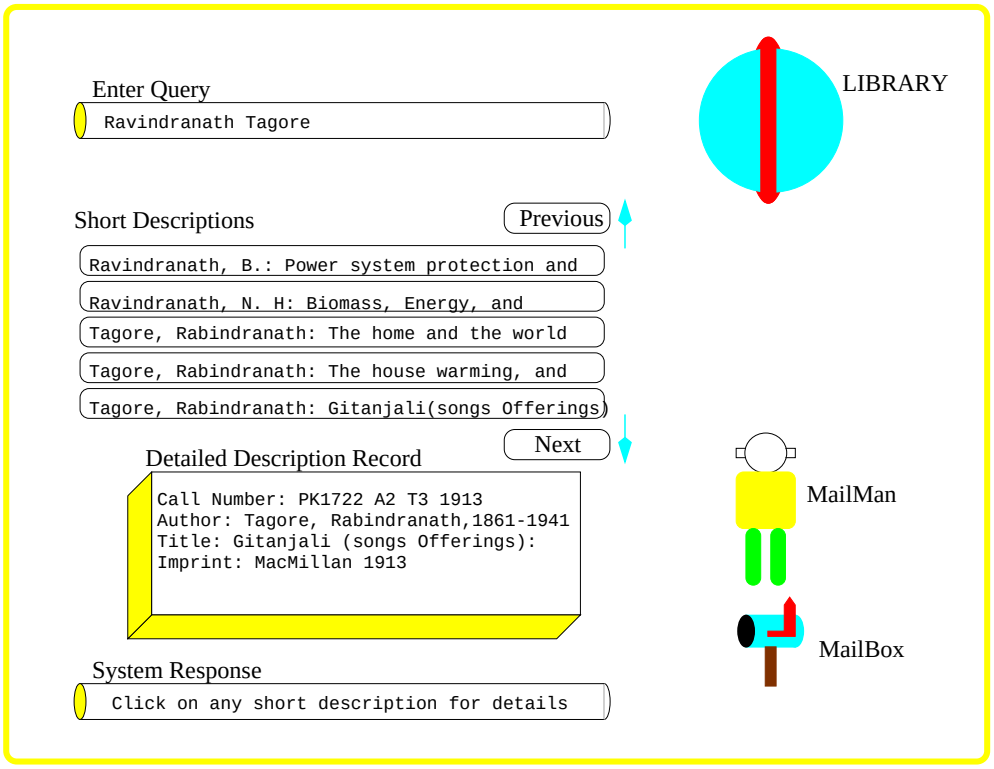

Figure 3.9: AGILE - Library Interface Sample Session

The AGILE interface consists of three major components as shown in Figure 3.10. The *User Interface* component, the *library* component, and the *communicator* component. The User interface component is implemented in Modula-3 using the 'Anim3D' animation library which is a 3-D graphics library with animation capabilities. The communicator component is written in C++ and it communicates with the library component via sockets using the 'gopher' protocol. The communicator component exists on the Sparc-10 machine whereas the user interface component exists on an Alpha-3000 machine. The library component is the server that responds to queries and it is installed on a group of NextStep machines. This component is written in C.

Using the AGILE interface, users can give queries with author names. The in-

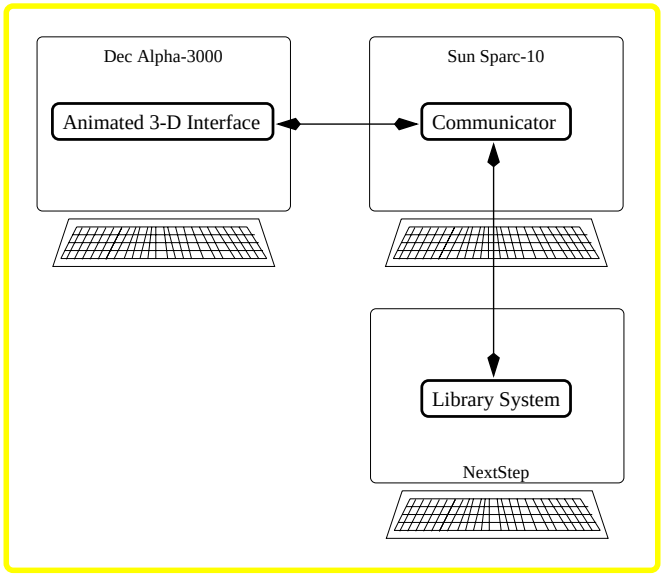

Figure 3.10: The AGILE System

terface component invokes the querySearch method on the communicator component. The communicator component in turn sends the query using the 'gopher' protocol to the library component. The results are eventually returned to the interface component and are displayed to the user. The issuing of the query and displaying of results is shown using animations of text as well as 3D objects representing the library and the query.

When the system starts, an animated title moves toward the center of the screen from the right corner. This is implemented using synchronous property values attached to the TextGO instances from the Anim3D package. The animation of the text is followed by a sphere (wrapped by a torus) that appears at the center of the screen. A group object consisting of spheres and cylinders, that looks like a robot rotates around the sphere. This animation is achieved by attaching synchronized property values to both the group object and the sphere object, using the Anim3D

package. The sphere is the object that represents the 'Library'. The group object represents the 'MailMan'. After a few seconds of animation, the 'Library' object moves to the top right corner of the screen and the 'MailMan' object moves to the bottom right corner of the screen. At the same time a mailbox is created at the bottom right corner of the screen.

A cylinder representing the query area, buttons for previous and next results, and rectangular boxes for displaying results appear on the screen immediately after the animation of the library and mailman animations towards the corners of the window.

At this point, the user can type in an author's name. After the enter button is hit, the query text moves from the top left corner to the bottom right corner towards the mailbox. After the query reaches the mailbox, the mailman takes the Query from the mailbox, and moves to the library object. At this point, the query is sent to the library using the communicator object. When the library component returns results, the mailman gets the results and moves back to the mailbox. The results are sent to the rectangular display boxes from the mailbox. The Previous and Next buttons can be used to move backward and forward on the results.

If the user wants to fetch a detailed record for a particular result, the result block can be clicked on with the mouse. The result displayed on the block now becomes the query. It reaches the mailbox and the mailman goes to the library object with the query. The communicator object sends the query to the library component. When the library component returns the results, the mailman comes back to the mailbox. The rectangular block moves from its position to the bottom of the screen and the results are displayed on it after moving from the mailbox.

At all times, there is a status box that displays brief messages intended for the user.

3.6 Summary

In this chapter, a typical distributed object system architecture is explained. The ICOM framework is described in detail. Brief descriptions of actor model, distributed actor model implemented in Act++ and Distributed Act++ are given. An example of a dynamic method binding in the ICOM framework is explained. The basic messages for communication between various actors are described. Finally, a 3-D graphical interface that uses the ICOM framework is described.

Chapter 4

Basic Polymorphism

4.1 Introduction

Polymorphism is an important abstraction mechanism used in object-oriented programming languages. It represents the quality or state of being able to assume different forms. A class is polymorphic if a method invocation of the class results in the execution of a method defined in a subclass of the class. A type is polymorphic if a variable of the type can change its type during the life-time of a program. Similarly, a polymorphic method can accept different types of arguments and a polymorphic operator can operate on multiple types of operands.

There are two general categories of polymorphism — *ad hoc polymorphism* and *universal polymorphism* [25]. Ad hoc polymorphism is achieved by *type coercions*, *operator overloading*, and *method overloading*. For example, consider the assignment of an integer value to a variable declared to hold a real (float or double) value. The integer value will be converted to a real number before the assignment takes place. This is type coercion. The addition operator '+' is an overloaded operator in many

programming languages because it works both for integer addition and real number addition. So, the operator '+' is a polymorphic operator. Method overloading in a class is achieved by defining multiple methods with the same name but with differences in argument count or argument types. For example, a class representing the ClothesStore interface from the 'Department Stores' problem can support an overloaded method with the name, priceOfItem. The class can have two methods with the same name, priceOfItem, but differ in their first argument type. The price of an item can be determined either with the item number assigned to it or with a label used to identify the item. So, the first overloaded method can take the item number (of type long) as its first parameter whereas the second overloaded method can take the item label (of type string) as its first parameter.

Universal polymorphism is achieved by *inheritance* (abstract methods defined in a base class can be inherited and overridden in a derived class, some times referred to as *inclusion polymorphism*), and *parameterized types* (the capability of handling more than one type in the parameter list of a generic type). Method overloading and universal polymorphism are discussed in detail in the next chapter.

A typical interpretation of polymorphism with reference to object-oriented programming is the binding of different methods at runtime when a method is invoked. In languages like C++, the concept of 'virtual methods' represents polymorphism. For example, an abstract method can be declared as 'pure virtual' in a base class. This method can be overridden in a derived class. A handle to the base class (in C++ terminology, it is a pointer or a reference) can hold a reference to an instance of the base class. When the virtual method is invoked using a handle to the base class that holds a reference to a derived class' instance, at the time of method binding, the method binding mechanism binds the invocation to the overriding method in the derived class. This is very difficult to manage in distributed systems because the base

class site may not be aware of the existence of the derived class at the time of its creation.

In many object-oriented programming languages, inheriting abstract methods is treated as the basis of polymorphism. One of the main drawbacks of the majority of existing distributed object systems identified in Chapter 1 is the lack of support for remote inheritance of abstract methods. In the ICOM framework, the inheritance of abstract methods is supported. This support is not limited to the objects whose state and method implementation exists in one process only. The components of an object can exist at distributed sites. The distributed components must be related to one another via superclass and subclass relationships. A subclass can override an abstract method defined in its superclass.

In the ICOM framework, inheritance of classes across language and machine boundaries is allowed. Instances of classes that have their base classes in different address spaces are defined as *truly distributed objects*. A common object model that incorporates truly distributed objects is said to support basic polymorphism.

Truly distributed objects, with the support for basic polymorphism, distinguish themselves from regular objects by keeping parts of their state and method information at distributed locations. These distributed parts of the object are related by a superclass and subclass relationship. A truly distributed object allows abstract methods in the superclass to be overridden in a subclass. A *split object* differs from a truly distributed object in the way the various parts of the object are distributed. The distributed parts of a split object need not be related by any superclass and subclass relationships. The distribution decisions can be arbitrary. Thus, every truly distributed object is a split object, but not every split object is a truly distributed object.

4.1.1 Interface Inheritance Vs Implementation Inheritance

Inheritance has multiple interpretations in object-oriented languages. Two such interpretations are: *subtype inheritance* and *implementation inheritance*. Subtype inheritance is also known as *interface inheritance* in the context of common object models. It means that an instance of a subtype can substitute for an instance of its base type in any context. An implementor of a subtype must implement the functionality of the base type of the subtype along with any extended functionality of the subtype. Implementation inheritance in some languages is known as *class inheritance*. An implementor of a derived class can inherit the implementation of its base class and implement only the extended or overridden functionality in the derived class. Since the behavior of a base class can be modified in a subclass, an instance of a derived class cannot be substituted by an instance of a base class in all contexts [38]. This is opposite of the substitution given for interface inheritance.

Existing distributed systems [20, 60, 70] support *interface inheritance* where only the specification is inherited from a parent interface. This means that an implementation supporting an interface child that inherits from another interface parent must provide implementation for the methods in both the interfaces, child and parent. The drawback of such an approach is that if an implementation exists for the interface parent on a remote machine, the implementation of the interface child cannot reuse it on the local machine. Thus, existing systems do not support objects that are distributed across processes possibly on different machines. In this chapter one approach is discussed that supports distribution of objects in different processes on different machines without losing the integrity of the objects.

In a typical object-oriented programming language, if a subclass inherits from a superclass, the state and methods of the superclass become a part of the instance of

the subclass. If a method defined in the superclass is invoked on an instance of the subclass, it is delegated to the superclass' implementation. A subclass implementor need not re-implement the superclass' code. Some distributed object systems support this type of inheritance using *proxies*. A proxy is a surrogate object that represents a remote object at the local site. A client using a proxy can treat it as a real object. The CORBA compliant system, ORBIX, from IONA Technologies, supports inheritance using proxies [175]. But, this works only for split objects. Because, in the ORBIX system, inheritance of abstract methods is not possible across different address spaces. It is fair to say that the ORBIX system supports partial implementation inheritance.

Objects in a distributed system can either be local or remote. A local object can exist in the same process or in a different process. Typically, for a client of a server object, the location of the server is transparent. The underlying system takes care of delivering the method invocations to the server objects regardless of their location. If the object is remote then the system packages the message, delivers it to the right object, and returns the results to the client. Since, existing models (except HERON [188] and to some extent ORBIX [175]) support interface inheritance only, the true power of object-oriented programming cannot be achieved in these systems. One can only invoke methods on remote objects, but there will not be any communication between the remote object and the objects in the local environment as far as the execution of the method is concerned. The main reason for this drawback is that the parent is not aware of its subclass, and there is no provision in existing systems to inform the parent of its subclasses existence. This is addressed in the next section.

4.2 Truly Distributed Objects in ICOM

The framework of Interoperable Common Object Model [28] realizes a distributed object in its true sense by supporting inheritance of abstract methods across processes. The parts of a distributed object can exist in more than one process without losing the object's integrity. The base class of the object can exist in one process on a remote machine and the derived class of the same object can exist in another process on a local machine. When overridden abstract methods are invoked in the parent, the redefined methods of the child are executed.

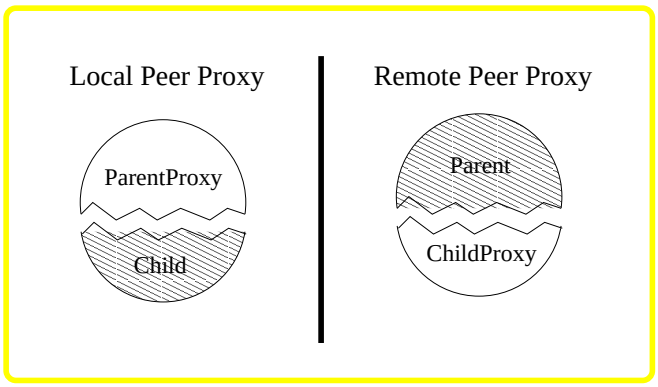

Figure 4.1: A Simplified Representation of a Truly Distributed Object

A simplified representation of the truly distributed object is shown in Figure 4.1. The cracked egg shell representation is used to show the distribution of state and methods of the truly distributed object. The patterned portions represent the code that must be implemented by a developer. The method implementation that is part of the white interior in the objects, represents stub methods that delegate any invocations on them to their corresponding implementation at the other node. The combination of the patterned portion and the white portion of the cracked egg shell representation denotes a *peer proxy* at that site. So, in the ICOM framework, peer

proxies represent truly distributed objects at the sites that have at least one component of the truly distributed object.

4.2.1 Distributed Compilation

In the existing distributed object systems, an IDL translator must be invoked independently (stand alone compilation) at a client site as well as at a server site for a client and a server to work together in a distributed environment. No information exchange takes place between a client node and a server node during the translation. The main barrier to truly distributed objects in existing systems can be attributed to the fact that at a remote site that implements a superclass interface, there is no information about the existence of a subclass that can override an abstract method defined at the superclass. So, an invocation of an abstract method defined at the superclass, on an instance of the subclass, results in the incorrect invocation of the method implemented at the superclass.

In ICOM, a new concept of *distributed compilation* is introduced to support communication between distributed nodes at IDL compilation time. Distributed compilation involves executions of translators at multiple sites when a truly distributed object description is to be translated from IDL to a native language. In the ICOM framework, each IDL translator is treated as an active object that has its own thread of execution. As seen in Chapter 3, the initialization phase of the distributed object system involves initial interaction between different distributed nodes. After the communication between different nodes is established, IDL translators can be registered with each node. So, each node can access the IDL translators on all nodes. Any IDL translator can potentially send messages to any other IDL translator.

In the current prototype, the IDL translations are hand generated. The rest of

the process involving actor creation, registration, and communication between the objects is done by the system at runtime.

The 'Department Stores' example discussed in the previous chapter is used to illustrate the distributed compilation of a truly distributed object. Figure 4.2 shows the interface descriptions of two interfaces, GeneralTax and VirginiaTax. In this figure, the interface GeneralTax is a base interface and the interface VirginiaTax is a derived interface. These two classes are used to find out the total tax for the state of Virginia in the United States of America. This tax is used to decide the price of an item sold at a retail clothes store in Virginia. The GeneralTax interface defines three methods, totalTax, stateTax, and salesTax. The VirginiaTax interface inherits from the GeneralTax interface and overrides the two methods, stateTax and salesTax methods.

The distributed compilation mechanism used in ICOM is illustrated in Figure 4.3. In the figure, the cracked egg shell representation shows a distributed object with its state and methods split on two different nodes. The dashed arrows represent the portion of code generated by the translator. The double circles represent classes and the single circles represent objects. The patterned objects represent the code that must be implemented by a developer. All remaining objects with white interior have their code generated automatically by the IDL translator.

The working of an IDL translator can be seen with the translation of a truly distributed object. Consider a truly distributed object with a superclass part GeneralTax existing on a remote node called remote, and a subclass part VirginiaTax existing on a local node called local. The nodes are separated physically via a local area network or a wide area network. Assume that an application developer at the local node intends to use an instance of the distributed object. At the local node the application developer starts the IDL translation on the IDL description of the truly distributed object that contains the interfaces of GeneralTax and VirginiaTax.

```
// GeneralTax defines two methods
interface GeneralTax {
    // returns the sales tax through 'tax'
    salesTax(out double tax);
    // returns the state tax through 'tax'
    stateTax(out double tax);
    // returns the total tax for a state
    totalTax(out double tax);
}

// VirginiaTax type inherits from GeneralTax and
// overrides the two methods
interface VirginiaTax : reuse GeneralTax {
    // returns the sales tax through 'tax'
    salesTax(out double tax);
    // returns the state tax through 'tax'
    stateTax(out double tax);
  }
```

Figure 4.2: IDL Description of Virginia Tax Distributed Object Interface

Assume that the implementation of the interface GeneralTax already exists at the remote node. This process is similar to the translation explained in Chapter 3. For each interface in the inheritance hierarchy four classes are generated. This generation preserves the inheritance relationships at each distributed node which has the implementation of an interface in the hierarchy. The IDL translator generates four classes for the GeneralTax interface: GeneralTaxMetaClass, GeneralTaxImplementation, GeneralTaxMessageClass, and GeneralTax. The GeneralTaxMetaClass contains a method table. Each row in the method table contains a method name, a class name, and a server method object corresponding to the method name. The table contains entries for each method defined in the interface. Each server method object is an instance of a local class (called server method class) defined in the metaclass. The purpose of a server method class is to decode an incoming message. If a message arrives over a network, the method class instance whose method name matches the method name in the incoming message, decodes the arguments in the message. In the case of a local

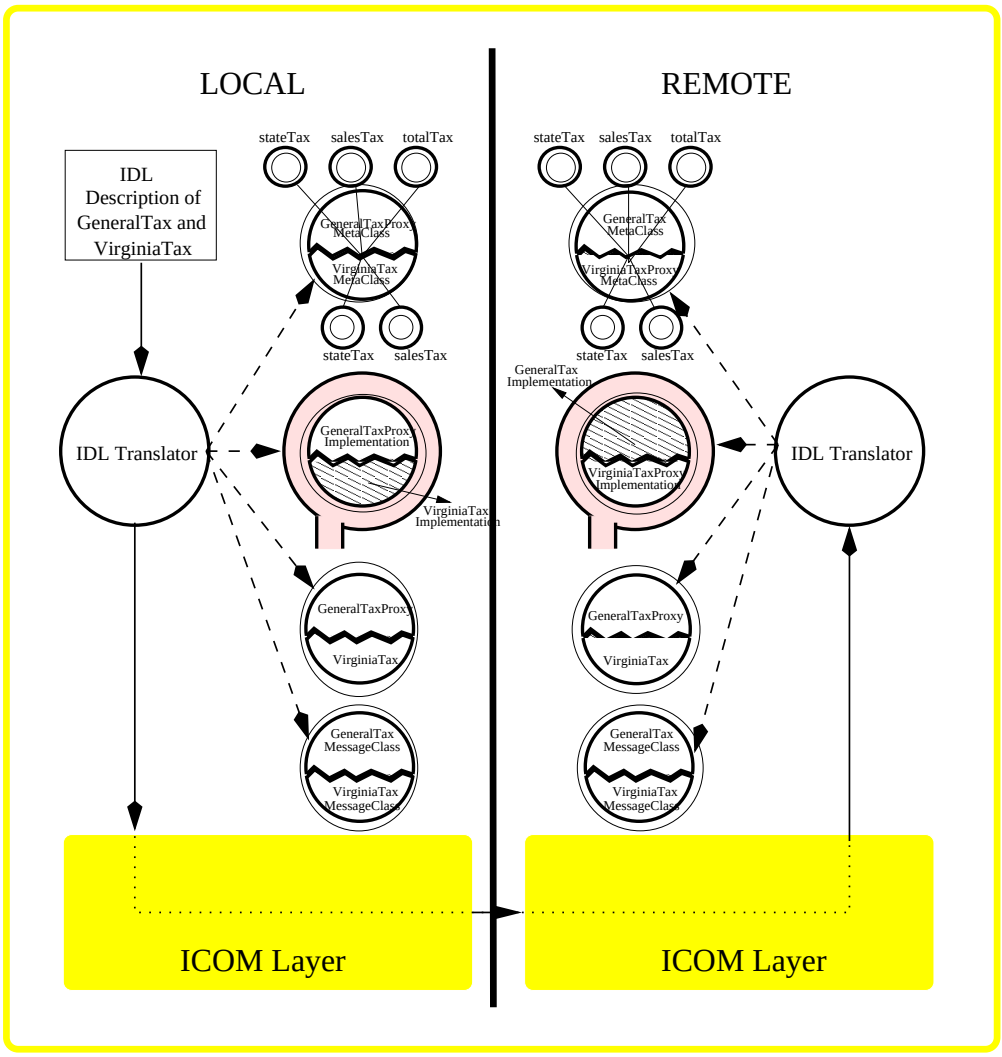

Figure 4.3: IDL Translation of Virginia Tax Distributed Object Interface

invocation that does not involve any message encoding, the decoding process simply consists of copying of the arguments from the message object. The GeneralTaxImplementation class is the actual implementation of the GeneralTax interface. It must be implemented by a developer at the remote node. All of the remaining classes are automatically generated by the IDL translator. The GeneralTaxMessage class contains classes defined for each method defined in the GeneralTax interface. These method classes are called *client method classes* because their purpose is to create message objects for each method invocation on an active object. The message objects are capable of encoding themselves into a network representation format. The GeneralTax class supports the GeneralTax interface. It does not contain any state information except the address of a corresponding active object of GeneralTaxImplementation. Any application that wants to invoke methods on a GeneralTax object must use a GeneralTax class instance as a handle.

The implementation of the GeneralTax interface is made available to others by linking the IDL generated code with the GeneralTaxImplementation code by an implementor, and using a native language compiler to generate the executable. Any application must load the GeneralTaxMetaClass either statically or dynamically before trying to create and use a GeneralTax object. In the current implementation of the ICOM framework, dynamic loading is used in C++ and static loading is used in Modula-3.

In the current example, the GeneralTax is implemented at the remote node and the VirginiaTax is to be implemented at the local node. The IDL translator, when invoked on the IDL description of the truly distributed object that contains both GeneralTax and VirginiaTax interfaces, realizes that GeneralTax is implemented remotely. As in CORBA, the IDL translator can communicate with an *interface repository* which has details about all interfaces available in the distributed object system, to find out the

location of the GeneralTax implementation. The translator sends a compile message to the IDL translator at the remote node by passing the interface information of the truly distributed object. It continues with its translation in parallel. The IDL translator at the remote node generates four classes representing the VirginiaTax at the remote node. The classes generated are: VirginiaTaxProxyMetaClass, VirginiaTaxProxyImplementation, VirginiaTax, and VirginiaTaxMessageClass. The VirginiaTaxProxyMetaClass inherits from the GeneralTaxMetaClass. Its functionality is the same as that of the GeneralTaxMetaClass except that it is oriented towards the VirginiaTax class. The VirginiaTaxProxyImplementation implements stub methods whose purpose is to create message objects for each method invocation of the methods in the VirginiaTax's interface and send them to the local node. The VirginiaTaxMessageClass is used to create message objects meant for VirginiaTax. The VirginiaTax class is a handle to the implementation of VirginiaTax. It supports the VirginiaTax's interface and sends all messages to it to the VirginiaTax active object at the local node. After generating the required classes, the IDL translator at the remote node invokes a native language compiler to generate executable code.

The IDL translator at the local node generates eight classes, four each for the GeneralTax and VirginiaTax interfaces. The parent classes generated are: GeneralTaxProxyMetaClass, GeneralTaxProxyImplementation, GeneralTaxMessageClass, and GeneralTax. Their functionality is similar to the VirginiaTax classes at the remote node, except that they are oriented towards the GeneralTax implementation at the remote node. They provide skeletal structures and delegate any method invocations to the remote node.

The classes generated at the local node are: VirginiaTaxMetaClass, VirginiaTaxImplementation, VirginiaTaxMessageClass, and VirginiaTax. The VirginiaTaxMetaClass inherits from GeneralTaxProxyMetaClass, the VirginiaTaxImplementation class inherits

from GeneralTaxProxyImplementation class, the VirginiaTaxMessageClass inherits from GeneralTaxMessageClass, and finally the VirginiaTax class inherits from GeneralTax-Class. The derived classes' functionality is similar to the base classes' functionality at the remote node.

A developer needs to implement the VirginiaTaxImplementation class and invoke the native language compiler on the source code of the VirginiaTaxImplementation class and the IDL translator generated code to build an executable. The developer can include the VirginiaTax class in the application and use it to create and invoke messages on an instance of VirginiaTax. A client application can treat the component of the distributed object at the client site as a complete object and invoke operations on it. The underlying system executes the correct implementations of the invoked methods.

4.2.2 Example: Method Invocation on a Truly Distributed Object

In the previous section, a detailed description of how the distributed translation works on a truly distributed object is given. In this section, a method invocation on an object supporting the interface VirginiaTax is described in detail.

Consider a node (machine in a network) that has class whose implementation (hereafter referred to as parent) supporting the GeneralTax interface. Assume that another node (hereafter referred to as the child node) has a class whose implementation supports the interface VirginiaTax, which inherits from the interface GeneralTax.

Assume that a client application has an instance variable handle that is a handle to the class GeneralTax, but contains an instance of the class VirginiaTax. A method invocation on the method stateTax on the handle results in a sequence of method calls.

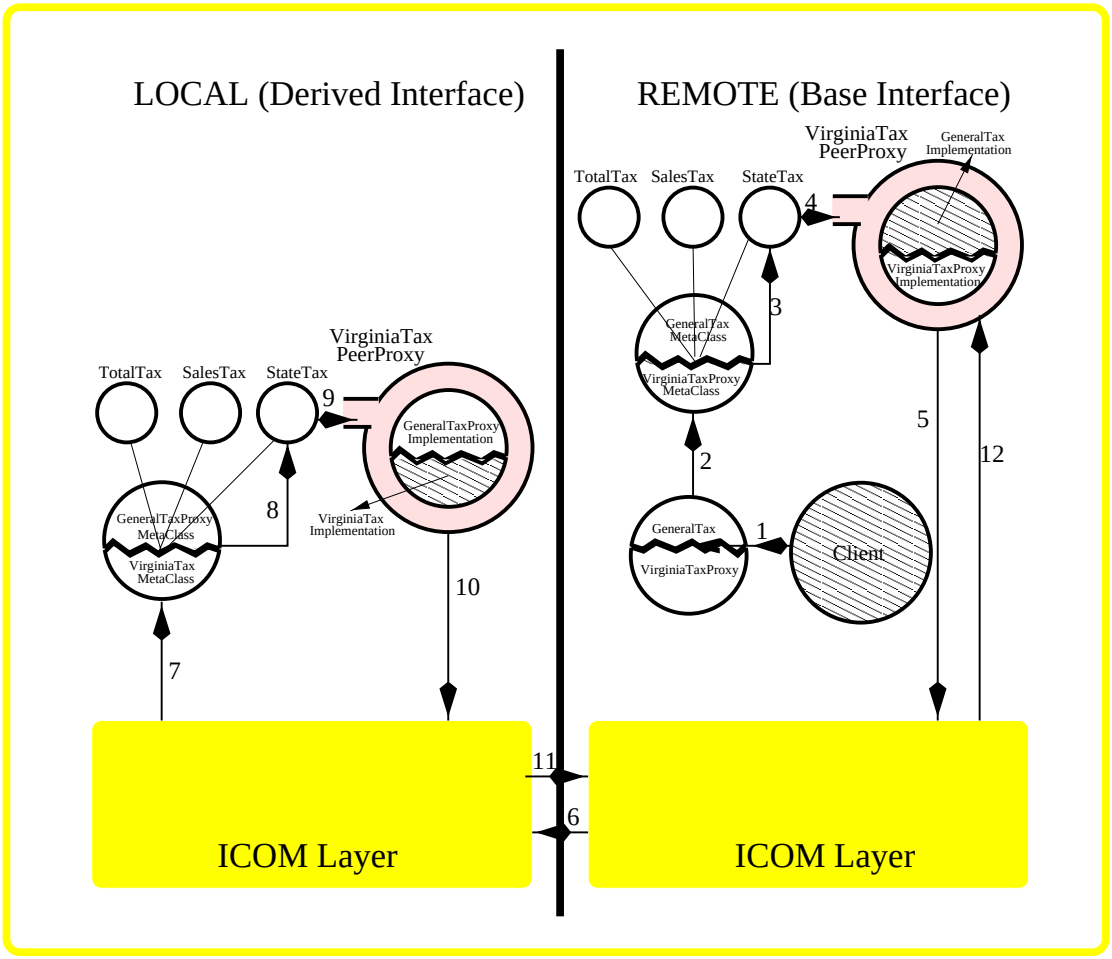

Figure 4.4: Dynamic Method Invocation on Virginia Tax Distributed Object

These calls are numbered in the order of their invocation as shown in Figure 4.4. The method invocation of stateTax is received by the VirginiaTaxProxy and is forwarded to the VirginiaTaxProxyImplementation. The VirginiaTaxProxyImplementation class forwards the method to the VirginiaTax's implementation. Eventually, results will be sent to the client application via the output parameter tax of the method stateTax. The call sequence is explained in more detail as follows.

At the parent node

A client application can make an invocation on the handle that refers to an instance of VirginiaTax class. This call is shown with the arrow labeled 1. The application uses a handle to the active object instance. This handle refers to an instance of VirginiaTax class which inherits from GeneralTax class. The handle object creates a message object for the method stateTax. The message object is forwarded (arrow labeled 2) to the instance of VirginiaTaxProxyMetaClass. This class inherits from GeneralTaxMetaClass. The message object then is forwarded to the server method object named stateTax (arrow labeled 3). Since the invocation is a local invocation at parent node, the decoding process in the server method object simply copies the arguments from the message into the method object stateTax. The server method object stateTax makes an invocation on the *peerProxy* object (arrow labeled 4). Even though a method implementation exists in the GeneralTaxImplementation class, since the method is an abstract method that is overridden in the derived class, the stateTax method of the class VirginiaTaxProxyImplementation is invoked. This proxy implementation is a stub implementation that creates a method object and forwards it to the child node (arrow labeled 5). The ICOM layer encodes the message object by calling the encode on the message object. The encoded message is then sent to the child

node (arrow labeled 6).

At the child node

The child node receives the encoded message from the network. It partially decodes the message to obtain the method name and the class name of the stateTax object. It identifies the metaclass instance corresponding the name 'VirginiaTax' and forwards the partially decoded message to it (arrow labeled 7). The metaclass instance hands the message to the server method object stateTax. The message is decoded in the object stateTax (arrow labeled 8). It then sends the decoded message to the *peerProxy* (arrow labeled 9). The stateTax method of the *peerProxy* is executed. Since the message contains a *future* variable, when the variable is a value assigned to it, a message goes to the parent node (arrow labeled 10). This message is encoded and sent to the parent node through ICOM layer (arrow labeled 11).

At the parent node

The ICOM layer at the parent node receives the message, decodes it and updates the value at the client application (arrow labeled 12).

4.2.3 Implementation Details

The framework of ICOM described in Chapter 3 is extended to support truly distributed objects by introducing two new message primitives to the existing Distributed Act++ environment. The Distributed Act++ environment defines three basic message primitives. These basic messages are: constructor, invocation, and reply messages. The new messages added are: peerInvoke and peerUnblock.

The new messages, peerInvoke and peerUnblock, are added to assure the atomicity of a method execution. That is, a message processing started must be completed

before another message processing starts as a result of an external invocation. Various peer components of a truly distributed object must coordinate with one another to guarantee the atomicity of method execution.

In the ICOM framework, any peer proxy that starts message processing is treated as the 'owner' that has permission to process a message. The ICOM framework blocks any external invocations on all peer proxies, after an owner peer proxy starts executing a message. Any new messages will be entered into the message queue of the proxy. If a local peer proxy needs to make an internal invocation on a method of a peer proxy at a remote site, a peerInvoke message will be used. A peerInvoke message is similar to an invocation message, except that its name indicates that it is from a peer proxy of the distributed object. The actor corresponding to the remote peer proxy lets any peer invocations pass through without blocking them. This avoids deadlocks in the system that may occur because of peer invocations.

When any peer component completes an execution of a method without making any peer invocations in the method implementation, it sends peerUnblock message to the owner peer proxy that originally started the message processing. This original peer proxy is the owner of that execution. The purpose of a peerUnblock message is to unlock the peer proxies that were locked after a message processing starts. The owner peer proxy after receiving a peerUnblock either from a remote peer proxy or from itself, sends peerUnblock message to all peer proxies and waits for acknowledgments from them. When a non-owner peer proxy receives a peerUnblock message, it responds by sending another peerUnblock message back to the sender. The owner starts to execute new external messages after it receives acknowledgments from all peer proxies. The acknowledgment message is again a peerUnblock message. The owner can distinguish between the first peerUnblock message and the remaining acknowledging peerUnblock messages.

4.3 Summary

In this chapter, a truly distributed object integration into the prototype system is explained in detail. A new concept of distributed compilation is introduced for a transparent integration of a truly distributed object. An example of a method invocation on a truly distributed object is described. The implementation framework extensions are explained.

Chapter 5

Advanced Polymorphism

5.1 Introduction

Polymorphism provides higher level abstraction mechanisms in object-oriented programming languages. It enhances domain understanding by representing in an abstract entity the common characteristics of domain-specific entities. As mentioned in Chapter 4, two general categories of polymorphism have been identified — ad hoc polymorphism and universal polymorphism. Ad hoc polymorphism represents 'type coercions', 'operator overloading', and 'method overloading'. Type coercions and operator overloading are not discussed any further because they involve language specific details which could not be addressed at a common object model level[1]. The advanced features of universal polymorphism are 'inclusion polymorphism' that involves inheritance of abstract methods, and 'parameterized types'.

Several advanced polymorphic features are present in statically typed object-oriented languages. But they are not present in the existing common object models.

[1]This is because one of the design considerations of this research is that language semantics and compilers should not be changed.

This results in poor reuse of software components written using these advanced features. Also, software components cannot be written using these features, because they are not present in the common object models. Specific features missing in the existing common object models are method overloading and parameterized types. Since one of the main objectives of this dissertation is to bring common object models closer to language object models, it is necessary to elevate the common object models to include method overloading and parameterized types.

An important design criterion to be observed while introducing the advanced features to a common object model is that a powerful common object model should appear as natural as possible to an application developer using a native language such as C++ or Modula-3. It is possible that most of the existing languages support a feature, and the feature is supported in a common object model. There will be languages that do not support some of the features of a common object model. In such a case, there should at least be a provision in the common object model translation framework for one-way interoperability which allows that language to use the components from other languages developed using the same feature. For example, Modula-3 does not support method overloading. But the ICOM framework in Modula-3 allows Modula-3 code to use this feature by letting it to make invocations on the components of other languages that have overloaded methods. This is very important because application developers are not limited to reusing just the code that is developed without method overloading.

5.2 Method Overloading

Method overloading is an aspect of ad hoc polymorphism. An interface with overloaded methods defines more than one method with the same name but different sig-

natures. The types and/or count of the parameters are different in each overloaded method. In most statically typed object-oriented languages, method overloading is achieved by defining multiple methods with the same name but different signatures. An alternative way of achieving method overloading is to inherit from a base class and define methods (in the derived class) whose names match with those in the base class. But, the signatures of the newly defined overloaded methods in the derived class must be different.

Method overloading is different from method overriding. In overriding, a base class method signature is the same as the overriding method in the derived class. An overriding method hides the overridden method. An explicit super class syntax is needed to invoke the method defined in the superclass. In method overloading, the base class method is not hidden. All methods are visible to a client. The method binding for a method invocation of an overloaded method can be done at compile time.

The ICOM framework supports the method overloading feature in its distributed object model. Overloaded methods can be described in an IDL interface. The IDL translator of ICOM generates metaclasses that are capable of identifying the correct method to invoke, based on the decoded arguments information from a message.

As mentioned in the previous section, a major design criterion in the ICOM framework is that a distributed object model should appear as close to the native language as possible. Some languages such as BETA and Modula-3 do not have the method overloading feature in their object models. But they have other features using which method overloading can be added to the language, but at the expense of 'name mangling' schemes being visible to the application developers. Also, 'name mangling' looks unnatural to a developer. So, even though it is possible to add the method overloading feature to these two languages, the ICOM framework chooses not to

translate the feature in these languages. Instead, ICOM provides a way that software components in these languages can access software components in other languages that support the method overloading feature. So, an application developer can use the feature from other languages, but cannot write code using this feature. Thus, the ICOM model provides one-way interoperability of the common object model features in languages that do not support the common object model features.

5.2.1 Example: Method Overloading in a Localized Object

In this subsection, an example for the 'Department Stores' problem listed in Chapter 3 is used to describe method overloading. Figure 5.1 shows a ClothesStore interface that contains three methods. The first two methods have the same name, priceOfItem, but different signatures. The first method takes an itemNumber as its first argument, whereas the second method takes an itemLabel as its first argument. These two parameters differ in their types. The method priceOfItem is used by a client to determine the price of an item. The third method of the interface is putItemOnSale. It is used to find the sale percentage on an item whose inventory remains the same for a long period of time, so that the item can be sold quickly.

During the IDL translation, each overloaded method will be numbered with a natural number as shown in Figure 5.2. If there are three methods with the same method name in an interface, they will be numbered as methods 1, 2 and 3. Both the client side and the server side agree upon the number assigned to each method. The IDL translator allocates method numbers in the order the methods appear in the IDL description.

At the receiving side (say remote receiving side) the metaclass instance looks up in its method table with the method name. Then the metaclass instance passes the

```
interface ClothesStore {

    // returns price of an item through 'price'
    priceOfItem(in long itemNumber,
                out double price);
    // returns price of an item through 'price'
    priceOfItem(in string itemLabel,
                out double price);
    // returns the sale percentage through 'salePercentage'
    putItemOnSale(in long itemNumber,
                  out double salePercentage);
}
```

Figure 5.1: IDL Description of Clothes Store Interface Containing Overloaded Methods

argument list to the method object. The method object decodes the first integer argument, which is the method number. With this information the method object knows which signature of the overloaded method to use. Using its own state information, the method object then decodes the rest of the arguments from the message object. The method object then invokes the correct method on the active object.

The IDL translator generates two classes on the client side: ClothesStore proxy class and ClothesStoreMessageClass. ClothesStoreMessageClass is different from a normal message class that is generated for non-overloaded methods in an interface. To create a message object for the method priceOfItem, the translator adds an extra argument to the argument list. In the case of an invocation of method priceOfItem(itemNumber, price), the translator generates the extra argument that has value 1 representing the first method in the interface. In the second case, it generates the argument with value 2, representing the second method.

When an invocation is made on an instance of ClothesStore, with the parameters of an item number, 'itemNumber' and a future variable, 'price' (arrow labeled 1), the

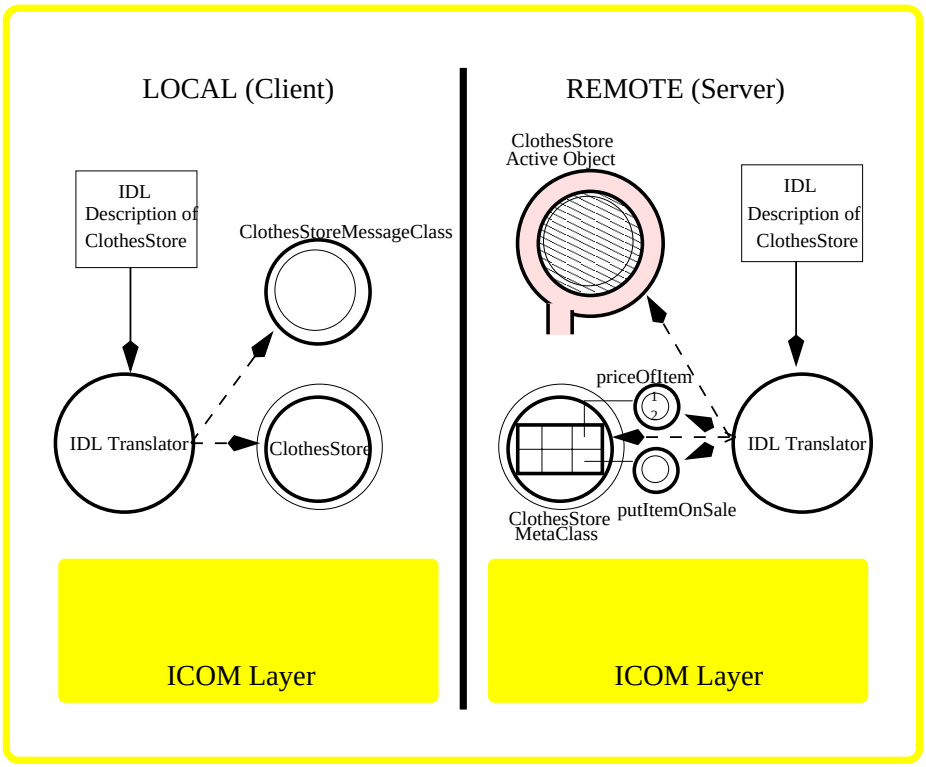

Figure 5.2: IDL Translation of Clothes Store Interface Containing Overloaded Methods

sequence of method calls are explained as follows:

At the local node

An invocation message is created by the ClothesStore instance using the ClothesStoreMessageClass whose class name and method name are set to ClothesStore and priceOfItem respectively. It takes the arguments list generated by the message creator class when invoked for message priceOfItem. The arguments list contains the method number as the first argument. In the example it is 1. This invocation message is encoded into a network representation (XDR) representation and sent to the server side (arrows labeled 2 and 3).

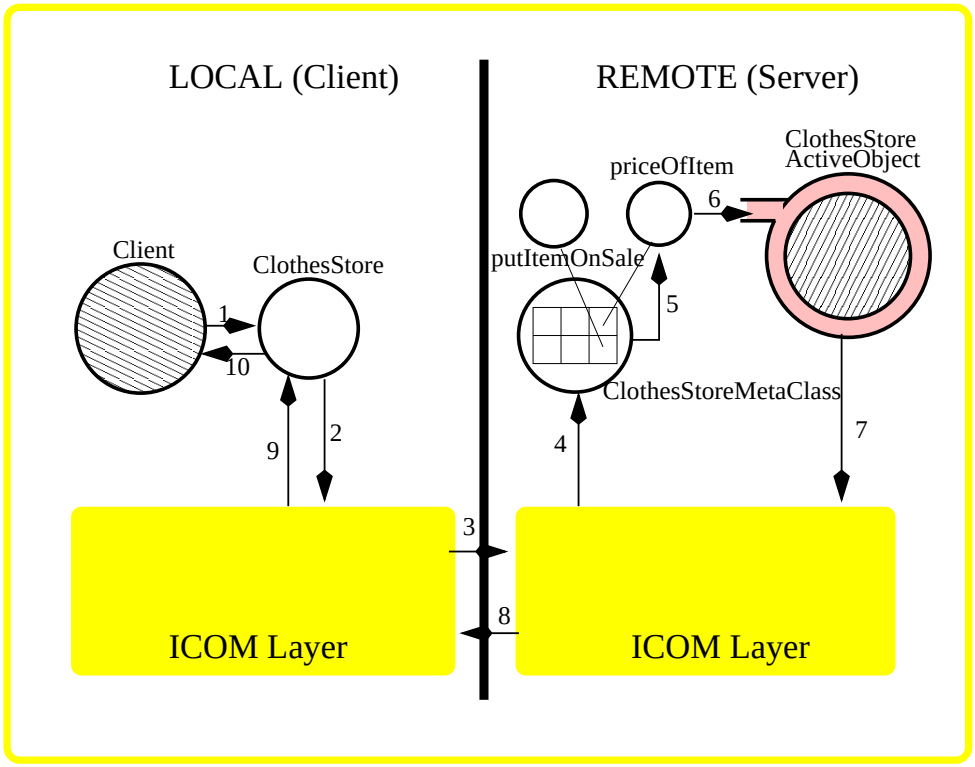

Figure 5.3: Dynamic Method Binding on an Instance of Clothes Store Interface Containing Overloaded Methods

At the remote node

At the server side, the encoded message is received. It is partially decoded to get the network address of the ClothesStore ActiveObject, and the method name and the class name for the method, priceOfItem. The ICOM layer uses the 'metaclass table' to get the instance of the metaclass, ClothesStoreMetaClass. It then passes the partially decoded message to the metaclass instance (arrow labeled 4). The metaclass instance, using its method table, looks up for a method object, and passes the message object to the server method object, priceOfItem (arrow numbered 5). This method object decodes the first argument

to get the number of the overloaded method to be invoked. It realizes based on the decoded value that it is the first method of the interface ClothesStore. The method object then decodes the remaining arguments, and invokes the correct method of the ClothesStore implementation (arrow numbered 6).

During the method execution, the output parameter of the method priceOfItem will be filled with a value at the ClothesStore Active Object implementation. Since it is an output parameter, internally it is represented as a future variable. So, the future variable at the client application receives the value which is set at the server side (arrows numbered 7, 8, 9, and 10). The client application thread blocked on the future variable can continue its execution.

5.2.2 Example: Method Overloading in a Distributed Object

In the previous subsection, method overloading in a localized object is explained in detail. Another way of achieving method overloading in an object is by inheritance. A method defined in a base class can be re-defined with the same name but different signature in a derived class. Both methods are visible for a client of a subclass instance.

In the 'Department Stores' problem, it is possible that two interfaces, as shown in Figure 5.4, have methods with the same name. The interface ClothesBase can be implemented at the CIPS center and the interface ClothesDerived can be implemented at a remote location, for instance, Virginia. The interface ClothesBase defines a method priceOfItem that takes an input parameter, itemNumber, and has an output parameter price. The derived interface ClothesDerived defines one method. This method has the same name as the method defined in the base interface ClothesBase. But the signatures of the method are different. The method in the derived interface takes an

input parameter, itemName, and an output parameter, price. So, the first arguments of both methods with the name priceOfItem differ in their type. In this example, the price of an item can be determined with the item number by using the method implemented in the base interface. It can also be determined by using an item label using the derived interface implementation. This is a useful mechanism because a designer of the derived interface need not invent new names for the methods providing the same functionality.

```
interface ClothesBase {
    // returns price of an item through 'price'
    priceOfItem(in long itemNumber,
                out double price);
    ⋮
}

interface ClothesDerived : reuse ClothesBase {
    // returns price of an item through 'price'
    priceOfItem(in string itemLabel,
                out double price);
    ⋮
}
```

Figure 5.4: IDL Description of Clothes Store Distributed Object Containing Overloaded Methods

The IDL translation scheme used for the above example is shown in Figure 5.5. As shown in the figure, the translation at the local site with the derived interface involves invocation of the translator at the remote site. The translation mechanism is similar to a distributed object translation shown in Chapter 3.

Assume that the base interface has already been implemented at the remote node. When an IDL translator is invoked at the local node, the classes generated relevant to the ClothesBase interface are: ClothesBaseProxyMetaClass, ClothesBaseProxyImple-

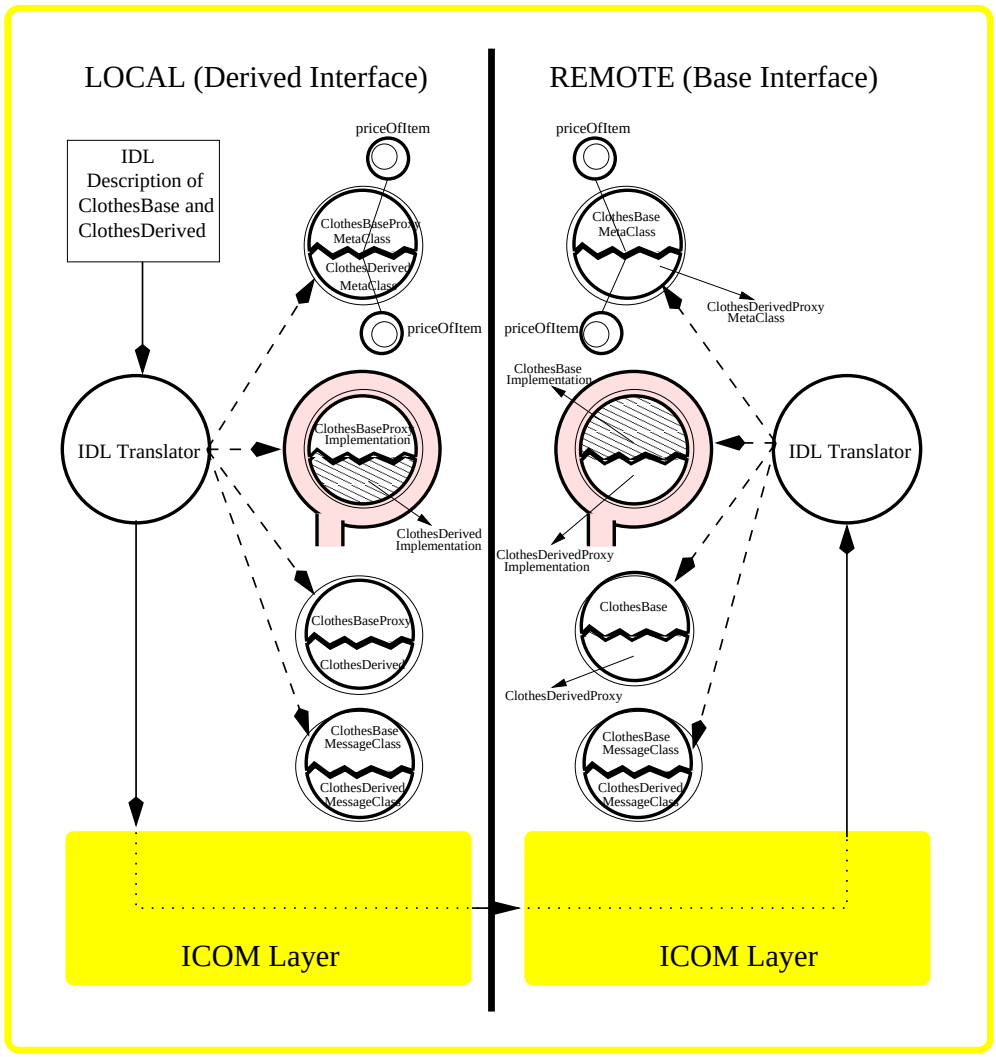

Figure 5.5: IDL Translation of the Clothes Store Distributed Object Containing Overloaded Methods

mentation, ClothesBaseProxy, and ClothesBaseMessageClass. The classes generated relevant to the ClothesDerived interface are : ClothesDerivedMetaClass, ClothesDerived-Implementation, ClothesDerived, and ClothesDerivedMessageClass. The classes relevant to the derived interface inherit from their counterparts in the base interface. The representation is shown using a cracked egg shell notation. The patterned portions in the figure represent the code that must be implemented by the developers. The Clothes-DerivedMetaClass contains two entries for the method priceOfItem in its method table. The first entry contains the method name, 'priceOfItem', the class name, 'Clothes-Base', and a method object that can decode the arguments of the method, priceOf-Item, defined in the ClothesBase interface. The second entry contains the method name, 'priceOfItem', the class name, 'ClothesDerived', and a method object that can decode the arguments of the method, priceOfItem, defined in the ClothesDerived interface. Even though these two methods are overloaded at a logical level, they can be identified separately because each message object contains both the method name and the class name for the invoked method.

The IDL translator at the local node, realizing that the inheritance involves a remote interface, sends a 'compile' message to the remote IDL translator. The compile message contains details about the inheritance hierarchy involved using the base and derived interfaces. At the remote node, the translator generates four classes corresponding to the derived interface ClothesDerived. These classes are: Clothes-DerivedProxyMetaClass, ClothesDerivedProxyImplementation, ClothesDerivedProxy, and ClothesDerivedMessageClass. The translator reuses the existing translator-generated classes for the remote interface ClothesBase. The newly generated classes at the remote node inherit from their counterparts of the base interface. For example, Clothes-DerivedImplementation class inherits from ClothesBaseProxyImplementation class. The translator then compiles the generated classes and links the base interface implemen-

tation code to generate an executable for the 'ClothesStore PeerProxy' at the remote node.

The application developer at the local node must provide implementation for the derived interface, compile it with the native language compiler and link the implementation code with the translator generated code to create an executable. The ClothesDerived class can be included in any application at the local node and used to create its instances and invoke methods on them.

A method invocation of the method priceOfItem with the first argument as an item number, on an instance of ClothesStoreDerived class, results in a series of method invocations at the local as well as remote nodes. This series of method invocations is shown in Figure 5.6.

In this example, the client application and the implementation of the interface ClothesDerived are assumed to be on a local node. The implementation of the interface ClothesBase is assumed to be on a remote node. After the translation phase is over, the proxies for the distributed object of ClothesDerived are created at both nodes. When the client application invokes the method priceOfItem with the first argument, 'item number', and the second argument, 'price' (arrow numbered 1), the sequence of method calls is as follows:

At the Client Site

The instance of ClothesDerived creates a message object with the method name, 'priceOfItem', and class name, 'ClothesBase', along with the arguments. This message is delivered to the metaclass instance, ClothesDerivedMetaClass (arrow labeled 2). The metaclass instance passes the message to the method object instance of priceOfItem for the first entry of the method table (arrow labeled 3). The method name and class name for the first entry are 'priceOfItem' and

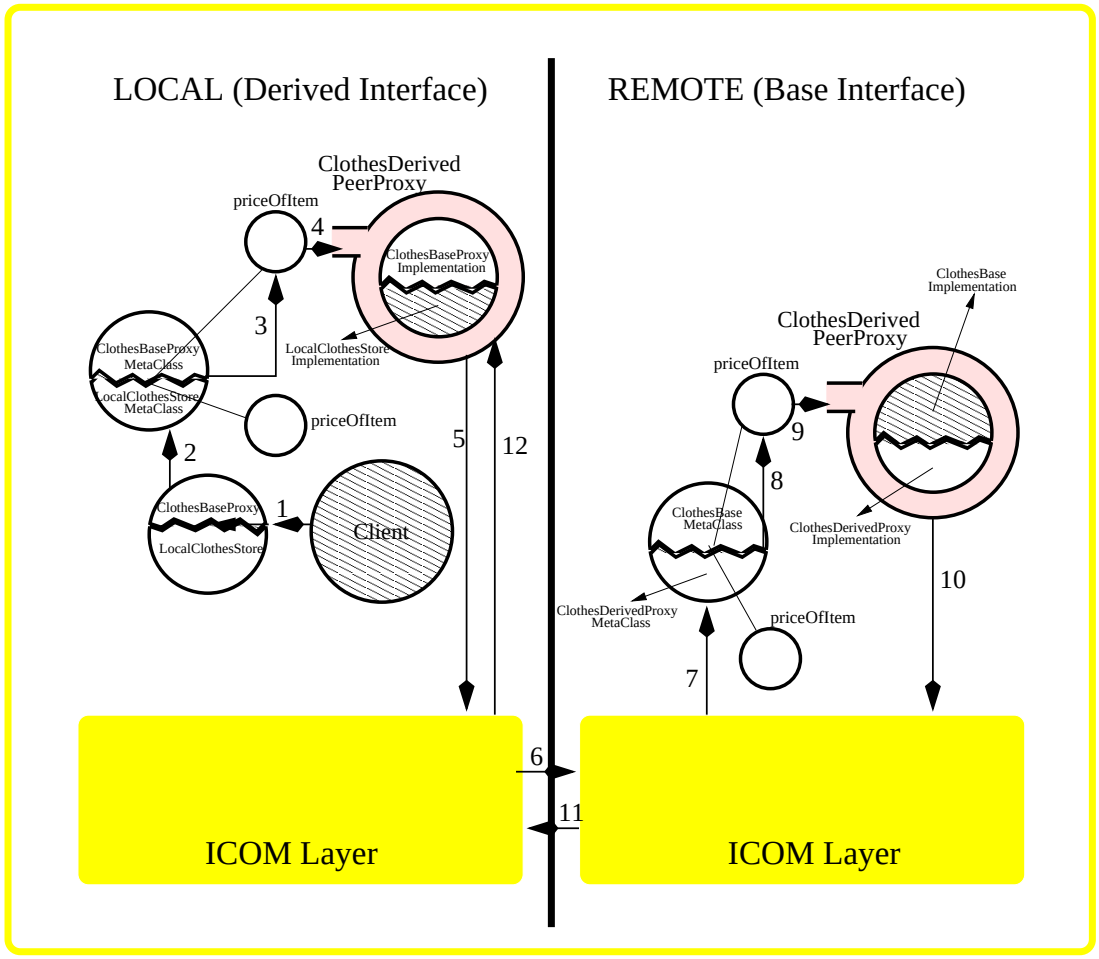

Figure 5.6: Dynamic Method Binding on a Clothes Store Distributed Object Containing Overloaded Methods

'ClothesBase' respectively. The method object copies the arguments from the message object, and hands the message to the ClothesDerivedPeerProxy instance. The peer proxy has a stub method for the corresponding method implemented at the remote node. The message is encoded and sent to the remote node (arrows labeled 5,6, and 7).

At the Server Site

The ICOM layer at the remote node partially decodes the message to extract the network address of the peer proxy, and the method name and the class name contained in the message. The peer proxy does a lookup operation in the 'metaclass table' to locate the instance of the metaclass, ClothesDerivedProxyMetaClass (arrow labeled 7). The metaclass instance, using the method name and the class name, looks up in the method table for a server method object. The metaclass instance then passes the message object to the server method object to decode the arguments (arrow labeled 8). The method object decodes the arguments and invokes the priceOfItem method of the ClothesDerivedPeerProxy (arrow labeled 9). The implementation of the method at the remote node executes, setting the value of the output parameter price. Since, every output parameter is a *future* variable, a message is returned to the local node setting the value of the corresponding future variable in the client application (arrows labeled 11 and 12). At this point, the client can proceed with its execution if it has been waiting for the future variable value.

Thus, the ICOM framework supports the method overloading feature of the ICOM object model. In languages that do not support method overloading, the framework supports one-way interoperability. The one-way interoperability allows these languages to use software components of other languages that use method overloading.

5.3 Parameterized Types

Type parameterization is used to allow a generic type to be instantiated with one or more types. For example we may define a class to implement a generic stack data type. This type can be instantiated with other types such as the basic types, integers and characters, to produce stacks of integers and stacks of characters. The same generic type can be instantiated with user defined types such as a database record to produce stacks of database records. Support for parameterized types exists in several popular languages: templates in C++, generics in Ada 95, Modula-3, and Eiffel. Experience with the use of type parameterization in software development shows that the ability to define and instantiate a general concept is useful in the design and construction of reusable software components. The prevalence of type parameterization in several popular languages, and recently the efforts to add parameterized types to the language Java shows the general applicability of parameterized types. Parameterized types can play an important role in the development of class hierarchies. A popular example is the C++ standard template library.

The generic type scheme, *constrained genericity*, is supported in the ICOM framework. Constrained genericity requires the interface of a parameter to be defined while defining a generic type that uses the parameter. It is not as flexible as a C++ template scheme (called *unconstrained genericity*) where the behavior of a template parameter is not required while defining a template. A C++ compiler can parse the source code of the template (generic type) implementation to determine the interface of the template parameter. The reason for choosing constrained genericity in the ICOM framework is because of the design criterion of the ICOM object model that language compilers should not be changed. If unconstrained genericity were to be supported, the source code of a native language that implements the generic type must be parsed

to extract the interface of the parameter, which is undesirable. Also, some of the statically typed object-oriented languages do not support unconstrained genericity. In ICOM, a parameter interface must be present for a generic type to be defined with the parameter. In other words, the interface of the parameter must be known at compilation time. But, the generic types in ICOM are powerful in the sense that a generic type can inherit from its parameter. The C++ language supports inheritance of template parameter type, but other languages such as Modula-3 do not support it.

Parametric types can be implemented in two ways. A separate implementation can be generated for each type instantiation, as in C++ [166], or a single implementation can be used for all type instantiations, as in ML [132], depending on how the language semantics are defined. For example, 'static' variables play different roles in these two types of implementations. In the first way, each type instantiation will have a separate static variable, whereas in the second way, only one static variable will be used for all type instantiations. The ICOM framework uses the first method of implementation, separate implementation for each type instantiation.

5.3.1 Example: Parameterized Types without Inheritance

In this section, the parameterized types of the ICOM object model are explained with an example. This example does not use any inheritance relationships. Figure 5.7 shows a representation of the 'Department Stores' problem discussed in Chapter 3. This figure assumes a system distributed across four states in the United States of America. The central information processing center (CIPS) is assumed to be in New York. A grocery store is in Kentucky and a clothes store is in Virginia. A manufacturer is assumed to be in Georgia. In the figure, parameterized types are shown as smaller circles. The overlapping circle is the parameter. The interface of the pa-

rameter type must be present along with the IDL description of the parameterized type StoreTemplate. The cracked egg shell fragments shown represent instantiation of the parameterized type with a truly distributed object. The GroceryBase, Tax, and ClothesBase interfaces are implemented at the New York site. The GroceryDerived and KentuckyTax are implemented at the Kentucky site. The VirginiaTax, and Clothes-Derived interfaces are implemented at the Virginia site. The Manufacturer interface is implemented at the Georgia site. The interfaces GroceryDerived and KentuckyTax inherit from GroceryBase and Tax interfaces, respectively. The interfaces VirginiaTax and ClothesDerived inherit from Tax and ClothesBase respectively.

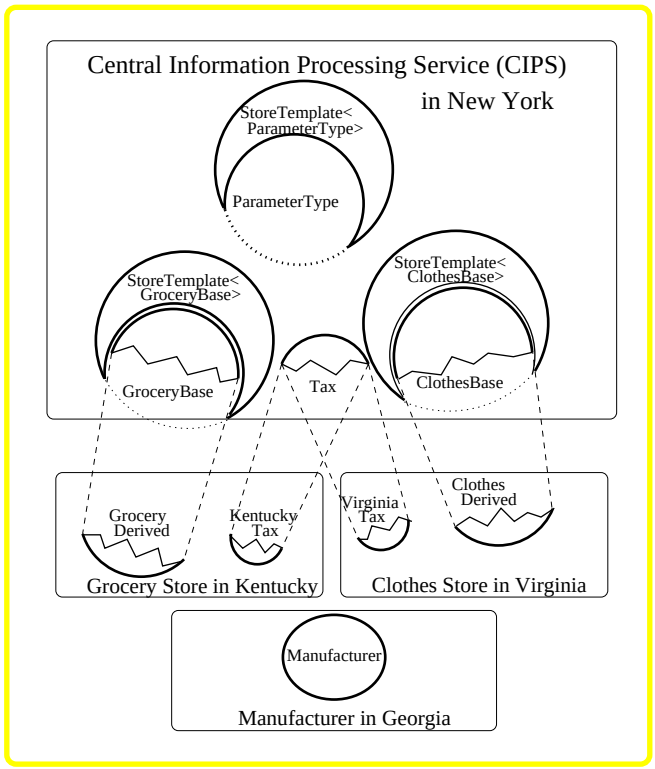

Figure 5.7: Parameterized Types

The central information processing center (CIPS) can define default interfaces

for grocery store, clothes store, and tax at the New York site. Through inheritance, these interfaces can be specialized for local stores in Kentucky and Virginia. From the perspective of the financiers of the chain stores, all stores exhibit the same behavior; all stores have the same interface containing the net profits and gross sales. So, the individual stores can be abstracted into a StoreTemplate generic type that has a general store behavior.

Figure 5.8 shows an instantiation of the generic type StoreTemplate with the interface ClothesStore, and three interfaces, StoreType, StoreTemplate, and ClothesStore. The StoreType is a parameter to the generic type, StoreTemplate. It supports two methods, monthlySales and quarterlySales. The monthlySales method takes two input parameters, monthNumber and yearNumber, and an output parameter, amount. The quarterlySales method takes two input parameters, quarterNumber and yearNumber, and an output parameter, amount. The StoreTemplate generic type defines two methods, netProfit and grossSales. The netProfit method takes an input parameter, yearNumber and an output parameter, amount. Similarly, the grossSales method takes an input parameter, yearNumber and an output parameter, amount. The interface ClothesStore can be used to instantiate the generic type because it supports the methods defined in the interface, StoreType. In addition to the two methods of the StoreType interface, the ClothesStore interface supports two other methods, priceOfItem and putItemOnSale. The first method is used to find out the price of an item given an item number and the second method is used to determine whether to keep an item on sale or not.

In this example two distributed nodes are assumed, local and remote. At the remote node, an implementation of the StoreTemplate is provided. Since the parameterized types are constrained, the types of the parameters must be known while defining a parameterized type. The StoreType interface must be defined before defining the

```
interface StoreType {
   // returns sales in a month,  of a store through 'amount'
   monthlySales(in int monthNumber, in int yearNumber,
                out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                   out double amount);
}

generic_interface StoreTemplate ⟨StoreType⟩ {
    // returns net profit through 'amount'
    netProfit(in int yearNumber, out double amount);
    // returns gross sales through 'amount'
    grossSales(in int yearNumber, out double amount);
}

interface ClothesStore {
    // returns sales in a month,  of a store through 'amount'
    monthlySales(in int monthNumber, in int yearNumber,
                 out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                   out double amount);
    // returns price of an item through 'price'
    priceOfItem(in long itemNumber, out double price);
}

 // an instantiation of the generic type
 StoreTemplate ⟨ClothesStore⟩ clothesStoreA;
```

Figure 5.8: IDL Description of the Clothes Store Generic Interface

StoreTemplate. So, at the remote node the StoreTemplate definition and implementation are available along with StoreType interface.

The IDL translation of the StoreTemplate ⟨ClothesStore⟩ instantiation is shown in Figure 5.9. The local node defines the ClothesStore interface that conforms to the StoreType interface. A client application also resides at the local node. To instantiate the StoreTemplate at the remote node using the ClothesStore available at local node, a client developer first declares the instantiation in IDL. The description of StoreTemplate is also available in IDL to the client developer.

The IDL translator generates a proxy class for the StoreTemplate. The client

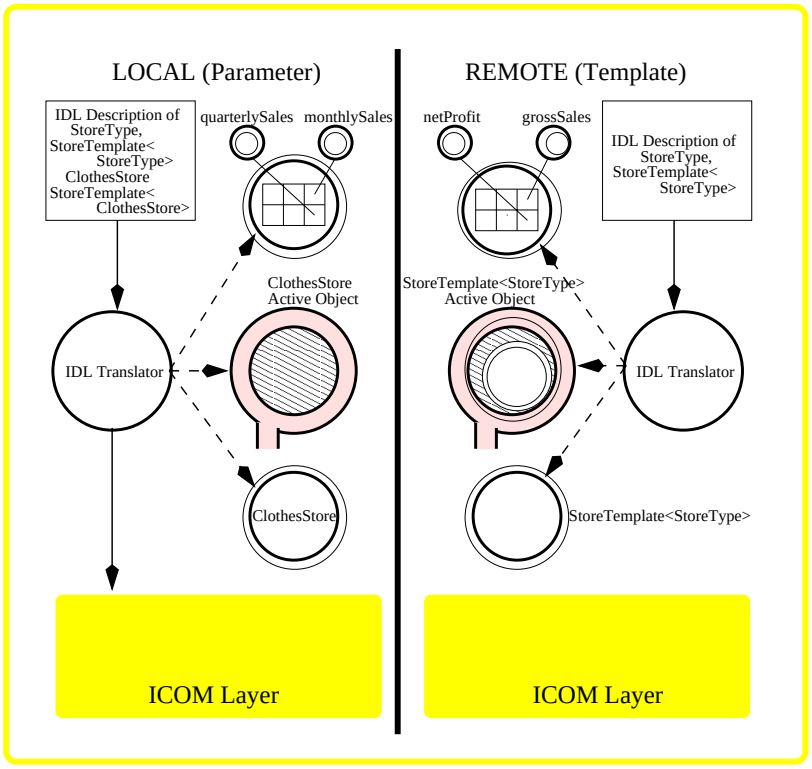

Figure 5.9: IDL Translation of Clothes Store Generic Interface

developer can use the proxy class to create an instance of the class which is an instantiation of the StoreTemplate with StoreType. The constructor for the proxy class takes the name and location of the ClothesStore interface and the node where it is implemented. The ICOM framework receives a constructor message that has the state information with name value pairs methodName: create, className:StoreTemplate, parameterName:ClothesStore, parameterLocation:local, and other constructor arguments for the StoreTemplate if any. This message is encoded into a common network notation (XDR) and transferred to the remote node. At the remote node, the message is partially decoded. An instance of StoreTemplate is created. This instance contains the information about the ClothesStore and its location. The StoreTemplate instance

creates an instance of StoreType, which acts as a handle to ClothesStore. The instance of StoreType in turn creates an instance of ClothesStore at the local node. Any method invocations on this handle will be sent to the ClothesStore implementation at the local node.

An invocation of the method grossSales on an instance of the parameterized type, StoreTemplate ⟨ClothesStore⟩ (arrow labeled 1) is shown in Figure 5.10. The patterned portions of the figure represent the code that must be implemented by the application developers. The method sequence is explained as follows:

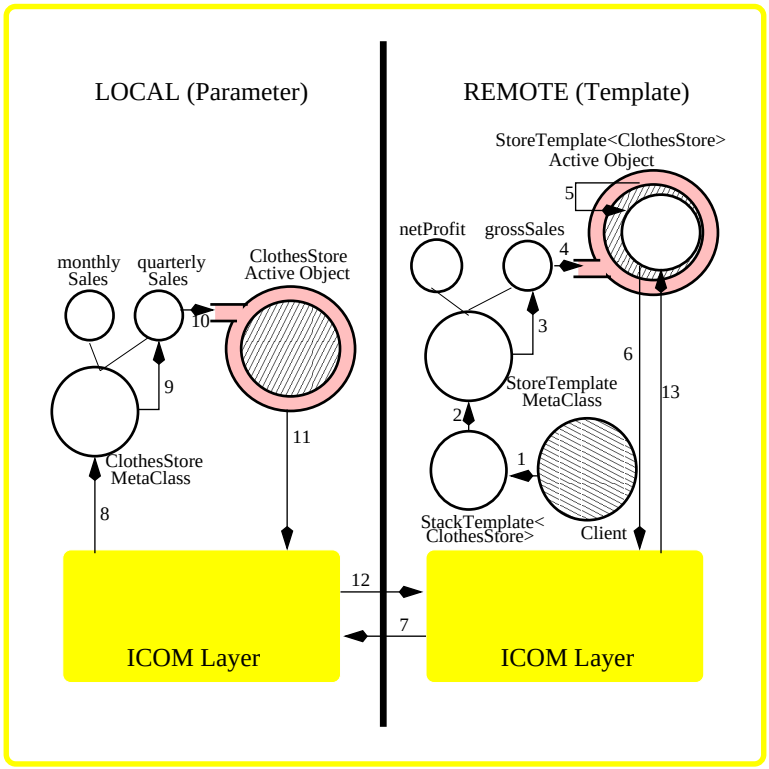

Figure 5.10: Dynamic Method Binding on an Instance of Clothes Store Generic Interface

At the remote node

The handle object which is an instance of StoreTemplate ⟨ClothesStore⟩ creates a message object for the method grossSales and sends it to the instance of StoreTemplateMetaClass (arrow labeled 2). The metaclass instance looks up in its method table for a method object, and passes the message object to the server method object for decoding of the method arguments (arrow labeled 3). The grossSales method object copies the method arguments (because the invocation is a local invocation) and invokes the grossSales method of the StoreTemplate ⟨ClothesStore⟩ instance (arrow labeled 4). The implementation of the method at the remote node, calls the quarterlySales method of the StoreType interface instance four times to get the gross sales for a particular year (arrows labeled 6,7).

At the local node

The ICOM layer receives the encoded message object for the method quarterlySales. This layer partially decodes the message to obtain the network address of the ClothesStore instance that is used as an argument to the generic type instantiation at the remote node. The message is then passed to the instance of ClothesStoreMetaClass (arrow labeled 8). The metaclass instance does a look up in its method table for a method object and passes the message object to the method object quarterlySales (arrow labeled 9). The method object decodes the message arguments and invokes the quarterlySales method on the instance of ClothesStoreImplementation class (arrow labeled 10). The output argument amount is assigned a value which is forwarded to the StoreTemplateImplementation instance at the remote node (arrows labeled 11,12, and 13). The client application eventually receives the result via the output parameter amount that was passed in the method call of grossSales.

Thus, generic types defined at a remote node can be used to instantiate a type

that is compatible with the parameter of the generic type.

5.3.2 Example: Parameterized Types with Inheritance

In this subsection, a simple example of a parameterized type (taken from the 'Department Stores' problem) that uses inheritance is explained. The same example used in the previous section is modified to describe inheritance involving generic types.

```
interface StoreTemplateBase {
    // returns net profit through 'amount'
    netProfit(in int yearNumber, out double amount);
    // returns gross sales through 'amount'
    grossSales(in int yearNumber, out double amount);
    // helper routine used by 'netProfit' and 'grossSales'
    calculate(in int yearNumber, in double inAmount, out double outAmount);
}

interface StoreType {
    // returns sales in a month,  of a store through 'amount'
    monthlySales(in int monthNumber, in int yearNumber,
                 out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                   out double amount);
}
generic_interface StoreTemplate ⟨StoreType⟩ : reuse StoreTemplateBase {
    // returns net profit through 'amount'
    netProfit(in int yearNumber, out double amount);
    // returns gross sales through 'amount'
    grossSales(in int yearNumber, out double amount);
}
interface ClothesStore {
    // returns sales in a month,  of a store through 'amount'
    monthlySales(in int monthNumber, in int yearNumber,
                 out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                   out double amount);
    // returns price of an item through 'price'
    priceOfItem(in long itemNumber, out double price);
}
 // an instantiation of the generic type
 StoreTemplate ⟨ClothesStore⟩ clothesStoreA;
```

Figure 5.11: IDL Description of Clothes Store Generic Interface involving Inheritance

Figure 5.11 shows four interfaces and an instantiation of a generic interface. The interface StoreTemplateBase defines three methods, grossSales, netProfit, and calculate. The methods grossSales and netProfit use the method calculate to finalize the gross sales and net profit at a store. The interface StoreTemplateBase is an example of a situation where the future development of software is not known when the software is developed. When a single store grows into a national chain, the abstraction of one store should be elevated to the abstraction of a chain of stores while reusing the existing code. In this case, the StoreTemplateBase interface can be inherited by a generic type StoreTemplate ⟨StoreType⟩ and the StoreType interface can be defined as a parameter type interface. The figure also shows three other interfaces, StoreType, StoreTemplate ⟨StoreType⟩, and ClothesStore. The generic type StoreTemplate ⟨StoreType⟩ inherits from the interface, StoreTemplateBase. This derived generic interface redefines the two methods of its base interface, grossSales and netProfit. The rest of the interfaces are similar to the example discussed in the previous section.

In this example, it is assumed that the interfaces, StoreTemplateBase, StoreTemplate ⟨StoreType⟩, and StoreType are implemented at the remote node. A client application also exists at the remote node. The interface, ClothesStore is assumed to be implemented at the local node. IDL translators can be independently invoked at both nodes. Since there are no distributed objects in this example, no exchange of information takes place between the translators at compilation time. Assume that the interface ClothesStore has been implemented after translating the IDL description of that interface into a native language at the local node.

The IDL translator at the remote node generates the classes: StoreTemplateBaseMetaClass, StoreTemplateBaseImplementation, StoreTemplateBase, and StoreTemplateBaseMessageClass for the interface StoreTemplateBase. The translator also generates the classes: StoreTemplateMetaClass, StoreTemplateImplementation, StoreTem-

plate, and StoreTemplateMessageClass for the generic interface StoreTemplate ⟨StoreType⟩. These classes inherit from the corresponding classes of the interface StoreTemplateBase. The translator generates, StoreTypeImplementation, and StoreTypeMessage classes for the parameter type StoreType.

The last declaration in the IDL description creates an instance of the generic type StoreTemplate ⟨StoreType⟩. It is instantiated with the type ClothesStore. The IDL translator generates a class StoreTemplateClothesStore that represents the instantiation of the generic type.

A statement in a client application that creates an instance of the class StoreTemplateClothesStore generates an instance of the generic type StoreTemplate that is associated with an instance of ClothesStore.

Figure 5.13 shows an example invocation of the method grossSales on the instance of StoreTemplate ⟨ClothesStore⟩ (arrow numbered 1). The sequence of calls can be explained as follows:

At the remote node

The instance of StoreTemplate ⟨ClothesStore⟩ creates a method object and passes it to the metaclass instance, StoreTemplateMetaClass (arrow numbered 2). The metaclass instance looks up in its method table and passes the message to the method object instance, grossSales (arrow numbered 3). The method object copies the argument from the message object (since the method invocation is local) and invokes the method on the generic type implementation (arrow numbered 4). The StoreTemplateImplementation in turn makes an invocation on the method quarterlySales of the StoreType instance four times to calculate the gross sales for a year (arrow numbered 5). The instance of StoreType acts like a proxy for the instance of ClothesStore at the local node. It forwards the method call to the local node (arrows numbered 6 and 7).

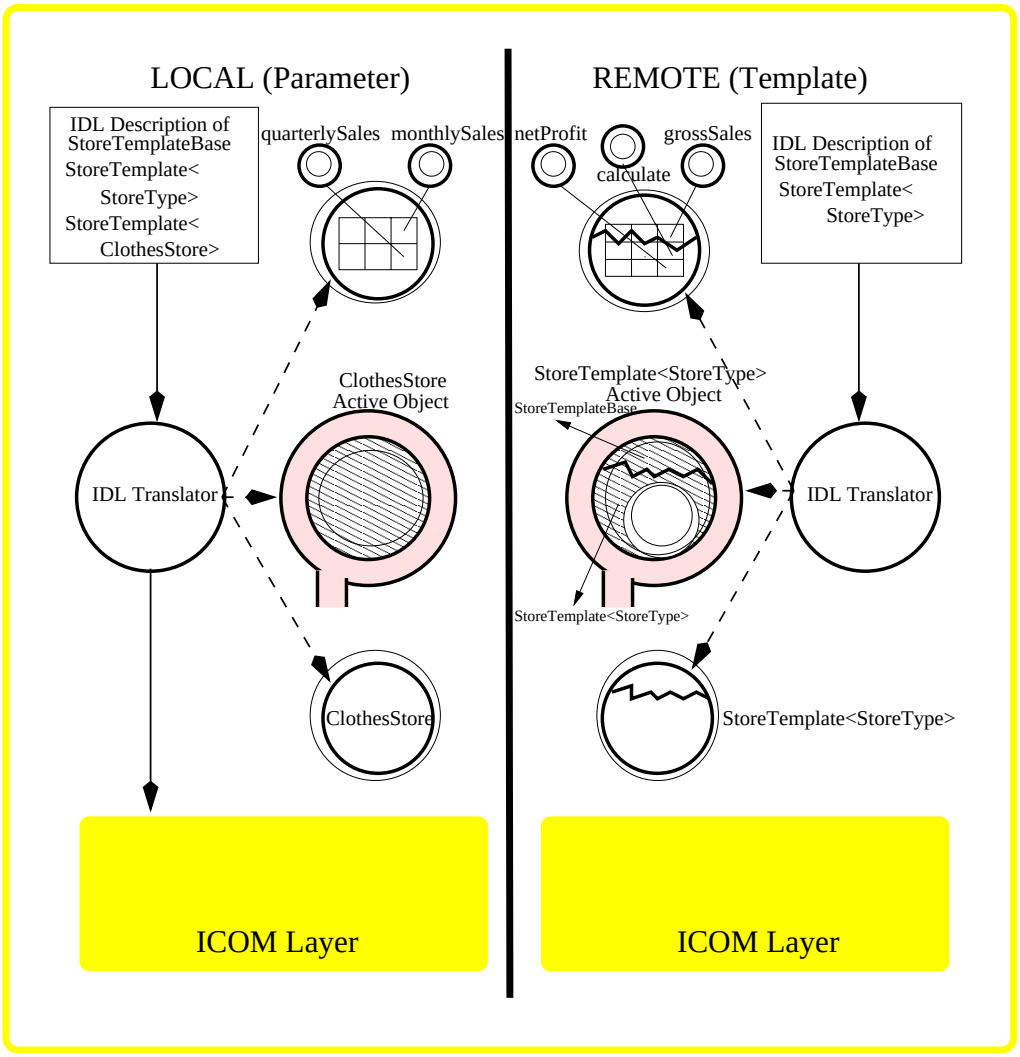

Figure 5.12: IDL Translation of Clothes Store Generic Interface involving Inheritance

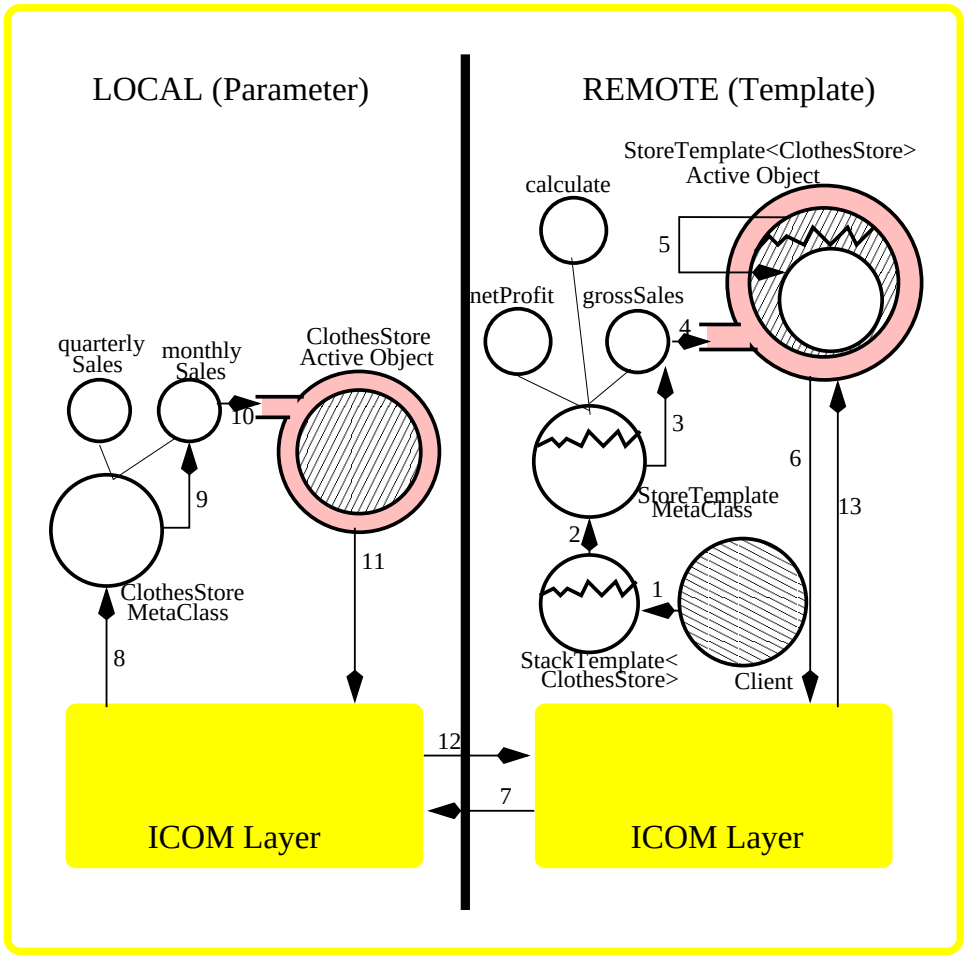

Figure 5.13: Dynamic Method Invocation on an Instance of Clothes Store Generic Interface Involving Inheritance

At the local node

The ICOM layer at the local node receives the encoded message and partially decodes it to get the network address of the active object of ClothesStoreImplementation. The partially decoded message is forwarded to the meta class instance of ClothesStoreMetaClass (arrow numbered 8). The metaclass does a lookup in its method table for a method object and passes the message to the method object, quarterlySales (arrow numbered 9). The method object decodes the arguments and invokes the quarterlySales method on the active object of ClothesStoreImplementation (arrow numbered 10). The output parameters are represented as future variables similar to the previous example. These values are returned to the remote node (arrows numbered 11, 12, and 13). The gross sales for the required year are returned to the client application from the StoreTemplate ⟨ClothesStore⟩ active object. The client can proceed with the execution if it was waiting on the results of the method call.

Thus, generic types that inherit from other types can be used to instantiate with a type that is compatible with the parameter type of the generic type.

5.4 Summary

In this chapter, the advanced features of method overloading added to the ICOM object model, is explained in detail. Examples that illustrate the power of this feature are illustrated. In languages that do not support the method overloading feature, one-way interoperability can be used to interoperate with software components written in other languages. The inclusion of generic types into the ICOM object model is described. Generic types that use inheritance are also explained with examples taken from the 'Department Stores' problem.

Chapter 6

Concurrency

6.1 Introduction

Concurrency is the ability of a system to have more than one of its software components (that are inherently sequential) in execution simultaneously. Concurrency in a program can be viewed as a set of sequential processes executing in a parallel fashion, either all on a single processor or on separate processors. Concurrency on a single processor can be achieved by interleaving multiple sequential processes as has been a common practice with 'multiprogramming' used in operating systems [33]. For example, a process waiting for an input/output event to happen can relinquish control over the central processing unit (CPU) to another process waiting to use the CPU. When this new process that has control over the CPU requires an input/output event, it can give up control over the CPU and another waiting process can take control. This can continue until all processes complete their execution. To avoid starvation, where a process never gets control of the CPU because of another computation intensive process with minimal input/output has control over the CPU all the time, 'time

slicing' can be used where each process is allocated a fixed time during which it has control over the CPU [159]. Thus, with concurrency, the CPU can be better utilized and multiple processes can share the CPU for achieving a common goal in a shorter time compared to the sequential execution of these processes executing one after the other.

In case of distributed systems, there is a strong opportunity for concurrency with interleaving because of the latencies involved in message transmissions. Processes invoking methods that are implemented on remote sites, can continue with their execution, and block for the results of the remote invocation only when necessary.

6.1.1 Synchronization

It is often the case that multiple concurrent processes need to coordinate with one another to achieve a common goal or to avoid interfering with the progress of one another. In such situations, concurrently executing processes need to communicate with one another. If mutually communicating processes need to agree that a certain event has to take place during the execution before they can proceed, then we say that those processes need to synchronize with one another. For example, two processes, *producer* and *consumer*, in case of a 'bounded buffer' example shown in Figure 6.1, synchronize with each other. If the buffer is empty, a consumer process waits until a producer process adds an item to the buffer. Similarly, if the buffer is full, a producer process waits until a consumer process removes an item from the buffer.

6.1.2 Types of Concurrency

There are two types of concurrency with regard to objects: *inter-object concurrency* and *intra-object concurrency*. Inter-object concurrency deals with simultaneous exe-

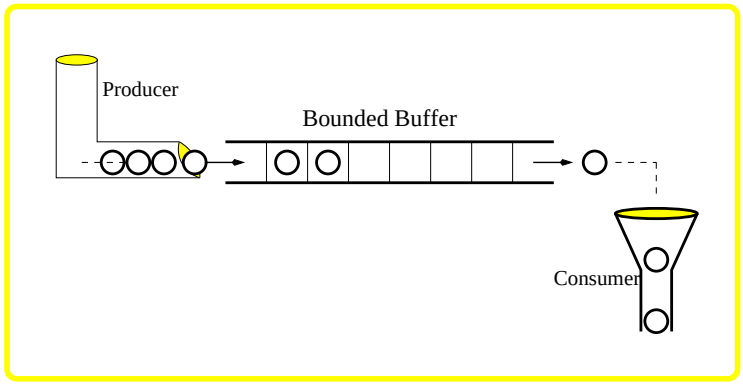

Figure 6.1: A Bounded Buffer

cution of methods among multiple objects. Intra-object concurrency involves concurrency within an object.

6.1.2.1 Inter-object concurrency

Inter-object concurrency is achieved through interleaving or simultaneous execution of methods of multiple objects. This may or may not involve multiple method simultaneous executions within an object. Distribution of objects strongly motivates the need for inter-object concurrency. For example, an object can continue to process message invocations after it sends a message to another remote object. The remote object can process the message in parallel with the sender object. Another example of inter-object concurrency is a distributed system based on the actor model of concurrent computation. Multiple distributed actors can process messages from their message queues simultaneously.

6.1.2.2 Intra-object concurrency

Intra-object concurrency involves the execution of more than one method of an object simultaneously. Three types of concurrent objects with reference to intra-object concurrency are identified.

- Atomic object — at most one method is in execution at any time.
- Quasi-concurrent object — more than one method can be executing in an object, but all of them will be in a suspended state except one.
- Concurrent object — more than one method can be in execution at a time.

Atomic objects require an *object-level locking* scheme to provide atomicity of method execution. Atomicity requires the complete processing of a message before another external message processing can start. A lock is typically associated with an object to provide atomicity. This lock can be controlled by either the invoking processes or by the object itself. Atomic objects are discussed in the next section.

Quasi-concurrent objects require monitor-like mechanism where primitive synchronization operations of wait and signal are available. These operations are used in the implementation of a method of the object. Quasi-concurrent objects are not chosen in this dissertation, because it involves language level details that cannot be considered with the given design criterion that language compilers and semantics should not be changed. Also, quasi-concurrent objects are not appropriate for a common object model.

Concurrent objects are further sub-divided into two categories - *fully concurrent* objects and *partially concurrent* objects. Fully concurrent objects do not require any locks at all. Any method can be executed at any time. Partially concurrent objects require support for a subset of methods to have atomicity. For example, in one

variation of the 'readers-writers' problem, the reader methods can be concurrently executed, but the writer methods must be atomic. The support for partially concurrent objects is in the form of information about all synchronized methods and history information of the method calls on the object. One way to realize this support is to use method level locks. This approach needs a lock for every group of atomic methods that require mutual exclusion with one another in the group. When a method starts to execute the corresponding group lock is set. In case of distributed objects, the owner of the lock will be decided prior to the execution. Once the execution starts, other external invocations of the same method are entered in the wait queue for that lock. The lock is unset when the method completes the execution. The completion of the method execution can be determined locally or by a remote peer sending a message to the lock's owner in case of distributed objects.

In languages like Java where a monitor scheme is used, individual methods can be locked and unlocked. Also a method in execution can communicate with other methods executing in their own threads, using 'wait' and 'signal' invocations of the monitor. Providing this kind of control in a distributed heterogeneous multi-language environment with the constraints that language translators should not be changed is a very tedious task.

An important design choice made in the ICOM framework is that only external invocations on an object can be synchronized. Any internal invocations of methods by the other methods of the same object are considered part of the external invocations and are not synchronized.

6.1.2.3 Inheritance Anomaly

In the previous sections, inter-object concurrency and intra-object concurrency are described. This section explains inheritance anomaly that is applicable to objects

involving inter-object concurrency and intra-object concurrency.

A concurrent object is said to be synchronized when it is programmed to accept only those messages that do not compromise the internal integrity of the object. A synchronized object can accept different subsets of its set of messages at different times. The code that controls the object's behavior is called the *synchronization code*. Various schemes have been proposed to express the synchronization code. Some of them are: semaphores, guards, and reflection techniques. In a typical synchronization scheme, synchronization code is merged with the class implementation code.

It has been pointed out that the synchronization code cannot be effectively inherited without non-trivial class redefinitions [113]. This problem is termed as the 'inheritance anomaly' and has been studied extensively [113]. Three main reasons for inheritance anomaly have been identified in [111]. They are described below.

Partitioning of acceptable states

An object is said to be in one of several 'states' at any time after its creation. These states can be divided into disjoint subsets based on its synchronization constraints. Each subset has a group of messages that are acceptable by the object in that state.

In the bounded buffer example, a buffer object has two messages that it supports: get and put. The method get takes an element out of the buffer and the method put adds an element to the buffer. The bounded buffer object has three states: empty, partial, and full. When the buffer is in the empty state the only acceptable method is put. When the buffer is in the partial state, the acceptable methods are put and get. When the buffer is in the full state, the only acceptable method is get.

Consider a subclass, xbuf, of the bounded buffer that adds a new method get2

which takes two elements out of the buffer at a time. The acceptable set of states is then split to have empty, partial, full, and x-one where x-one is the state that requires the buffer to have just one element in it. In the derived class, the acceptable methods for each state change. For the state empty the acceptable method is put; for state x-one the acceptable methods are put and get; for state full the acceptable methods are get and get2; for the state partial the acceptable methods are get, put, and get2. If the implementation of the base class has the synchronization code inside the method implementations, the addition of the get2 method in the subclass requires the redefinition of all the base class methods.

History only sensitiveness of acceptable states

Assume that a subclass of the 'bounded buffer' includes a new method gget which takes an element out of the buffer, but can be invoked only after a put message. Then an extra variable is needed in the subclass to keep track of the put and gget invocations. This also requires considerable rewriting of the code for inherited methods in the subclass. For example, the methods put and get must be rewritten.

Modification of acceptable states

In some cases it is possible that the addition of a new method may require modifications to the existing states instead of causing a split in the acceptable states. In the gget method addition, it is required to add a new state as well as modify the existing states.

There have been several proposals that addressed the inheritance anomaly. One such approach is *guard methods* used in [157]. A guard method is a boolean method that returns either 'true' or 'false' based on its arguments and the state of the object

that contains the guard method. A guard method acts as a guard for another method of the same object called the *guarded method*. When a guarded method is invoked, the guard method will be executed first. If the guard method returns 'true' then the guarded method will be executed. If the guard method returns 'false' then the guarded method invocation will not be executed. The guard method will be executed at a later time when the state of the object changes. Depending on the value returned by the guard method, the guarded method will be executed or its execution will be postponed until the guard method is executed again when the state of the object changes.

The guard method approach, while not eliminating the anomaly, attempts to minimize the amount of rewriting. The approach uses two types of inheritance, one for inheriting synchronization constraints and the other for normal inheritance. It is possible to get confused with the two types of inheritance [113]. In this dissertation only one type of inheritance scheme is considered for both synchronization and normal inheritance.

In the following sections, atomic objects, fully concurrent objects, and synchronization of atomic objects, including truly distributed objects are discussed in detail.

6.2 Atomic Objects

Atomic objects execute only one method at any time. That is, more than one method of the same object will not be in execution simultaneously. If there are multiple invocations on the object at the same time, only one invocation at a time will be processed by the object. There are two possible actions that can take place with the remaining invocations. In one case, all remaining invocations will be rejected so that the senders of those invocations can either attempt to invoke the same method at a

later time or cancel the invocation. In the second case, the object maintains a queue for the remaining invocations and processes them one at a time. The selection can be in a First-In/First-Out (FIFO) order or some other scheduling algorithm can be used.

In case of objects existing in a single process, a single lock per object can be maintained to ensure the atomicity property. There are two ways to use the lock. In the first way, the process trying to invoke a method of the atomic object can lock the object first by getting control over the lock prior to the invocation and releasing the lock after the completion of the invocation. One of the other processes waiting to acquire the lock gets control over the lock next, locks the object, and starts the execution of a method. Different types of locks can be used to control the locking/unlocking mechanism. Some of the mechanisms are: simple mutex variables, semaphores, and monitors. The second way to achieve the atomicity property is to let the object be responsible for managing its own lock. Before executing a message, the object sets the lock and continues with the execution. The object unsets the lock after the completion of the method execution. Any other method invocations that occur during the time the lock is set either wait in the queue meant for the object or get rejected completely for invocation at a later time. Again, different locking/unlocking schemes can be used here.

In the case of distributed objects, it is possible that method invocations can occur at any distributed component. Two conditions must be valid with the distributed components. First, the execution sequence started by a method invocation must be completed before a second invocation starts. This sequence may involve method calls between distributed peers, or recursive method calls. Second, there should be a mechanism to share a distributed lock by the various distributed parts of the object. Any part of the object must first acquire the lock before it can start an execution.

Once an execution starts, all parts of the object must agree that they cannot start another new execution. All parts of the distributed object will be in a blocked state. All of them are unblocked when the execution in progress is completed.

6.2.1 Example: A Simple Atomic Object

An example from the 'Department Stores' problem discussed in Chapter 3 is used to describe a simple atomic object in the ICOM framework. Figure 6.2 shows the interface GeneralTax. This interface defines three methods: stateTax, salesTax, and totalTax. The methods salesTax and stateTax have a single output parameter tax. The implementation of the method totalTax uses the other two methods of the interface to calculate the total tax. The methods 'stateTax' and 'salesTax' can be overridden in a derived class to provide 'inclusion polymorphism' described in Chapter 5. This type of interface can be used to incorporate general tax rules for all 'states' in the United States of America. In the 'Stores' problem, this object can be used to fix an approximate price for an item to be sold at a retail store.

```
// GeneralTax defines two methods that can be overridden
// by state tax objects
interface GeneralTax {

   // returns the sales tax through 'tax'
   salesTax(out double tax);

   // returns the state tax through 'tax'
   stateTax(out double tax);

   // returns the total tax for a state
   totalTax(out double tax);
}
```

Figure 6.2: IDL Description of General Tax Interface with Atomic Methods

When the IDL translator is invoked at a remote node that implements the inter-

face, the set of classes generated contains: GeneralTax, GeneralTaxMetaClass, GeneralTaxMessageClass, and GeneralTaxImplementation. The generation of these classes is similar to the generation used in the example shown in Chapter 3 to describe the dynamic method binding in the ICOM framework.

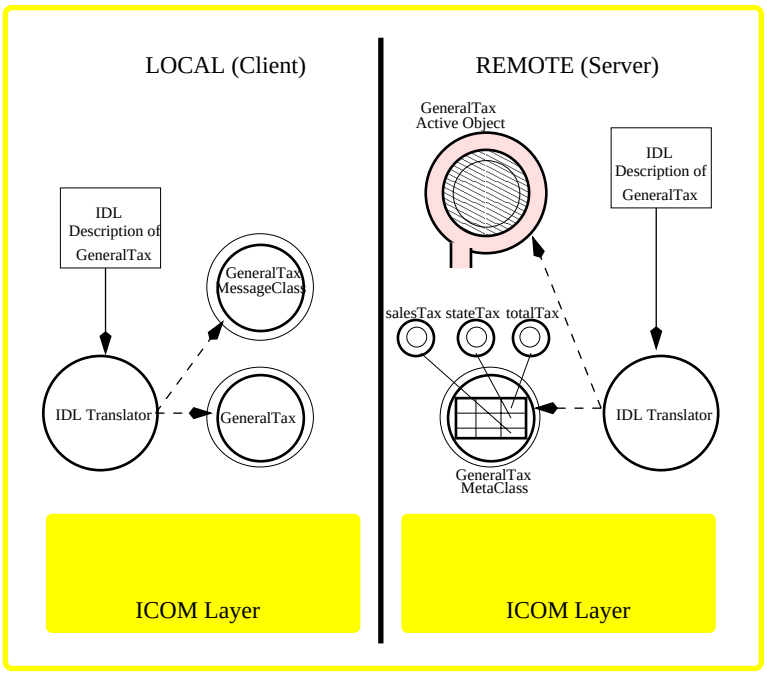

Figure 6.3: IDL Translation of General Tax Interface with Atomic Methods

The GeneralTax class is the class available to the application code that wants to use the GeneralTax interface to create instances that support the interface. A GeneralTax object simply delegates all method invocations on it to the implementation code in GeneralTaxImplementation. The GeneralTaxMetaClass is a metaclass of the class GeneralTaxImplementation. The metaclass' purpose is to intercept any messages intended for instances of GeneralTaxImplementation and process them one at a time.

To create and invoke messages on an object that supports the interface GeneralTax,

a client application first creates an instance of GeneralTax class and invokes the create method on it.

If a local object is to be created, the GeneralTax class creates an instance of GeneralTaxMetaClass. During the creation of the instance of GeneralTaxMetaClass the method objects corresponding to each method (stateTax, salesTax, totalTax, and create) are registered with the *methodTable*, which is a state variable in the GeneralTaxMetaClass instance. Among the registered method objects, the object corresponding to the create method has a method called new that creates an instance of the GeneralTaxImplementation class. At this point the GeneralTax class instance has access to the address of the active object representing the GeneralTaxImplementation class. The other registered method objects have a method called invoke that invokes the corresponding actual method of a GeneralTaxImplementation instance created using the new method of the registered method object of create method.

If a remote object is to be created, the GeneralTax class creates a 'constructor' message object using the GeneralTaxMessageClass class, and hands it to the underlying communication framework. The framework encodes the message and sends it to the site where the object is to be created. At the receiving site, the message will be decoded and handed to an instance of GeneralTaxMetaClass that is automatically created. The instance of GeneralTaxMetaClass creates an instance of GeneralTaxImplementation and it passes the network address of the newly created object to the client application object at the requesting site.

An invocation of the method, totalTax is shown in Figure 6.4. In this figure, it is assumed that two distributed nodes, local and remote exist in a distributed system supported by the ICOM framework. A client application exists at the local node and the implementation of the GeneralTax interface is at the remote node. The client application developer must first run the IDL translator on the GeneralTax interface

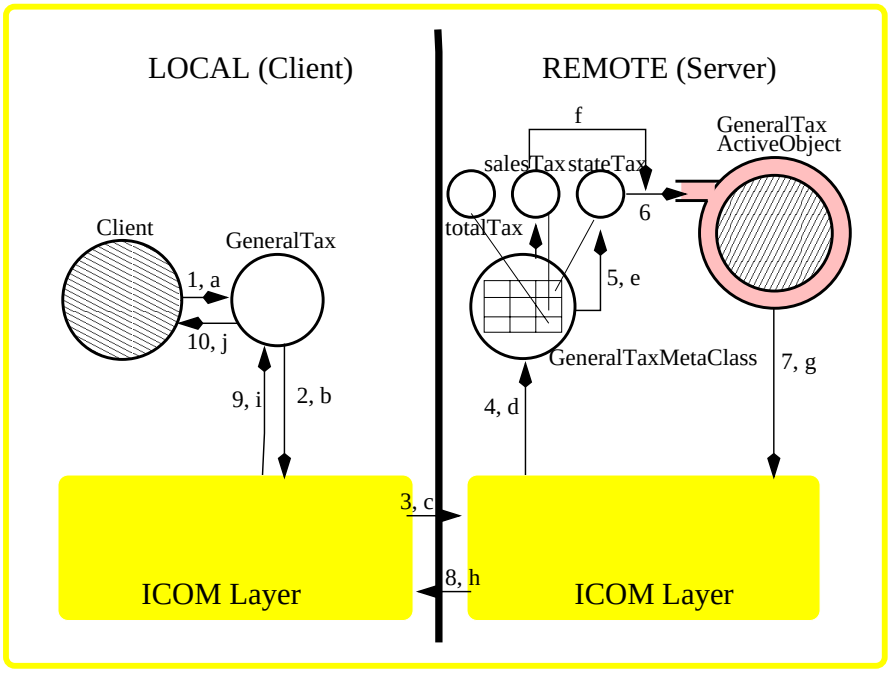

Figure 6.4: Dynamic Method Invocation on an Instance of General Tax Interface with Atomic Methods

at the local node. The application code must be linked with the IDL translated code. The translator generates a proxy class, GeneralTax at the local node. The client application can use this proxy class to create instances of GeneralTax objects at the remote node and invoke methods on it.

To invoke a method on the instance of GeneralTaxImplementation, the application invokes the method of the GeneralTax instance. The GeneralTax instance creates a message and sends it to the active object at the remote node. If multiple invocations are involved, messages get queued at the active object queue. After a method completes its execution in GeneralTaxImplementation, the active object retrieves a pending message from its queue and hands it to the GeneralTaxImplementation instance. Here the active object maintains a lock, and blocks the incoming messages and handles

one message at a time as required by the atomicity property.

An invocation of the stateTax method (arrow labeled 1) and an invocation of the salesTax method (arrow labeled 'a') on the on the GeneralTax proxy instance results in a series of method calls. The sequence can be explained as follows:

At the local node

the GeneralTax proxy object communicates with the GeneralTaxMessageClass and creates two message objects corresponding to the methods, stateTax and salesTax and forwards them to the remote site one after the other (arrows labeled 2, b, 3, and c). The ICOM layer at the local node encodes the message objects before sending them over to the remote node.

At the remote node

the ICOM layer at the remote node receives the messages, partially decodes each message one at a time, to get the method name, class name, and a network address of the active object of the GeneralTax interface. The ICOM layer then forwards the message objects to the instance of GeneralTaxMetaClass (arrow labeled 4,d). The metaclass instance looks up in its method table for a server method object, and passes the message objects to the server method objects stateTax and salesTax respectively. The method objects decode the arguments from the messages, and try to invoke the stateTax and salesTax methods of the active object of GeneralTax (arrows labeled f and 6). The active object accepts one method invocation at a time and sends the results back to the local node using the output parameters (arrows labeled 7,g, 8,h, 9, i, 10, and j).

Thus, the ICOM framework supports atomic objects that are active.

6.2.2 Example: Atomic Objects involving Remote Inheritance

In this section an example demonstrating the use of inheritance in atomic objects is given. Inheritance involving a GeneralTax interface and a VirginiaTax interface from the 'Stores' problem described in Chapter 3 are used to describe the working mechanism of translation and method invocation in the ICOM framework. Figure 6.5 shows two interfaces, GeneralTax and VirginiaTax. The GeneralTax interface defines two methods, salesTax and stateTax and totalTax. All methods have a single output parameter, tax. The VirginiaTax inherits from GeneralTax and overrides the two methods, salesTax and stateTax. The two interfaces are used to calculate the state tax, sales tax, and total tax for the 'state' of Virginia.

```
// GeneralTax defines two methods
interface GeneralTax {
    // returns the sales tax through 'tax'
    salesTax(out double tax);
    // returns the state tax through 'tax'
    stateTax(out double tax);
    // returns the total tax for a state
    totalTax(out double tax);
}

// VirginiaTax type inherits from GeneralTax and
// overrides the two methods
interface VirginiaTax : reuse GeneralTax {
    // returns the sales tax through 'tax'
     salesTax(out double tax);
    // returns the state tax through 'tax'
     stateTax(out double tax);
  }
```

Figure 6.5: IDL Description of Virginia Tax Interface with Atomic Methods involving Inheritance

In this example it is assumed that two nodes, local and remote, exist in the distributed system supported by the ICOM framework. The remote node contains the

implementation of the GeneralTax interface and the local contains the implementation of VirginiaTax interface. For simplicity, it is assumed that the the GeneralTax implementation has already been done at the remote node and the VirginiaTax implementation is to be done at the local node.

An invocation of the distributed IDL translator at the local node on the IDL descriptions shown in Figure 6.5 results in the generation of classes at the local node. The IDL translator realizes the existence of the implementation of the GeneralTax implementation at the remote node. It sends a 'compile' message to the distributed IDL translator at the remote node, by passing the IDL description. Both IDL translators can run in parallel.

The classes for GeneralTax assumed to be available at the remote node are: GeneralTax, GeneralTaxMetaClass, GeneralTaxMessageClass, and GeneralTaxImplementation. Of these classes, the GeneralTaxImplementation class is a skeleton class that must be filled in with method implementations by a developer.

The classes for VirginiaTax generated at the remote node are: VirginiaTax, VirginiaTaxProxyMetaClass, VirginiaTaxMessageClass, and VirginiaTaxProxyImplementation. The VirginiaTax class is a proxy class that inherits from GeneralTax proxy class. An instance of VirginiaTax classes at the remote node contains the network addresses of the peer proxy objects of the VirginiaTax interface at the local node and the remote node. The VirginiaTaxProxyMetaClass inherits from GeneralTaxMetaClass, the VirginiaTaxMessageClass inherits from GeneralTaxMessageClass, and the VirginiaTaxProxyImplementation class inherits from GeneralTaxImplementation. It should be noted that the implementation of all classes related to the VirginiaTax interface at the remote node are automatically generated by the IDL translator.

At the local node, the classes generated for the GeneralTax interface are: GeneralTax, GeneralTaxProxyMetaClass, GeneralTaxMessageClass, and GeneralTaxProxyImple-

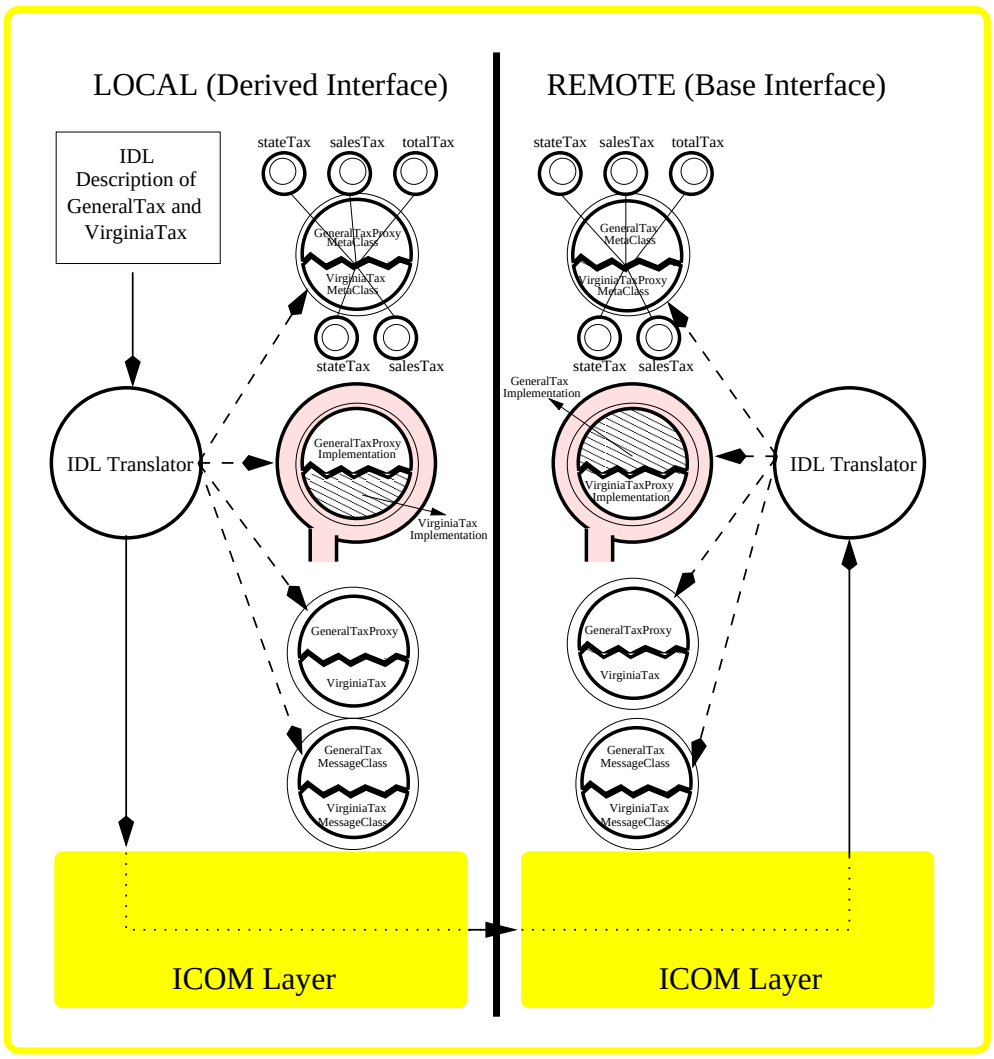

Figure 6.6: IDL Translation of Virginia Tax Interface with Atomic Methods involving Inheritance

mentation. The implementation of all these classes is generated by the IDL translator.

The classes generated for the VirginiaTax interface are: VirginiaTax, VirginiaTaxMetaClass, VirginiaTaxMessageClass, and VirginiaTaxImplementation.

The VirginiaTax class inherits from the GeneralTax class, VirginiaTaxMetaClass inherits from GeneralTaxMetaClass, VirginiaTaxMesageClass inherits from GeneralTaxMessageClass, and finally VirginiaTaxImplementation inherits from GeneralTaxImplementation. It must be noted that the VirginiaTaxImplementation class is a skeleton class and the application developer must implement the methods of this class. The implementation for the remaining classes is generated by the IDL translator.

Consider a truly distributed object from the above example, an instance of the GeneralTaxImplementation, on the remote node, and VirginiaTaxImplementation is on the local node. To guarantee the atomicity property for message processing, a shared lock can be maintained at both nodes by the active objects representing the peer proxies. Any distributed shared locking scheme can be used to maintain the locks. The algorithm used in the ICOM framework is the Suzuki and Kasami algorithm for sharing distributed locks. The algorithm is described in Section 6.2.4.

It is assumed that it is possible to invoke methods on both peers. Consider the method invocations salesTax and totalTax at the remote node and the local node respectively. Initially, the peer at local is the owner because we assume that the application node creates an instance of VirginiaTax at local and invokes the create() method on it. Since VirginiaTax is implemented on local, a proxy for it is created at remote. This proxy is the instance of the class VirginiaTaxProxyImplementation. Similarly, a proxy for GeneralTax class is created at local. This instance in turn creates a corresponding instance of GeneralTaxImplementation on remote by sending a create message to it.

Two invocations of the method stateTax on an instance of VirginiaTaxProxy at the

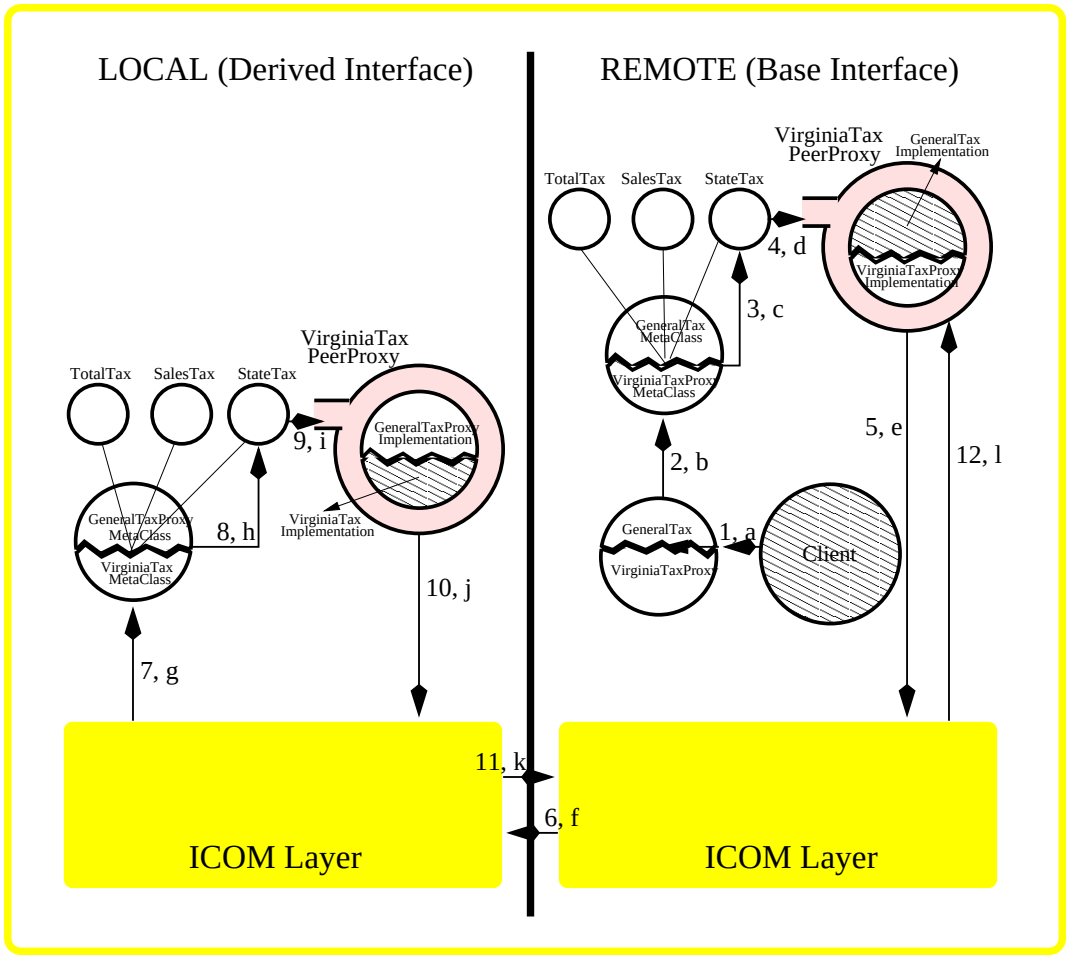

Figure 6.7: Dynamic Method Invocation on an Instance of Virginia Tax Interface involving Inheritance and Atomicity

remote node are shown in Figure 6.7 (arrows labeled 1 and 'a'). The processing of the second invocation waits until the processing of the first invocation is completed. The sequence of method calls are described as follows:

At the remote node

the VirginiaTax instance creates a message object for each of the two method invocations and passes them to the metaclass instance of VirginiaTaxProxyMetaClass (arrows labeled 2 and b). The metaclass instance looks up in its method table for a method object and passes the messages to the method object stateTax (arrows labeled 3 and 'c'). The method object copies the arguments from the message and sends a message to the peer proxy on the remote node (arrow labeled 4). It repeats the process for the second message (arrow labeled d). The active object, VirginiaTax PeerProxy, starts the processing of the first message and blocks the second message. This active object sends an invocation to the peer proxy at the local node (arrows labeled 5 and 6).

At the local node

the ICOM layer receives the encoded message and partially decodes it to get the network address of the peer proxy at the local node, method name and the class name of the message. The ICOM layer then forwards the message to the metaclass instance VirginiaTaxMetaClass (arrow labeled 7). The metaclass instance looks up in the method table for a server method object and passes the message to the server method object stateTax (arrow labeled 8). The method object decodes the arguments and passes the message to the peer proxy, VirginiaTax PeerProxy (arrow labeled 9). The peer proxy executes the method and sends the result through the output parameter tax to the remote node (arrows labeled 10, 11, and 12). The peer proxy sends a *peerUnblock* (discussed in the

next section) message to the peer proxy at the remote node (arrow labeled 13).

At the remote node

the ICOM layer receives the messages and passes them to the peer proxy. The peer proxy after receiving the *peerUnblock* message, resets the lock and allows the next message to be processed.

The same sequence of method calls is repeated for the second message. Thus, the ICOM framework supports concurrency that allows method invocation on any component of a distributed object. At the same time the framework assures atomicity.

6.2.3 Implementation Details

In the ICOM framework, the atomicity property is supported with a simple boolean variable and a message queue for each object. So messages for any part of a distributed object are enqueued if there is already an execution in progress.

To achieve the atomicity property, the distributed actor model described in Chapter 3 is extended with four additional basic messages, namely, newOwner, peerUnblock, ownershipRequest, and ownershipRelease. Any distributed part must first establish with the remaining peers that it is the owner before it can start a new execution. Any non-owner peer with a new message sends a request for ownership using the ownershipRequest message to all peers. The owner peer, after completing the execution of messages intended for it, relinquishes the control of the ownership by sending an ownershipRelease message to a requesting peer. The peer, upon receiving an ownershipRelease message, declares itself as the new owner by sending a newOwner message to all peers. All non-owner peers acknowledge by sending the same newOwner message back to the owner. The owner after receiving acknowledgments from all peers can start a new execution.

Potentially, any distributed shared lock algorithm can be used for implementing a shared lock in a distributed environment. In this dissertation Suzuki and Kasami algorithm [167] is used for implementing atomic locks. A brief description of the algorithm is given below.

6.2.4 Suzuki and Kasami Algorithm

The Suzuki and Kasami algorithm was originally designed to provide distributed mutual exclusion to critical sections among N nodes in a distributed network of computers. The computers (nodes in the network) are assumed not to share memory. They communicate with one another only through message passing. The mutual exclusion requirement satisfied in the algorithm is that at most one node is in its critical section at any time.

The algorithm provides a data structure called 'Token' that is transferred among the nodes. Any node that receives the 'Token' can enter its critical section repeatedly until it sends the 'Token' to some other node. A node requesting the 'Token' sends a 'Request' message to all other nodes. A 'Request' message from a node j has the form Request(j,s) where j is the identifier of the requesting node and s is the 'sequence number' which indicates that the node is requesting its (N+1)st critical section invocation. Each node maintains an array, RN, of size N to record the largest sequence number ever received from each of the other nodes.

Initially node 1 has the 'Token'. If Request(j,s) is received by node i, then node i updates its RN array by RN[i] = max(RN[j],s), where max returns the maximum of the two arguments. The 'Token' is a message that has the form token(Queue, LN) where Queue is the queue of requesting nodes and LN is an array of size n such that LN[j] is the sequence number of the request of that node j most recently made.

When a node i completes executing its critical section, the array LN contained in the last message with 'Token' received by i, is updated by LN[i] = RN[j] to indicate that current request has been granted. Next, all node identifiers j, such that RN[i] = LN[j]+1 (i.e., node j is requesting for the 'Token'), is appended to Queue provided that j is not already in Queue. When these updates are complete and if Queue is empty then node i retains the 'Token' until a requesting node is found through the arrival of a Request message.

This algorithm is deadlock free and starvation free. Critical sections are granted in the First Come First Served manner. It requires, at most, N message exchanges per mutual exclusion invocation; (N-1) 'Request' messages and one 'Token' message or no message at all if the node having the 'Token' happens to be the node requesting.

6.2.5 Shared Lock Abstraction

The abstraction of the shared lock of truly distributed objects in terms of the Suzuki and Kasami algorithm is realized as follows. The nodes in the algorithm are mapped to peer components of a distributed object. A 'Token' is treated as the ownership of a peer component. Any peer component with the ownership can potentially start the execution of an external method invocation. A 'Request' message is mapped to the OwnershipRequestMessage. The 'Token' is mapped to the message OwnershipRelease that contains the addresses of peer components requesting ownership. The OwnershipRelease message also contains a second array of size N which holds the information about whether a particular peer is requesting ownership because of a guard method or not. This is required to avoid the transfer of the OwnershipRelease message among the peer components when all of the peer components are waiting on guard methods. Transfer of ownership is ensured by the message NewOwner. The NewOwner

message is used to inform other nodes about the new owner and also to act as an acknowledgment between the owner peer component and other peer components.

6.3 Synchronized Objects

It is often the case that real world objects exhibit different behaviors in different situations. A classic example from the literature is that of a 'bounded buffer'. A bounded buffer cannot accept a get method when the buffer is empty. The only method accepted in this state is the put method. Similarly, the buffer cannot accept a put method when the buffer is full. It accepts only a get method at that time. But, when the buffer is not full and is not empty it accepts both put and get messages.

In the ICOM framework, guard methods are used as a way of providing synchronization to objects. The ICOM model views guard methods as regular methods that have one boolean output parameter and any number of input parameters. By definition, they do not affect the state of an object. The restriction on the input parameters is that they must correspond to the input parameters of the method guarded by the guard method. Guard methods always evaluate to either true or false. The key word *guardedBy* is used to represent a guard method that guards another method in an interface. Consider an instance of an object supporting an interface with a method guarded by another method. When this object receives an invocation of the guarded method, the guard method will be evaluated first. If the guard method returns true via its output parameter, then the guarded method is executed. Otherwise, the original message will enter the queue and the guard method will be evaluated again at a later point of time. The evaluation takes place when the state of the object is modified by another method of the same object.

6.3.1 Example: An Atomic Object with a Guard Method

The 'Department Stores' problem described in Chapter 3 is used to describe guard methods in the ICOM framework. Figure 6.8 shows the Manufacturer interface that contains a method startShipment with four input parameters, brandName, itemNumber, itemCount, and storeNumber. This method is used by a client application requiring a product to be shipped to a store. The client application and the Manufacturer implementation can exist in a distributed environment. The method startShipment is guarded by the method suppliesAvailable. The relationship between a method and its guard is represented by the keyword *guardedBy*. The guard method, suppliesAvailable takes three input parameters, brandName, itemNumber, and itemCount, and an output parameter, okay. The manufacturer cannot start shipment unless the product is available. The method suppliesAvailable will be evaluated first when an invocation on the method startShipment is received by the active object representing Manufacturer interface. The method startShipment will be executed only when there are supplies available. The availability is ensured by the guard method suppliesAvailable

```
interface Manufacturer {

   // this method is guarded by suppliesAvailable method
   startShipment(in string brandName,
                 in long  itemNumber,
                 in long  itemCount,
                 in long storeNumber) guardedBy

      suppliesAvailable(in string brandName,
                        in long  itemNumber,
                        in long  itemCount,
                        out boolean okay);

}
```

Figure 6.8: IDL Description of Manufacturer Interface with a Guard Method

An IDL translation of the Manufacturer interface is shown in Figure 6.9. It is assumed that the implementation of the interface is to be done at a remote node and the client application at a local node. IDL translators can run independently at both nodes, because there is no remote inheritance involved in this example.

At the remote node, the IDL translator generates four classes: ManufacturerMetaClass, ManufacturerImplementation, ManufacturerMessageClass, and Manufacturer proxy class. The metaclass ManufacturerMetaClass contains a method table representing the two methods, startShipment and suppliesAvailable. Two server method objects are attached to the metaclass and they can decode the method arguments of the messages represented by them. The ManufacturerImplementation class is a skeleton class and its implementation must be written in a native language by a developer. The ManufacturerMessageClass is used to generate message objects for the two methods of the interface. The Manufacturer class is a proxy class for the Manufacturer interface. It is mainly used in the client applications, but the ManufacturerImplementation may use it to make invocations on other Manufacturer active objects.

The implementation of the server method objects that represent guarded methods is different from other server method objects. When an invocation of the guarded method is to be made on the active object representing the Manufacturer interface, the guard method will be invoked first. If the guard method returns true, then the guarded method will be executed. Otherwise, the message enters the queue of the active object for later reevaluation of the guard method.

At the local node, where the client application is to be implemented, the IDL translator generates two classes: Manufacturer proxy class and ManufacturerMessageClass. The proxy class is used in the client application to create and invoke messages on the active object at the remote node. It uses the ManufacturerMessageClass to generate message objects that can be sent to the remote node. The client application

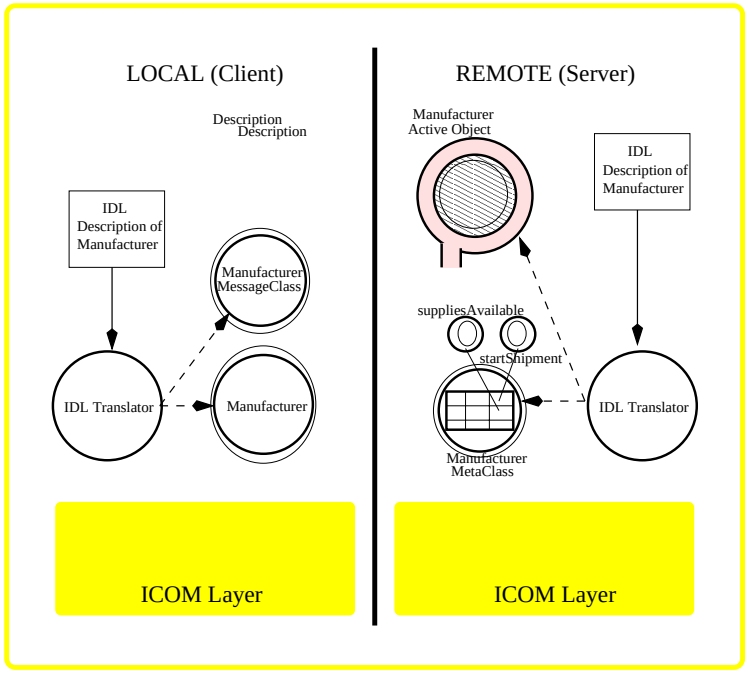

Figure 6.9: IDL Translation of Manufacturer Interface with a Guard Method

can be a financial agency that maintains an information processing service center that represents a chain of stores. This center can request for the shipment of a product whose inventory is low at a clothes store. This is done by the invocation of the startShipment method on the manufacturer object.

Figure 6.10 shows a method invocation of startShipment method, on an active object representing the Manufacturer interface (arrow labeled 1). The sequence of method calls can be explained as follows:

At the local node

The instance of Manufacturer proxy class generates a message object using the instance of ManufacturerMessageClass and sends it to the remote node (arrows labeled 2 and 3).

At the remote node

the ICOM layer at the remote node receives the message and partially decodes it to get the network address of the Manufacturer active object, the method name and class name. It then passes the message object to the instance of ManufacturerMetaClass (arrow labeled 4). The metaclass instance looks up in the method table for a server method object, and passes the message object to the server method object (arrow labeled 5), startShipment. The server method object decodes the arguments and realizes that this method is guarded by the method suppliesAvailable. The server method object first tries to invoke the guard method on the active object (arrow labeled 6). If the guard method returns true via its output parameter, then it invokes the startShipment method of the active object (arrow labeled 7). If the guard method returns false via its output parameter, then the message enters the queue and the guard method will be re-evaluated when the state of the active object changes.

Thus, in the ICOM framework, objects can be synchronized using guard methods.

6.3.2 Synchronized Objects involving Inheritance

In this subsection, a simple example of a method invocation on a truly distributed object with a guarded method is described. The 'Stores' problem is used to define the interface shown in Figure 6.11. There are two interfaces, ClothesStore and LocalClothesStore. The second interface inherits from the first interface. The ClothesStore interface defines a method putItemOnSale, which is guarded by another method isInventoryOkay. The method putItemOnSale takes an input parameter, brandName, and an output parameter, salePercentage. The method isInventoryOkay takes one input parameter, brandName and one output parameter, okay.

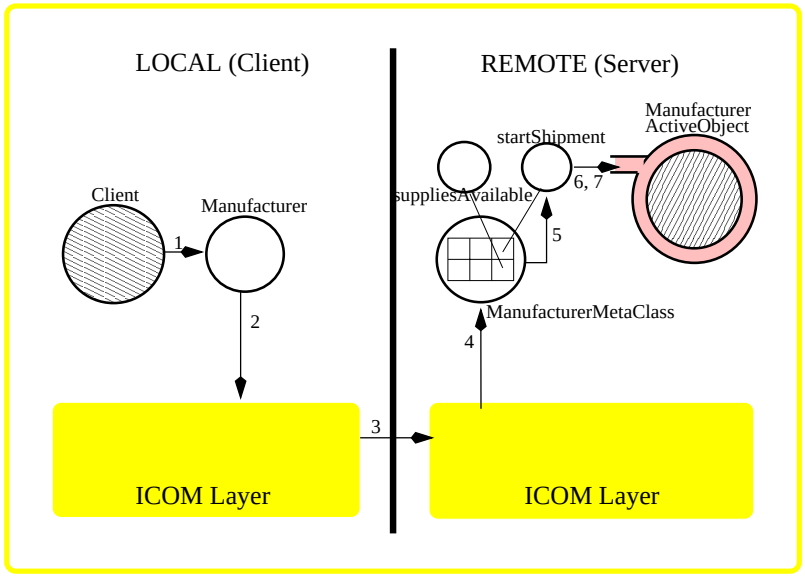

Figure 6.10: Dynamic Method Invocation on an Instance of Manufacturer Interface with a Guard Method

In this example, it can be assumed that the ClothesStore interface is implemented at the CIPS center and the LocalClothesStore is implemented at a distributed location. An application using the instance of ClothesStore can spawn a thread and send a request to the CIPS center to put an item on sale if the inventory of the item remains high over a long period of time. The CIPS center can make a decision based on the inventory status and the time of the year, and decide whether to put the item on sale or postpone the decision.

In this example it is assumed that a client application and the implementation of LocalClothesStore are at a local node, and the implementation of the ClothesStore interface is at a remote node. For simplicity it is assumed that the implementation of the ClothesStore has already been done. When an IDL translator is invoked on the IDL description shown in Figure 6.11, it realizes the existence of remote inheritance

```
interface ClothesStore {
   putItemOnSale(in string brandName,
                 out long salePercentage) guardedBy

      isInventoryOkay(in string brandName,
                      out boolean okay);,
   ⋮
}

interface LocalClothesStore : reuse ClothesStore {
   updateSales(in long itemNumber,
               in long itemCount,
               in double pricePerItem);
   ⋮
}
```

Figure 6.11: IDL Description of Local Clothes Store Distributed Object with a Guard Method

and issues a 'compile' message to the IDL translator at the remote node.

The local IDL translator generates four classes corresponding to the ClothesStore interface. They are: ClothesStore, ClothesStoreProxyMetaClass, ClothesStoreProxyImplemetation, and ClothesStoreMessageClass. All these classes represent the corresponding classes at the remote node. The implementation of all these classes is generated by the IDL translator. The classes generated for the LocalClothesStore interface are: LocalClothesStore LocalClothesStoreMetaClass LocalClothesStoreImplemetation, and LocalClothesStoreMessageClass. The LocalClothesStore class inherits from ClothesStore class. The LocalClothesStoreMetaClass inherits from ClothesStoreProxyMetaClass. The LocalClothesStoreImplemetation class inherits from ClothesStoreProxyImplemetation class. It should be noted that the implementation generated for the class LocalClothesStoreImplemetation is a skeleton implementation and the actual coding for the methods must be done by the application developer. The LocalClothesStoreMessageClass inherits from the class ClothesStoreMessageClass.

At the remote node, the IDL translator generates four classes corresponding to the LocalClothesStore interface. They are: LocalClothesStore, LocalClothesStoreProxyMetaClass, LocalClothesStoreProxyImplemetation, and LocalClothesStoreMessageClass. The LocalClothesStore classes inherit from the corresponding classes of the ClothesStore interface that were already generated when ClothesStore is implemented earlier.

In this example the stores and central information processing service are geographically distributed. A local store client application can spawn a thread and invoke the method putItemOnSale of the store object. The thread blocks and waits for a response in the salePercentage output parameter. The invocation (arrow labeled 1) is shown in Figure 6.13. The sequence of method calls is explained as follows:

At the local node

the instance of LocalClothesStore proxy class generates a message object for the method putItemOnSale and passes it to the metaclass instance of LocalClotheStoreMetaClass (arrow labeled 2). The metaclass instance looks up in the method table for a server method object and passes the message object to the server method object of putItemOnSale (arrow labeled 3). The server method object copies the arguments and invokes the method on the active object implementation of LocalClothesStoreImplementation. Since the actual implementation of the method exists at the remote node, the active object sends an invocation to the remote node (arrows labeled 5 and 6). The encoded message is sent to the ICOM layer at the remote node.

At the remote node

the ICOM layer at the remote node receives the message and partially decodes it to get the network address of the active object of ClothesStoreImplementation, method name and class name of the message object. The ICOM layer

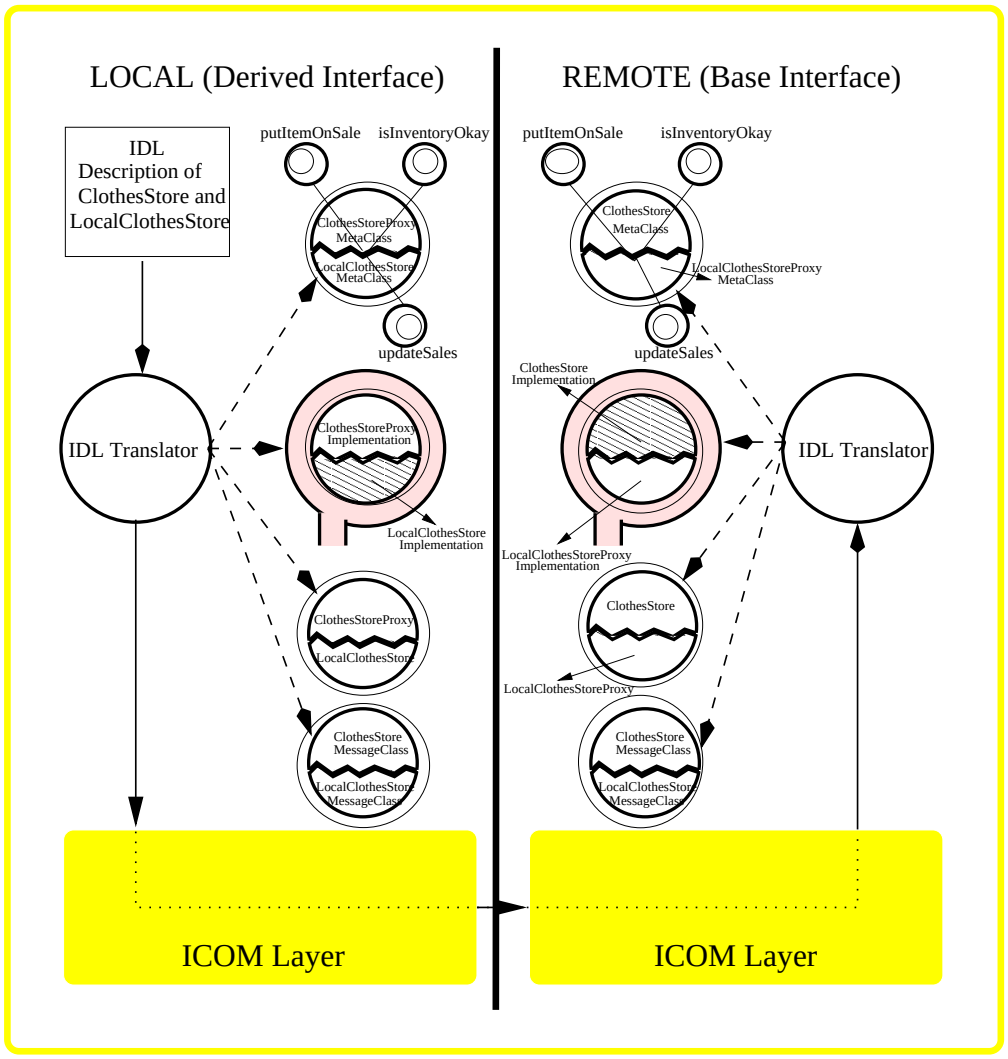

Figure 6.12: IDL Translation of the Local Clothes Store Distributed Object with a Guard Method

then passes the message object to the ClothesStoreProxyMetaClass instance (arrow labeled 7). The metaclass instance does a look up in its method table for a server method object and passes the message to the server method object, putItemOnSale. The method object decodes the message arguments. Before invoking the method object realizes the existence of the guard method, isInventoryOkay, and invokes it first (arrow labeled 9). If it evaluates to true, then the method object invokes the putItemOnSale method on the active object (arrow labeled 10). The result is sent to the local node with the output parameter, salePercentage (arrows labeled 11,12 and 13).

Thus, in the ICOM framework, guard methods can be used for synchronizing method invocations on object. The guard methods can be inherited and overridden just like any other methods.

6.3.3 General Limitations

The current implementation framework of ICOM supports atomic object only. It does not allow a full set or subset of methods in an interface to be concurrent. To provide the flexibility of having some methods as atomic and some methods as synchronized methods, method level locks can be used. The subset of methods requiring atomicity among themselves can have a lock associated with their group. The inheritance of a class with some of its methods concurrent needs to be investigated further.

6.4 Summary

In this chapter, concurrency issues involved with objects in a distributed environment are discussed in detail. Implementation details of atomic objects and synchronized

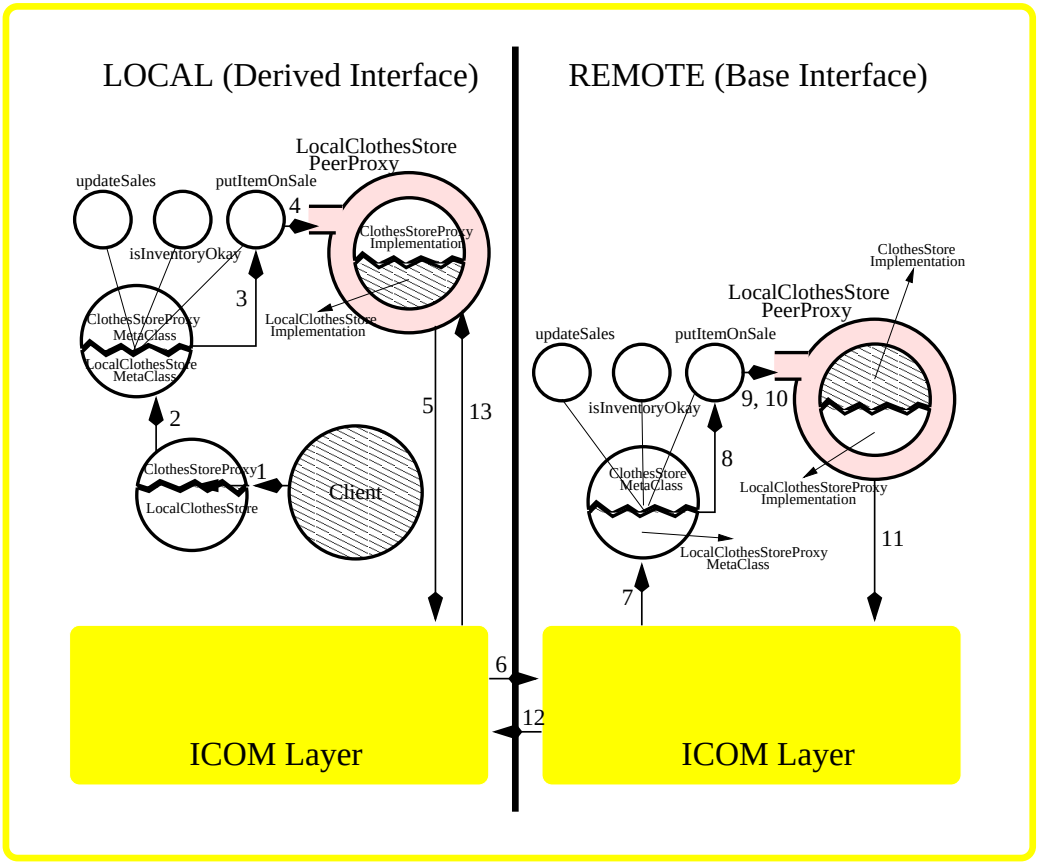

Figure 6.13: Dynamic Method Invocation on the Local Clothes Store Distributed Object with a Guard Method

objects are given with respect to the implementation framework of the dissertation. General limitations of the approach followed are explained.

Chapter 7

Summary and Future Directions

This dissertation addresses the issues involved in developing a powerful common object model that supports advanced features of polymorphism and concurrency. These advanced features are not supported in most of the existing common object models that are geared towards building applications in distributed and heterogeneous environments.

This dissertation focuses on the development of the model with the design criterion that existing languages, compilers, and virtual machines should not be changed. The dissertation also focused on building the model by narrowing the implementation languages to statically typed object-oriented languages. The specific languages chosen for implementation were C++ and Modula-3.

The actor model is used as a basic communication model for exchanging messages. The Distributed Act++ framework is extended to support a common object model in C++ and Modula-3. Reflection-like techniques are used to implement the framework of ICOM. This framework has a uniform design in all implementation languages. A prototype application is developed to demonstrate the feasibility of the defined model.

The features of the languages used for implementing the ICOM framework are:

object, class, single inheritance, method overloading (or a support for run time type checking), concurrency (or support for concurrency via libraries). Most of the statically typed object-oriented languages, including C++, Modula-3, Eiffel, Ada 95, BETA, and Java, support these features. So, it is possible to map the ICOM framework into these languages. The languages Modula-3 and BETA can use one-way interoperability for the method overloading feature, because they do not support this feature in their object models. The one-way interoperability allows these languages to use the method overloading feature in software components written in other languages.

7.1 Contributions

The main contributions of this dissertation are as follows:

Powerful Common Object Model

A powerful common object model's (ICOM) features are defined. The advanced features of polymorphism and concurrency are included in the ICOM model. A uniform implementation framework (ICOM framework) is developed and is implemented in C++ and Modula-3. The feasibility of the model is demonstrated with prototype implementations.

Concurrency

Specific aspects of concurrency considered are atomicity and guard methods. Adding the features of concurrency that support atomic objects and guard methods to a common object model is demonstrated through prototype implementations. Atomicity in truly distributed objects also is guaranteed in the ICOM framework.

Polymorphism

Specific aspects of polymorphism considered are inheritance of abstract methods across address spaces, method overloading, and parameterized types. In this dissertation, adding the features of polymorphism, parameterized types and method overloading, to a common object model is demonstrated. The scheme used for the parameterized types is 'constrained genericity'.

Distributed Compilation

A new concept of 'distributed compilation' is introduced. Distributed compilation allows 'truly distributed objects' to be incorporated into a common object model. Truly distributed objects are the distributed objects whose components exist in different address spaces and are related by 'subclass' and 'superclass' relationships. The translation scheme from an IDL description to a native programming language is explained with several examples. These examples use the advanced features of the ICOM object model.

Distributed Object Model using Actor Model

Implementation of the ICOM object model using the actor model is explained. The implementation framework of the Distributed Act++ system is used as a basis for implementing the ICOM object model framework.

Reflection Techniques

The effectiveness of reflection techniques in implementing a common object model is shown by implementing the ICOM framework in multiple languages.

Distributed Act++

The basic actor message set used in the Distributed Act++ system is extended with four new messages to implement the ICOM framework in multiple languages.

A minor contribution of this dissertation is the taxonomy of existing interoperable systems. There have not been any taxonomies defined in the literature to categorize the existing interoperable systems.

7.2 Future Directions

In the current ICOM object model, inheriting a base class' methods and overriding them with the same number of parameters is allowed. But, covariance or contravariance is not allowed in the method arguments while overriding. The reason for this is that the statically typed object-oriented languages follow different semantics while considering co/contravariance of method arguments. The Simula family of languages support invariance of method arguments while overriding whereas the Eiffel family of languages support covariance of method arguments [1]. It will be interesting to see if there is a direct mapping of interface hierarchy of the ICOM model that uses covariant methods into a language that uses contravariant methods and vice versa.

One of the object-oriented features not considered in this dissertation is 'persistence'. This feature is not selected because it is not present in most of the statically typed object-oriented languages considered. But, this feature is proven to be very useful in applications such as database applications. This feature can be added to a common object model so that other applications can use the model for interoperability.

Some of the aspects of concurrency that would be of interest are to provide intra-object concurrency. In the ICOM framework only atomic object are considered. When intra-object concurrency is allowed, several implementation issues surface. Design must be carefully chosen to allow both synchronized and unsynchronized methods in an object. The ICOM framework seems to be a promising basic on which these extension can be made. For example, object-level locks used for locking objects for the atomicity feature can be extended to methods or groups of methods. The actor model used as the underlying communication model allows multiple executions in one actor which helps to add intra-object concurrency to the ICOM object model.

Non object-oriented languages and dynamically typed languages are not used to implement the ICOM framework. It will be interesting to see how these languages play a role in the ICOM framework.

The major goal of elevating common object models to language object models seems to be an excellent way of providing powerful features and higher levels of abstraction to distributed object systems. Providing mappings to non object-oriented languages and dynamically typed object-oriented languages may show some interesting aspects of the ICOM framework.

Appendix A

The ICOM Object Model

This appendix describes the essential features of the object model used in the ICOM framework. The syntax of the model is given by an Interface Definition Language (IDL) and the semantics are given by natural language descriptions and examples. The syntax of the IDL used in this model is an extension of the CORBA IDL [60]. Only the extensions made to the CORBA IDL and its object model are described here.

The ICOM framework differs from the other distributed object systems in the way messages are communicated and the semantics of method invocations. Specific aspects of the ICOM framework are as follows:

Autonomous Objects

All objects that have their interfaces described in IDL are autonomous. They have their own thread of control for processing method invocations.

Asynchronous invocations

The method invocations in the ICOM framework are asynchronous. A client object can continue after making an invocation, without waiting for a result.

Future variables

To support synchronous invocation semantics, the ICOM framework provides future variables. Any output parameter in an operation of an interface will be mapped to a future variable. A future variable blocks an invoking process that tries to access its value when the value is not yet set. The future variable unblocks the invoking process when the variable value is set.

No return types

There are no return types for operations of the interfaces in the ICOM framework. Any expected result must be represented as an output parameter.

Message queues

Every autonomous object has a message queue associated with it. If an autonomous object is processing a message when a new message arrives, the new message will be entered into the message queue. After the processing of the previous method invocations is complete, the object removes the message from the queue and processes it.

Atomicity

In the ICOM framework, atomicity is guaranteed for all method invocations on autonomous objects. A method invocation is processed completely before another method invocation can start.

The following sections describe the unique features of the ICOM model that distinguish it from other common object models: implementation inheritance, overloaded methods, parameterized types, and guard methods. For each feature, the IDL syntax, semantics, and examples are provided.

A.1 Implementation inheritance

Figure A.1 illustrates the syntax of an interface that supports both interface and implementation inheritance. An interface consists of an interface header and an interface body. They are described below.

⟨interface⟩ ::= ⟨interface_dcl⟩
⟨interface_dcl⟩ ::= ⟨interface_header⟩ ''{'' ⟨interface_body⟩ ''}''
⟨interface_header⟩ ::= ''interface'' ⟨identifier⟩ [⟨inheritance_spec⟩]
⟨inheritance_spec⟩ ::= '':'' [⟨**reuse**⟩] ⟨scoped_name⟩
⟨scoped_name⟩ ::= ⟨identifier⟩ | ''::'' ⟨identifier⟩ | ⟨scoped_name⟩ ''::'' ⟨identifier⟩
⟨interface_body⟩ ::= ⟨op_dcl⟩*
⟨op_dcl⟩ ::= ⟨identifier⟩ ⟨parameter_dcls⟩
⟨parameter_dcls⟩ ::= ''('' ⟨param_dcl⟩ '','' ⟨param_dcl⟩* '')'' | ''('' '')''
⟨param_dcl⟩ ::= ⟨param_attribute⟩ ⟨param_type_spec⟩⟨simple_declarator⟩
⟨param_attribute⟩ ::= ''in'' | ''out''
⟨param_type_spec⟩ ::= ⟨base_type_spec⟩ | ⟨string_type⟩ | ⟨scoped_name⟩
⟨simple_declarator⟩ ::= ⟨identifier⟩

Figure A.1: Interface and Implementation Inheritance

A.1.0.1 Interface Header

The interface header consists of an interface_header and an interface_body. The interface_header contains two elements, an interface name, and an optional inheritance specification. The interface name must be preceded by the keyword interface, and consists of an identifier that names the interface. The ⟨scoped_name⟩ in an ⟨inheritance_spec⟩ must denote an interface. The ICOM model supports single inheritance for interfaces. The keyword reuse can be used when implementation inheritance is desired.

Inheritance An interface can be derived from another interface, which is then called a base interface of the derived interface. A derived interface, like all interfaces, may declare new elements (constants, types, attributes, and operations). In addition, unless redefined in the derived interface, the elements of a base interface can be referred to as if they were elements of the derived interface. The name resolution operator ("::" may be used to refer to a base element explicitly; this permits reference to a name that has been redefined in the derived interface.

A.1.0.2 Interface Body

The interface body contains operation declarations. These declarations specify the operations that the interface exports and the format of each, including the operation name, and the types of all parameters of an operations.

Operation Declaration An operation declaration consists of an identifier that names the operation in the scope of the interface in which it is defined, a parameter list that specifies zero or more parameter declarations for the operation, and an optional keyword ⟨guardedBy⟩ followed by another method declaration. The parameters of the second method are restricted to be among the parameters of the primary method.

Parameter Declarations A parameter declaration must have a directional attribute that informs the communication service in both the client and the server of the direction in which the parameter is to be passed. The directional attributes are in, the parameter is passed from client to server, and out, the parameter is passed from server to client.

A.1.1 Semantics

The ICOM framework supports both the interface inheritance and the implementation interitance. The interface inheritance forces the implementor of a derived interface to provide implementation for the base interface too.

Implementation inheritance in some languages is known as *class inheritance*. In the ICOM framework the implementation inheritance is realized by using the keyword 'reuse'. An implementor of a derived class can inherit the implementation of its base class and implement only the extended or overridden functionality in the derived class.

The framework of Interoperable Common Object Model [28] realizes a truly distributed object by supporting abstract methods across processes. The parts of a distributed object can exist in more than one process without losing the object's integrity. The base class of the object can exist in one process on a remote machine and the derived class of the same object can exist in another process on a local machine. When overridden abstract methods are invoked in the parent, the redefined methods of the child are executed.

The ICOM framework provides a distributed shared lock mechanism for supporting the atomicity of method invocations even in distributed objects that use implementation inheritance.

A.1.2 Example: An interface with implementation inheritance

Figure A.2 shows an example of a truly distributed object with the GeneralTax interface and VirginiaTax interface that inherits from the GeneralTax interface. The VirginiaTax interface uses the keyword 'reuse' to represent implementation inheritance.

In this example it is assumed that the GeneralTax interface is already implemented at a remote node and the VirginiaTax interface is to be implemented at a local node. An IDL translator when invoked on the VirginiaTax interface at the local node is expected to invoke the IDL translator on the remote node by passing the VirginiaTax interface details, resulting in distributed compilation. Peer-proxies are expected to be generated by the IDL translators on both nodes. The VirginiaTax peer proxy at the remote node inherits from the GeneralTax implementation and the VirginiaTax implementation inherits from the GeneralTax peer proxy at the local node. This type of distributed compilation can be extended to an arbitrary number of nodes depending on how many interfaces are implemented that are related by inheritance.

In this example, when the method totalTax is invoked at the remote node, the overridden methods of salesTax and stateTax implemented at the local node are executed.

```
// GeneralTax defines two methods
interface GeneralTax {
   // returns the sales tax through 'tax'
   salesTax(out double tax);
   // returns the state tax through 'tax'
   stateTax(out double tax);
   // returns the total tax for a state, uses salesTax() and stateTax()
   totalTax(out double tax);
}

// VirginiaTax type inherits from GeneralTax and
// overrides the two virtual methods
interface VirginiaTax : reuse GeneralTax {
   // returns the sales tax through 'tax'
   salesTax(out double tax);
   // returns the state tax through 'tax'
   stateTax(out double tax);
  }
```

Figure A.2: IDL Description of Virginia Tax Distributed Object Interface

A.2 Method Overloading

A.2.1 Syntax

The syntax of interfaces that support method overloading is similar to the syntax of the interfaces described in the previous section. There are two ways of realizing method overloading in interfaces. First, a single interface can have multiple methods with the same name but different signatures. Second, a derived interface can have a method with the name matching another method in its base class.

A.2.2 Semantics

Method overloading allows an interface to have more than one method with the same name but different signatures. When an overloaded method is invoked, the ICOM framework can identify the correct method and bind the invocation by by verifying the signatures of the methods.

The ICOM framework supports one-way interoperability in languages that do not support method overloading such as Modula-3 and BETA. One-way interoperability allows applications in these languages to invoke overloaded methods in software components of other languages that support method overloading.

A.2.3 Examples: Interfaces with overloaded methods

Figure A.3 and Figure A.4 show how method overloading can be used in the ICOM framework. Figure A.3 shows an example that uses two methods with the same name, priceOfItem, but different signatures in the same interface. One of the methods defines an item by a numeric value, which represents the item number and the other method defines an item by a character string, which represents the item label. When this

overloaded method is invoked, the ICOM framework can distinguish between the two methods by verifying the signatures of the methods.

Figure A.4 shows an example that uses two methods with the same but different signatures in two different interfaces related by inheritance. This example shows the overloaded method discussed in the previous example, but uses inheritance to overload the method, priceOfItem. The ICOM framework can distinguish between different invocations of this method by verifying the signature.

```
interface ClothesStore {

    // returns price of an item through 'price'
    priceOfItem(in long itemNumber,
                out double price);
    // returns price of an item through 'price'
    priceOfItem(in string itemLabel,
                out double price);
    // returns the sale percentage through 'salePercentage'
    putItemOnSale(in long itemNumber,
                  out double salePercentage);
}
```

Figure A.3: IDL Description of Clothes Store Interface Containing Overloaded Methods

A.3 Generic Interfaces

The generic types supported in the ICOM model are 'constrained generic types'. These generic types are similar to C++ templates(unconstrained generic types), but are not as powerful as them. The generic types of the ICOM object model can have one or more type parameters. But, the interfaces of the parameters must be defined before using them as the type parameters. Their syntax and semantics are described below.

```
interface ClothesBase {
    // returns price of an item through 'price'
    priceOfItem(in long itemNumber,
                out double price);
    .
    .
    .
}

interface ClothesDerived : reuse ClothesBase {
    // returns price of an item through 'price'
    priceOfItem(in string itemLabel,
                out double price);
    .
    .
    .
}
```

Figure A.4: IDL Description of Clothes Store Distributed Object Containing Overloaded Methods

A.3.1 Syntax

The syntax of generic types is given in figure A.5. The syntax of an interface described earlier is extended by the addition of the keyword 'generic_interface'.

A generic interface contains three elements: a generic interface name, a parameter list, and an optional inheritance specification. The generic interface name must be preceded by the keyword generic_interface, and consists of an identifier that names the interface. The interface name must be followed by the symbol "⟨", a list of identifiers representing interfaces, and the symbol "⟩". There can be more than one identifier in the id_list. Each identifier in the id_list must denote a previously defined interface. Generic interfaces also support interface and implementation inheritance.

A.3.2 Semantics

The generic types in the ICOM framework are restricted to constrained genericity where the interfaces of the parameter types must be defined prior to the generic

```
⟨interface⟩ ::= ⟨interface_dcl⟩
  ⟨interface_dcl⟩ ::= ⟨interface_header⟩ ‘‘{’’ ⟨interface_body⟩ ‘‘}’’
    ⟨interface_header⟩ ::= ‘‘interface’’ ⟨identifier⟩ [⟨inheritance_spec⟩] |
                           ‘‘generic_interface’’ ⟨identifier⟩ ‘‘⟨’’ ⟨id_list⟩ ‘‘⟩’’[⟨inheritance_spec⟩]
      ⟨inheritance_spec⟩ ::= ‘‘:’’ [⟨reuse⟩] ⟨scoped_name⟩
        ⟨scoped_name⟩ ::= ⟨identifier⟩ | ‘‘::’’ ⟨identifier⟩ | ⟨scoped_name⟩ ‘‘::’’ ⟨identifier⟩
      ⟨id_list⟩ ::= ⟨identifier⟩ [ ‘‘,’’ ⟨identifier⟩]*
    ⟨interface_body⟩ ::= ⟨export⟩*
      ⟨export⟩ ::= ⟨type_dcl⟩ ‘‘;’’ | ⟨const_dcl⟩ ‘‘;’’ |
                   ⟨attr_dcl⟩ ‘‘;’’ | ⟨op_dcl⟩ ‘‘;’’
        ⟨op_dcl⟩ ::= ⟨identifier⟩ ⟨parameter_dcls⟩
          ⟨parameter_dcls⟩ ::= ‘‘(’’ ⟨param_dcl⟩ ‘‘,’’ ⟨param_dcl⟩* ‘‘)’’ | ‘‘(’’ ‘‘)’’
            ⟨param_dcl⟩ ::= ⟨param_attribute⟩ ⟨param_type_spec⟩⟨simple_declarator⟩
              ⟨param_attribute⟩ ::= ‘‘in’’ | ‘‘out’’
              ⟨param_type_spec⟩ ::= ⟨base_type_spec⟩ | ⟨string_type⟩ | ⟨scoped_name⟩
              ⟨simple_declarator⟩ ::= ⟨identifier⟩
```

Figure A.5: Parameterized Types

interface that uses the parameters. An interface that is used to instantiate the generic interface can be implemented on a remote node different from the node that has the implementation of the generic interface.

A.3.3 Examples: Generic interfaces

Figure A.6 shows a simple generic interface. In this example two distributed nodes are assumed, local and remote. At the remote node, an implementation of the StoreTemplate is provided. Since the parameterized types are constrained, the types of the parameters must be known while defining the parameterized type, StoreTemplate. The StoreType interface must be defined before defining the StoreTemplate. So, at the remote node the StoreTemplate definition and implementation are available along with StoreType interface.

Figure A.7 shows a generic interface that uses inheritance. In this example, it

is assumed that the interfaces, StoreTemplateBase, StoreTemplate ⟨StoreType⟩, and StoreType are implemented at the remote node. The interface, ClothesStore is assumed to be implemented at the local node. IDL translators can be independently invoked at both nodes. Since there are no distributed objects in this example, no exchange of information takes place between the translators at compilation time.

In the above examples, the generic interface, StoreTemplate, is instantiated with a specific interface, ClothesStore. Client applications can create objects of the instantiated generic interface and invoke methods on them.

```
interface StoreType {
   // returns sales in a month,  of a store through 'amount'
   monthlySales(in int monthNumber, in int yearNumber,
                out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                   out double amount);
}

generic_interface StoreTemplate ⟨StoreType⟩ {
    // returns net profit through 'amount'
    netProfit(in int yearNumber, out double amount);
    // returns gross sales through 'amount'
    grossSales(in int yearNumber, out double amount);
}

interface ClothesStore {
    // returns sales in a month,  of a store through 'amount'
    monthlySales(in int monthNumber, in int yearNumber,
                 out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                   out double amount);
    // returns price of an item through 'price'
    priceOfItem(in long itemNumber, out double price);
}

 // an instantiation of the generic type
 StoreTemplate ⟨ClothesStore⟩ clothesStoreA;
```

Figure A.6: IDL Description of the Clothes Store Generic Interface

```
interface StoreTemplateBase {
    // returns net profit through 'amount'
    netProfit(in int yearNumber, out double amount);
    // returns gross sales through 'amount'
    grossSales(in int yearNumber, out double amount);
    // helper routine used by 'netProfit' and 'grossSales'
    calculate(in int yearNumber, in double inAmount, out double outAmount);
}

interface StoreType {
    // returns sales in a month,  of a store through 'amount'
    monthlySales(in int monthNumber, in int yearNumber,
                 out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                  out double amount);
}
generic_interface StoreTemplate ⟨StoreType⟩ : reuse StoreTemplateBase {
    // returns net profit through 'amount'
    netProfit(in int yearNumber, out double amount);
    // returns gross sales through 'amount'
    grossSales(in int yearNumber, out double amount);
}
interface ClothesStore {
    // returns sales in a month,  of a store through 'amount'
    monthlySales(in int monthNumber, in int yearNumber,
                 out double amount);
    // returns sales in a quarter,  of a store through 'amount'
    quarterlySales(in int quarterNumber, in int yearNumber,
                  out double amount);
    // returns price of an item through 'price'
    priceOfItem(in long itemNumber, out double price);
}
 // an instantiation of the generic type
 StoreTemplate ⟨ClothesStore⟩ clothesStoreA;
```

Figure A.7: IDL Description of Clothes Store Generic Interface involving Inheritance

A.4 Guard Methods

Guard methods are a useful feature to define run-time constraints on method invocations of an object. They are used for synchronization, preserving internal consistency of interacting objects. When a method that has a guard method associated with it is invoked, the guard method will be executed first. If the guard method returns 'true' via its output parameters, then the guarded method will be executed.

A.4.1 Syntax

A complete syntax of interfaces that can be used in the ICOM framework is given in figure A.8. The syntax of an interface described in the previous sections is extended with the keyword, 'guardedBy'.

⟨interface⟩ ::= ⟨interface_dcl⟩
⟨interface_dcl⟩ ::= ⟨interface_header⟩ ''{'' ⟨interface_body⟩ ''}''
⟨interface_header⟩ ::= ''interface'' ⟨identifier⟩ [⟨inheritance_spec⟩] |
''generic_interface'' ⟨identifier⟩ ''⟨'' ⟨id_list⟩ ''⟩''[⟨inheritance_spec⟩]
⟨inheritance_spec⟩ ::= '':'' [⟨reuse⟩] ⟨scoped_name⟩
⟨scoped_name⟩ ::= ⟨identifier⟩ | ''::'' ⟨identifier⟩ | ⟨scoped_name⟩ ''::'' ⟨identifier⟩
⟨id_list⟩ ::= ⟨identifier⟩ ['','' ⟨identifier⟩]*
⟨interface_body⟩ ::= ⟨export⟩*
⟨export⟩ ::= ⟨type_dcl⟩ '';'' | ⟨const_dcl⟩ '';'' |
⟨attr_dcl⟩ '';'' | ⟨op_dcl⟩ '';''
⟨op_dcl⟩ ::= ⟨identifier⟩ ⟨parameter_dcls⟩
[⟨**guardedBy**⟩ ⟨identifier⟩ ⟨parameter_dcls⟩]
⟨parameter_dcls⟩ ::= ''('' ⟨param_dcl⟩ '','' ⟨param_dcl⟩* '')'' | ''('' '')''
⟨param_dcl⟩ ::= ⟨param_attribute⟩ ⟨param_type_spec⟩⟨simple_declarator⟩
⟨param_attribute⟩ ::= ''in'' | ''out''
⟨param_type_spec⟩ ::= ⟨base_type_spec⟩ | ⟨string_type⟩ | ⟨scoped_name⟩
⟨simple_declarator⟩ ::= ⟨identifier⟩

Figure A.8: Guard Methods

A.4.2 Semantics

Any invocation of a guarded method results in the invocation and evaluation of guard method first. If the guard method evaluates to 'true' then the guarded method invocation is processed. Otherwise, the guarded method invocation enters a queue maintained by the object. The the guard method is reevaluated whenever the state of the object changes, for further processing of the guarded method invocation.

A.4.3 Examples: Interfaces with guard methods

Figure A.9 shows the Manufacturer interface that contains a method startShipment with four input parameters, brandName, itemNumber, itemCount, and storeNumber. This method is used by a client application requiring a product to be shipped to a store. The client application and the Manufacturer implementation can exist in a distributed environment. The method startShipment is guarded by the method suppliesAvailable. The relationship between a method and its guard is represented by the keyword 'guardedBy'. The guard method, suppliesAvailable takes three input parameters, brandName, itemNumber, and itemCount, and an output parameter, okay. The manufacturer cannot start shipment unless the product is available. The method suppliesAvailable will be evaluated first when an invocation on the method startShipment is received by the active object representing Manufacturer interface. The method startShipment will be executed only when there are supplies available. The availability is ensured by the guard method suppliesAvailable.

Figure A.10 shows the example of inheritance used in an interface with guard methods. There are two interfaces, ClothesStore and LocalClothesStore. The second interface inherits from the first interface. The ClothesStore interface defines a method putItemOnSale, which is guarded by another method isInventoryOkay. The

method putItemOnSale takes an input parameter, brandName, and an output parameter, salePercentage. The method isInventoryOkay takes one input parameter, brandName and one output parameter, okay. The method isInventoryOkay will be evaluated first when an invocation on the method putItemOnSale is received by the active object representing LocalClothesStore interface. The method putItemOnSale will be executed only when the method, isInventoryOkay, returns 'true' via the output parameter, okay. Otherwise, the message object enters the message queue of the LocalClothesStore instance for later evaluation.

```
interface Manufacturer {

   // this method is guarded by suppliesAvailable method
   startShipment(in string brandName,
                 in long  itemNumber,
                 in long  itemCount,
                 in long storeNumber) guardedBy

      suppliesAvailable(in string brandName,
                        in long  itemNumber,
                        in long  itemCount,
                        out boolean okay);

}
```

Figure A.9: IDL Description of Manufacturer Interface with a Guard Method

```
interface ClothesStore {
    putItemOnSale(in string brandName,
                  out long salePercentage) guardedBy

        isInventoryOkay(in string brandName,
                        out boolean okay);,

    ⋮
}

interface LocalClothesStore : reuse ClothesStore {
    updateSales(in long itemNumber,
                in long itemCount,
                in double pricePerItem);

    ⋮
}
```

Figure A.10: IDL Description of Local Clothes Store Distributed Object with a Guard Method

GLOSSARY

Abstract Class An abstract class defines a series of abstract methods (methods with optional implementation) where the implementation is defined within specific subclasses. An abstract class cannot be instantiated.

Abstract Method An operation that is declared but not implemented by an abstract class.

Acquaintance A network address of an actor that is known to another actor.

Active Object An object that has its own thread of control. It can buffer messages and process messages asynchronously by creating an autonomous thread of control to execute the method requested in each message.

Actor Address A unique identification(network address) for an actor in an application domain.

Actor A semantic entity that represents a concurrent object.

Actor Model A concurrent computation model that uses asynchronous communication of messages as the basic communication mechanism.

Ad hoc Polymorphism A category of polymorphism that involves type coercions, operator and method overloading.

Architecture The logical and physical structure of a system, forged by all strategic and tactical design decisions applied during development.

Asynchronous Communication Sending a message in a non-blocking operation: the sender of a message does not wait for the message to reach its destination.

Atomicity A constraint that requires an indivisible processing of a message. Another message processing cannot start until the current processing of a message is completed.

Atomic Object An object all of whose methods are atomic. *See* atomicity.

Behavior How an object acts and reacts, in terms of its state changes and message passing; the outwardly visible and testable activity of an object. In the actor model, a *behavior* is operated upon by the actor that represents it.

Cbox A future variable in Act++ system. *See Future*

CCITT Consultative Committee on International Telephony and Telegraphy.

Class A skeleton structure for objects. Objects are created by instantiating a class. Objects created from the same class have the same structure but individual states.

Class Conformance A class is said to conform with another class if it can be used in all contexts where the other is expected, that is, if it can understand and respond to all the method calls handled by the other.

COM Component Object Model from Microsoft.

Common Object Model An object model that is used in a distributed, heterogeneous environment for interoperating multiple programming languages that have mappings from the object model.

Concurrency ability to have more than execution simultaneously.

Concurrent Object An object whose semantics are guaranteed in the presence of multiple threads of control.

Constructor Message A message that contains creation information for an actor.

Contravariance A characteristic of inheritance that allows the parameters of a derived class method, which redefines a base class method, to be super types of the corresponding parameters of the base class method. The return type of the derived class method must be either the same type or a subtype of the return type of the base class method.

COOL Chorus Object-Oriented Layer from BBN Systems.

CORBA Common Object Request Broker Architecture from the Object Management Group.

Covariance A characteristic of inheritance that allows the parameters of a derived class method, which redefines a base class method, to be same type or subtypes of the corresponding parameters of the base class method. The return type of the derived class method must be either the same type or a subtype of the return type of the base class method.

CRT Common Run Time from Xerox.

Delegation The act of forwarding a message by an entity to some other entity. The act of one object forwarding an operation to another object, to be performed on behalf of the first object.

Distributed Active Object An active object whose state and methods are distributed and which can have at most one thread of control executing among all distributed components.

Distributed Actor An actor in a distributed environment.

Distributed Object An object whose methods and state are split and kept at distributed sites.

DOM Distributed Object Model from GTE Laboratories.

Dynamic Binding Binding denotes the association of a name with a class; dynamic binding is a binding in which the name/class association is not made until the object designated by the name is created at execution time.

Encapsulation The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.

Framework A collection of classes that provide a reusable design for a particular domain; a framework thus exports a number of individual classes and mechanisms that clients can use or adapt.

Future A type whose instance blocks an invoking process that tries to access its value when the value is not yet set and lets the invoking process to continue, by supplying its value when the value is available.

Generics Generic types in languages such as Modula-3, and Ada 95. *See* Generic Type.

Generic Type A class that serves as a skeleton for other types, in which the generic type may be parameterized by other types. A generic type must be instantiated before its variables can be created. The terms 'Generic Type' and 'Parameterized Type' are interchangeable.

Guarded Method method that has a guard method associated with it.

Guard Method A method that returns a boolean value. A guard method is called first, before a guarded method that it guards, is called. The guarded method can be executed only when the guard method returns true.

HERON A system supporting truly distributed objects developed at the Free University of Berlin, Germany.

ICOM Interoperable Common Object Model.

ILU Inter Language Unification from Xerox.

Inclusion Polymorphism Inheritance of abstract methods by a subclass from a superclass. Invocation of an abstract method in the superclass results in the execution of the corresponding method in the subclass.

Information Hiding The process of hiding the irrelevant details that do not contribute to the essential characteristics expected of an entity.

Inheritance A relationship among classes, where in one class shares the structure or behavior defined in another class.

Inheritance Anomaly The inherent problem that forces a subclass implementor to redefine most of the superclass methods while inheriting synchronization constraints of the superclass.

Interface A programming entity that defines the methods that are supported by an object, but it does not provide any implementation for the methods.

Interoperability the ability of the components of a software system to coexist, communicate, and cooperate with one another to achieve a common goal in a distributed and heterogeneous environment involving disparate implementation languages, operating systems, and hardware platforms.

Invariance A characteristic of inheritance that does not allow a derived class to change the types of the arguments, while redefining a base class method.

ISL Interface Specification Language defined in the ILU system from Xerox

ISO International Standards Organization.

Marshalling the process of building up a sequence of bytes that contains the values of the arguments or results in a proper format.

Message An operation that one object performs on another object.

Message Queue A queue for messages that an actor possesses.

Method A procedure defined as part of an object. A method may be part of an object's interface, in which case other objects may request invocations of the method by means of message passing.

Method Table A per-class data structure that contains entries for each method of the object's interface, with each entry having a method name, class name, and a

pointer to an instance of a method object that is capable of decoding arguments from a message object and invoke a call on an object that contains *method* in its interface.

Method Overloading existence of more than one method with the same name but different signatures in the interface of an object.

NewOwner Message A message in the extended actor message set that is used to implement the ICOM framework. Used for declaring and acknowledging ownership of message processing control by the peer proxies of a truly distributed object.

Node An entity representing an abstract machine in the ICOM framework.

Object A software entity with its own state and a number of procedures to manipulate this state. The state of an object is encapsulated by an interface that constitutes an abstraction boundary between the object and its environment.

Object Based Programming A method of programming in which programs are organized as cooperative collections of objects, each of which represents an instance of some type, and whose types are all members of a hierarchy of types united via other than inheritance relationships.

Object Model The collection of principles that form the foundation of object-oriented design; a software engineering paradigm emphasizing the principles of abstraction, encapsulation, modularity, hierarchy, typing, and concurrency.

Object-Oriented Programming A method of implementation in which programs are organized as cooperative collections of objects, each of which represents an

instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

Object State *See* State.

ODP Open Distributed Processing

OMG Object Management Group.

One-way Interoperability The ability of a language to use the software components in other languages that use a feature which is not present in the language.

Operator Overloading The characteristic of an operator that can accept operands of different types.

OwnershipRequest Message A message in the extended actor message set that is used to implement the ICOM framework. Used for requesting the ownership of message processing control by a peer proxy.

OwnershipRelease Message A message in the extended actor message set that is used to implement the ICOM framework. Used for relinquishing the ownership of message processing control by the peer proxy to another peer proxy.

Parameterized Type *See* Generic Type.

Parametric Polymorphism A category of Universal polymorphism that allows a general concept to be instantiated with one or more types.

Passive Object An object that does not encompass its own thread of control.

PeerInvoke Message A message in the extended actor message set that is used to implement the ICOM framework. Used for invoking a message on a peer proxy by another peer proxy of the same distributed object.

Peer Proxy A component of a truly distributed object.

PeerUnblock Message A message in the extended actor message set that is used to implement the ICOM framework. Used for unlocking the locked peer proxies that were locked as a result of a message processing.

Polymorphism an abstraction mechanism that represents a quality or state of being able to assume different forms.

Parameterized type the abstraction of a generic type based on the common features of existing types. These generic types can be instantiated with other types.

Proxy A surrogate object representing a remote object.

Reply Message A message in the extended actor message set that is used to implement the ICOM framework. Used for sending a reply through an output parameter in an IDL description; is a Cbox (future variable) in the ICOM framework.

RPC Remote Procedure Call.

Signature The complete profile of an operation's formal arguments and return type.

SOM System Object Model from International Business Machines.

Split Object A distributed object whose components are split arbitrarily and distributed.

State The cumulative results of the behavior of an object; one of the possible conditions in which an object may exist; At any given point in time, the state of an

object encompasses all of the properties of the object plus the current values of each of these properties.

Strongly Typed A characteristic of a programming language, according to which all expressions are guaranteed to be type-consistent.

Subclass A class defined as a refinement of another class by means of inheritance.

Superclass A class used to define other classes by means of refinement under inheritance.

Synchronous Communication Sending a message in a blocking operation: the sender of a message waits for a response for the message sent.

Synchronization The act of coordination among a group of concurrent objects in achieving a common task.

Synchronized Object An object some of whose methods have certain constraints (expressed in terms of a method that returns a boolean value) that must be evaluated to true, for invocations on the methods to be processed.

Templates Generic types in C++. *See* Generic Types

Thread of Control A single process. The start of a thread of control is the root from which independent dynamic action within a system occurs; a given system may have many simultaneous threads of control, some of which may dynamically come into existence and then cease to exist.

Truly Distributed Object A distributed object that has its various components split among distributed sites. The components are related by a superclass and a subclass relationship.

Type The definition of the domain of allowable values that an object may possess and the set of operations that may be performed upon the object.

Type Conformance One type is said to conform with another type if it can be used in all contexts where the other is expected.

Universal Polymorphism A category of polymorphism that involves inclusion polymorphism and parameterized types.

Unmarshalling the reverse process to marshalling; recovers values from the sequence of bytes.

Virtual Method Abstract method in C++. *See* Abstract Method.

REFERENCES

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer, New York, NY, 1996.

[2] Martin Abadi, Luca Cardelli, and Pierre-Louis Curien. Formal Parametric Polymorphism. Technical report, SRC, SRC Research Report 109, Digital Equipment Corporation, July 1993.

[3] Bruno Achauer. The DOWL Distributed Object-Oriented Language. *Communications of the ACM*, 36(9):48–55, September 1993.

[4] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to Java language. In *Proceedings OOPSLA '97*, pages 49–65, October 1997.

[5] Gul Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. The MIT Press, Cambridge, MA, 1986.

[6] Gul Agha, Svend Frolund, and WooYoung Kim. Abstraction and Modularity Mechanisms for Concurrent Computing. *IEEE Parallel and Distributed Technology*, pages 3–14, May 1993.

[7] Gul Agha and Carl Hewitt. Concurrent Programming using Actors. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 37–53. The MIT Press, 1993.

[8] Gul Agha, Peter Wegner, and Akinori Yonezawa, editors. *Research Directions in Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, MA, 1993.

[9] Pierre America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *Proceedings ECOOP '87*, pages 234–242, July 1987.

[10] Pierre America. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *Proceedings OOPSLA '90*, pages 161–168, October 1990.

[11] Egin P. Andersen and Trygve Reenkaug. System Design by Composing Structures of Interacting Objects. In *Proceedings ECOOP '92*, pages 133–152, July 1992.

[12] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[13] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution: An Ada-based Approach*. Addison Wesley, Reading, MA, 1991.

[14] Guiseppe Attardi, Cinzia Bonini, Maria Rosaria Boscotrecase, Tito Flagella, and Mauro Gaspari. Metalevel programming in CLOS. In *Proceedings ECOOP '89*, pages 243–256, July 1989.

[15] Andrew Birrel, Greg Nelson, Susan Owicki, and Edward Wobber. Network Objects. In *SIGOPS '93*, pages 217–230, December 1993.

[16] Grady Booch. *Object-Oriented Analysis and Design with Applications, Second Edition*. Addison Wesley, Reading, MA, 1994.

[17] Gilad Bracha and William Cook. Mixin-based Inheritance. In *Proceedings OOPSLA '90*, pages 303–311, October 1990.

[18] Jean-Pierre Briot. An Experiment in Classification and Specialization of Synchronization Schemes. In *Object Technologies for Advanced Software, Second JSSST International Symposium ISOTAS '96 Proceedings*, pages 227–249, March 1996. Printed in LNCS 1049.

[19] Jean-Pierre Briot and Akinori Yonezawa. Inheritance and synchronization in concurrent OOP. In *Proceedings ECOOP '87*, pages 32–40, July 1987.

[20] K. Brockschmidt. *Inside OLE 2*. Microsoft Press, Redmond, 1994.

[21] Gerald Brose. JacORB – Design and Implementation of a Java ORB. In *Proceedings of the International Working Conference on Distributed Applications and Interoperable Systems, DAIS'97, Cotbus, Germany*, September 1997.

[22] Gerald Brose, Klaus-Peter Lohr, and Andre Spiegel. Java does not Distribute. In *Proceedings TOOLS Pacific '97, Melbourne, Australia*, November 1997.

[23] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded Polymorphism for Object-Oriented Languages. In *The Fourth International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[24] Peter S. Canning, Willian R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for Strongly-Typed Object-Oriented Programming. In *Proceedings OOPSLA '89*, pages 457–467, October 1989.

[25] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):470–522, December 1985.

[26] Eduardo Casais. An Incremental Class Reorganization Approach. In *Proceedings ECOOP '92*, pages 114–132, July 1992.

[27] Siva Challa. Towards an Interoperable Reflective Common Object Model. In *the Workshop on Multi-language object Models, OOPSLA '94, Portland*, October 1994.

[28] Siva Challa and Dennis Kafura. Using reflection for implementing ICOM, An Interoperable Common Object Model. Technical Report TR95-23, Virginia Tech, Blacksburg, VA, 1995.

[29] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proceedings ECOOP '92*, pages 33–56, July 1992.

[30] Shigeru Chiba. Open C++ Release 1.2 Programmer's Guide. Technical report, University of Tokyo, Department of Computer Science, 1993.

[31] Shigeru Chiba and Takashi Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings ECOOP '93*, pages 482–501, July 1993.

[32] Andrew K. Chiu and Jay Parekh. Class: A collection of objects. *Journal of Object-Oriented Programming*, pages 29–32, May 1992.

[33] Randy Chow and Theodore Johnson. *Distributed Operating Systems and Algorithms*. Addison Wesley, Reading, MA, 1997.

[34] Alberto Coen-Porisini, Luigi Lavazza, and Roberto Zicari. Assuring Type Safety of Object-Oriented Languages. *Journal of Object-Oriented Programming*, pages 25–30, February 1994.

[35] Pierre Cointe. Metaclasses are first class: the ObjVLisp model. In *Proceedings OOPSLA '87*, pages 156–167, October 1987.

[36] W. R. Cook. A proposal for making Eiffel type-safe. In *Proceedings ECOOP '89*, pages 57–70, July 1989.

[37] William Cook and Jens Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proceedings OOPSLA '89*, pages 433–443, October 1989. Printed in SIGPLAN Notices, 24(10), October 1989.

[38] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is Not Subtyping. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 125–135, January 1990.

[39] Doug Cutting, Bill Janssen, Mike Spreitzer, and Farrel Wymore. ILU Reference manual. Technical report, Xerox Corporation, Palo Alto, CA, 1993.

[40] Scott Danforth. A Bird's Eye View of SOM. *IBM Personal Systems Programming*, pages 1–9, September 1992.

[41] Scott Danforth and Chris Tomlinson. Type Theories and Object-Oriented Programming. *ACM Computing Surveys*, 20(1):29–72, March 1988.

[42] Ph. Darondeau, P. Le Guernic, and M. Raynal. Types in a Mixed Language System. *BIT*, 21:246–254, 1981.

[43] Amitabh Dave, Mohlalefi Sefika, and Roy H. Campbell. Proxies, Application Interfaces, and Distributed Systems. Technical report, UIUC, Illinois, 1993.

[44] Stephen R. Davis. C++ Objects That Change Their Types. *Journal of Object-Oriented Programming*, pages 27–32, July-August 1992.

[45] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In *Proceedings ECOOP '87*, pages 151–170, July 1987.

[46] Bo Einarsson. Mixed Language Programming. *Software-Practice and Experience*, 14(4):383–395, April 1984.

[47] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, Reading, MA, 1994.

[48] Jacques Ferber. Conceptual Reflection and Actor Languages. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 177–193. North-Holland, 1988.

[49] Jacques Ferber. Computational Reflection in Class Based Object-Oriented Languages. In *Proceedings OOPSLA '89*, pages 317–326, October 1989.

[50] S. Finke, P. Jahn, O. Langmack, K. P. Lohr, I. Piens, and Th. Wolff. Distribution and inheritance in the HERON approach to heterogeneous computing. In *Thirteenth Internation Conference on Distributed Computing Systems, Pittsburgh*, 1993.

[51] International Organization for Standardization. Information Processing - Open Systems Interconnection - Specification of Abstract Syntax Notation One(ASN.1). Technical Report ISO 8824/CCITT X.208, 1987.

[52] International Organization for Standardization. Information Processing - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One(ASN.1). Technical Report ISO 8825/CCITT X.209, 1987.

[53] Ira R. Forman, Scott Danforth, and Hari Madduri. Composition of Before/After metaclasses in SOM. In *Proceedings OOPSLA '94*, October 1994.

[54] Svend Frolund and Gul Agha. A Language Framework for Multi-Object Coordination. In *Proceedings ECOOP '93*, July 1993.

[55] Lucy Garnette. Wrapping objects. *Journal of Object-Oriented Programming*, 9(8), 1993.

[56] Giorgio Ghelli. A Static Type System for Message Passing. In *Proceedings OOPSLA '91*, pages 129–145, October 1991.

[57] Phillip B. Gibbons. A Stub Generator for Multi-language RPC in Heterogeneous Environments. *IEEE Transactions on Software Engineering*, 13(1):77–87, January 1987.

[58] Ennio Grasso. Passing object by value in CORBA. In *Trends in Distributed Systems - CORBA and Beyond*, pages 43–56, October 1996. Printed in LNCS 1161.

[59] Nicolas Graube. Metaclass Compatibility. In *Proceedings OOPSLA '89*, pages 305–315, October 1989.

[60] Object Management Group. Common Object Request Broker: Architecture and Specification. Technical Report OMG Document Number 97.9.1, Rev. 2.1, OMG, 1997.

[61] Sabine Habert and Laurence Mosseri. COOL: Kernel support for object-oriented environments. In *Proceedings OOPSLA '90*, pages 269–277, October 1990.

[62] Samuel P. Harbison, editor. *Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1992.

[63] Franz J. Hauck. Inheritance Modeled with Explicit Bindings: An Approach to Typed Inheritance. In *Proceedings OOPSLA '93*, pages 231–239, October 1993. Printed in SIGPLAN Notices, 28(10), October 1993.

[64] Roger Hayes, Steve W. Manweiler, and Richard D. Schlichting. A Simple System for Constructing Distributed, Mixed-Language Programs. *Software-Practice and Experience*, 18(7):641–659, July 1988.

[65] Roger Hayes and Richard D. Schlichting. Facilitating Mixed Language Programming in Distributed Systems. *IEEE Transactions on Software Engineering*, 13(12):1254–1264, December 1987.

[66] Patrick Hellemans, Frank Steegmans, Hans Vanderstraeten, and Han Zuidweg. Implementation of Hidden Concurrency in CORBA Clients. In *Trends in Distributed Systems - CORBA and Beyond*, pages 30–42, October 1996. Printed in LNCS 1161.

[67] Robert Henderson and Benjamin Zorn. A Comparison of Object-Oriented Programming in Four Modern Languages. Technical Report CU-CS-641-93, University of Colorado, Boulder, Department of Computer Science, July 1993.

[68] Ian M. Holland. Specifying Reusable Components Using Contracts. In *Proceedings ECOOP '92*, pages 287–308, July 1992.

[69] Chris Horn. Conformance, Genericity, Inheritance and Enhancement. In *Proceedings ECOOP '87*, pages 223–233, July 1987.

[70] IBM. SOMObjects Developer Toolkit Technical Overview, Version 2.0. Technical report, IBM, New York, November 1993.

[71] Harry H. Porter III. Separating the Subtype Hierarchy From the Inheritance of Implementation. *Journal of Object-Oriented Programming*, pages 20–29, February 1992.

[72] C. Jacquemot, P. Gautron, H. G. Baumgarten, F. Herrmann, J. Mukerji, H. Hartlage, and P. S. Jensen. COOL-IDL Extensions to C++ to Support Distributed Programming. Technical report, Chorus Group, Esprit Project No 6603, 1994.

[73] C. Jacquemot, P. Gautron, H. G. Baumgarten, F. Herrmann, J. Mukerji, H. Hartlage, and P. S. Jensen. COOL: The CHORUS CORBA Compliant Framework. Technical report, Chorus Group, Esprit project No 6603, 1994.

[74] Suresh Jagannathan and Gul Agha. A Reflective Model of Inheritance. In *Proceedings ECOOP '92*, pages 350–371, July 1992.

[75] Ralph E. Johnson and Jonathan M. Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, pages 31–34, November/December 1991.

[76] Michael B. Jones, Richard F. Rashid, and Mary R. Thompson. Matchmaker: An interface specification language for distributed processing. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 225–235, January 1985.

[77] D. G. Kafura and R. G. Lavender. Concurrent Object-Oriented Languages and Inheritance Anomaly. In Thomas L. Casavant, Pavel Tvrdik, and Frantisek Plasil, editors, *Parallel Computers: Theory and Practice*, pages 221–264. IEEE Computer Society Press, 1996.

[78] Dennis Kafura, Siva Challa, and Greg Lavender. Workshop on Multi-Language Object Models. In *Addendum to Proceedings OOPSLA '94*, October 1994.

[79] Dennis Kafura, Manibrata Mukherji, and Greg Lavender. ACT++: A class library for concurrent programming in C++ using actors. *Journal of Object-Oriented Programming*, pages 47–62, October 1993.

[80] Kenneth M. Kahn. Objects - A fresh look. In *Proceedings ECOOP '89*, pages 207–223, July 1989.

[81] D. Katiyar, D. Luckham, and J. C. Mitchell. A type system for prototyping languages. In *Proceedings of the 21st ACM Symposium on Principles of Programming Languages*, January 1994.

[82] Arjun Khare. ACT++ 3.0 : Implementation of the Actor Model Using POSIX Threads. Master's thesis, Virginia Tech, Blacksburg, VA, 1994.

[83] Gregor Kiczales, James des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, MA, 1991.

[84] Andrew Koenig. Templates as Interfaces. *Journal of Object-Oriented Programming*, pages 51–55, September 1991.

[85] Dimitri Konstantas. Object Oriented Interoperability. In D. Tsichritzis, editor, *Visual Objects*. University of Geneva, 1993.

[86] Kai Koskimies and Juha Vihavainen. The Problem of Unexpected Subclasses. *Journal of Object-Oriented Programming*, pages 53–59, October 1992.

[87] David Kranz, Kirk Johnson, and Anant Agarwal. Integrating message-passing and shared-memory: Early experience. *ACM SIGPLAN Notices*, 28(7):54–63, July 1993.

[88] David Alex Lamb. IDL: Sharing intermediate representations. *ACM Transactions on Programming Languages*, 9(3):297–318, July 1987.

[89] John Lamping. Typing the Specialization Interface. In *Proceedings OOPSLA '93*, pages 201–214, October 1993. Printed in SIGPLAN Notices, 28(10), October 1993.

[90] Konstantin Laufer and Martin Odersky. Self-Interpretation and Reflection in a Statically Typed Language. In *OOPSLA '93 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1993.

[91] Rodger Lea, Christian Jacquemot, and Eric Pillevesse. COOL: System support for distributed programming. *Communications of the ACM*, 36(9):37–47, September 1993.

[92] Gary T. Leavens and William E. Weihl. Reasoning about Object-Oriented Programs that use Subtypes. In *Proceedings OOPSLA '90*, pages 212–223, October 1990.

[93] H J Lee and W T Tsai. A New Partial Inheritance Mechanism and Its Applications. *Journal of Object-Oriented Programming*, pages 53–63, July-August 1993.

[94] Donald Liib. A Note on Communicational Reflection. In *ECOOP '92 Workshop on Object-Oriented Reflection and Metalevel Architectures*, July 1992.

[95] Barbara Liskov. Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 150–159, January 1986.

[96] Barbara Liskov and Jeannette M. Wing. A New Definition of the Subtype Relation. In *Proceedings ECOOP '93*, pages 118–141, July 1993.

[97] Barbara Liskov and Jeannette M. Wing. Specifications and Their Use in Defining Subtypes. In *Proceedings OOPSLA '93*, pages 16–28, October 1993. Printed in SIGPLAN Notices, 28(10), October 1993.

[98] Ole Lehrmann Madsen. What Object-Oriented Programming May Be and What It Does Not Have To Be. In *Proceedings ECOOP '89*, pages 1–20, July 1989.

[99] Ole Lehrmann Madsen and Boris Magnusson. Strong Typing of Object-Oriented Languages Revisited. In *Proceedings OOPSLA '90*, pages 140–149, October 1990.

[100] Ole Lehrmann Madsen and Birger Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings OOPSLA '89*, pages 397–406, October 1989.

[101] Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison Wesley, Reading, MA, 1993.

[102] Pattie Maes. Concepts and Experiments in Computational Reflection. In *Proceedings OOPSLA '87*, pages 147–155, October 1987.

[103] Pattie Maes and Daniele Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, 1988.

[104] Silvano Maffeis. A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming. In *Workshop on Object-Based Distributed Computing, ECOOP '93*, pages 213–224, July 1993.

[105] Silvano Maffeis. A Flexible System Design to Support Object-Groups and Object-Oriented Distributed Programming. Technical Report IFI TR 94.02, University of Zurich, Dept of Computer Science, 1994. to appear in LNCS.

[106] Messac Makpangou, Yvon Gourhant, Jean-Pierre Le Narzul, and Marc Shapiro. Structuring Distributed Applications as Fragmented Objects. Technical Report INRIA 1404, INRIA, France, January 1991.

[107] Jawahar Malhotra. Dynamic Extensibility in a Statically-Compiled Object-Oriented Language. In *Object Technologies for Advanced Software*, pages 297–314, November 1993. Printed in LNCS 742.

[108] Frank Manola. X3H7 Object Model Features Matrix, May 1994.

[109] Frank Manola and Sandra Heiler. An Approach to Interoperable Object Models.

[110] Frank Manola and Sandra Heiler. A "RISC" Object Model for Object System Interoperation: Concepts and Applications. Technical Report TR-0231-08-93-165, GTE Laboratories, Waltham, MA, December 1993.

[111] Satoshi Matsuoka. *Language Features for Reuse and Extensibility in Concurrent Object-Oriented Languages.* PhD thesis, University of Tokyo, Tokyo, Japan, April 1993.

[112] Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming. In *Proceedings ECOOP '91*, pages 231–250, July 1991.

[113] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. The MIT Press, Cambridge, MA, 1993.

[114] Mark Maybee, , Dennis M. Heimbigner, and Leon J. Osterweil. Multilanguage interoperability in distributed systems: Experience Report. In *Proceedings of the 18th Internation Conference on Software Engineering*, March 1996.

[115] Mark Maybee, Leon J. Osterweil, and Stephen D. Sykes. Q: A multi-lingual interprocess communications system for software environment implementation. Technical Report CU-CS-46-90, University of Colorado, Boulder, Dept. of Computer Science, June 1990.

[116] Paul L. McCullough. Transparent Forwarding: First Steps. In *Proceedings OOPSLA '87Proceedings*, pages 331–341, October 1987.

[117] Robert W. Mecklenburg. *Towards a Language Independent Object System*. PhD thesis, University of Utah, Salt Lake City, Utah, June 1991.

[118] Anurag Mendhekar and Daniel P. Friedman. Towards a Theory of Reflective Programming Languages. In *OOPSLA '93 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1993.

[119] Bertrand Meyer. Genericity versus Inheritance. In *Proceedings OOPSLA '86*, pages 391–405, September 1986.

[120] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, NY, 1992.

[121] Josephine Micallef. Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, 1(1):12–35, april/may 1988.

[122] Sun Microsystems. NEO. http://www.sun.com/solaris/neo/solaris_neo/.

[123] Sun Microsystems. XDR: External Data Representation Standard(RFC 1014). Technical report, Network Information Center, SRI International, June 1987.

[124] Sun Microsystems. Java Remote Method Invocation Specification. Technical Report Revision 1.4, Sun Microsystems, Palo Alto, CA, February 1997.

[125] John C. Mitchell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 109–124, January 1990.

[126] R. Morrison, A. L. Brown, R. Carrick, R. C. H. Connor, A. Dearle, and M. P. Atkinson. Polymorphism, Persistence and Software Reuse in a Strongly Typed Object-Oriented Environment. *Software Engineering Journal*, pages 199–205, November 1987.

[127] Thomas J. Moubray and Ron Zahzvi. *The Essential CORBA*. John Wiley and Sons, New York, NY, 1995.

[128] Patrick A. Muckelbauer and Vincent F. Russo. Building a Large-Scale Distributed Object System for a Multilingual Programming Environment. Technical Report CSD-TR-93-022, Purdue University, Dept. of Computer Science, April 1993.

[129] Warwick B. Mugridge, John Hamer, and John G. Hosking. Multi-Methods in a Statically-Typed Programming Langugage. In *Proceedings ECOOP '91*, pages 307–324, July 1991.

[130] S. Muralidharan and Bruce W. Weide. Should Data Abstraction be Violated to enhance Software Reuse? In *8th Annual National Conference on Ada Technology 1990*, pages 515–524, 1990.

[131] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 132–145, January 1997.

[132] Colin Myers, Chris Clack, and Ellen Poon. *Programming with Standard ML*. Prentice Hall, 1993.

[133] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[134] John R. Nicol, C. Thomas Wilkes, and Frank A. Manola. Object Orientation in Heterogeneous Distributed Computing Systems. *IEEE Computer*, pages 57–67, June 1993.

[135] O. M. Nierstrasz. Active Objects in Hybrid. In *Proceedings OOPSLA '87*, pages 243–253, October 1987.

[136] Oscar Nierstrasz. A Survey of Object-Oriented Concepts. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 3–21. ACM Press and Addison Wesley, 1989.

[137] Oscar Nierstrasz. Regular types for Active Objects. In *Proceedings OOPSLA '93*, pages 1–15, October 1993. Printed in SIGPLAN Notices, 28(10), October 1993.

[138] Martin Odersky and Philip Wadler. Pizza into Java: Translating Theory into Practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, January 1997.

[139] Atsushi Ohori and Peter Buneman. Static Type Inference for Parametric Classes. In *Proceedings OOPSLA '89*, pages 445–456, October 1989.

[140] Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley and Sons, New York, NY, 1996.

[141] International Standards Organization. Reference Model for Open Distributed Processing. Technical Report WG7 N. 885, Open Distributed Processing, New South Wales, Australia, November 1993.

[142] Harold Ossher and William Harrison. Combination of Inheritance Hierarchies. In *Proceedings OOPSLA '92*, pages 25–40, October 1992.

[143] Nicholas Oxhoj, Jens Palsberg, and Michael I. Schwartzbach. Making Type Inference Practical. In *Proceedings ECOOP '92*, pages 329–349, July 1992.

[144] Hewlett Packard. ORB Plus 2.0. http://www.hp.com/gsy/orbplus.html.

[145] Andreas Paepcke, editor. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, Cambridge, MA, 1993.

[146] Jens Palsberg and Michael I. Schwartzbach. Type Substitution for Object-Oriented Programming. In *Proceedings OOPSLA '90*, pages 151–160, October 1990.

[147] Jens Palsberg and Michael I. Schwartzbach. Object-Oriented Type Inference. In *Proceedings OOPSLA '91*, pages 146–161, October 1991.

[148] Jens Palsberg and Michael I. Schwartzbach. What is Type-Safe Code Reuse? In *Proceedings ECOOP '91*, pages 325–341, July 1991.

[149] M. Papathomas. Concurrency Issues in Object-Oriented Programming Languages. In D. Tsichritzis, editor, *Object-Oriented Development*, pages 207–245. Center Universitaire D'informatique, July 1989.

[150] Claus H. Pedersen. Extending Ordinary Inheritance Schemes to Include Generalization. In *Proceedings OOPSLA '89*, pages 407–417, October 1989.

[151] John Peterson and Mark Jones. Implementing Type Classes. In *SIGPLAN '93 Conference on Programming Language Design and Implementation, Albuquerque*, pages 227–236, June 1993. also appeared in ACM SIGPLAN Notices, 28,6, June '93.

[152] Ramana Rao. Implementational reflection in Silica. In *Proceedings ECOOP '91*, pages 251–267, July 1991.

[153] Ward Rosenberry, David Kenney, and Gerry Fisher. *Understanding DCE*. O'Reilly and Associates, Sebastopol, CA, 1993.

[154] Francois Rousseau and Jacques Malenfant. Browsing in Explicit Metaclass Languages: An Essay in Reflective Programming Environments. In *OOPSLA '93 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1993.

[155] M. Sakkinen. Disciplined Inheritance. In *Proceedings ECOOP '89*, pages 39–56, July 1989.

[156] John H. Saunders. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, pages 5–11, march-april 1989.

[157] Etsuya Shibayama. Reuse of Concurrent Object Descriptions. In *Proceedings of TOOLS 3, Sydney, Australiz*, pages 254–266, November 1990.

[158] Jon Siegel. *CORBA Fundamentals and Programming.* John Wiley and Sons, New York, NY, 1996.

[159] Pradeep K. Sinha. *Distributed Operating Systems: Concepts and Design.* IEEE Computer Society Press, New York, NY, 1997.

[160] Brian C. Smith. Reflection and semantics in Lisp. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, January 1983.

[161] Alan Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings OOPSLA '86*, pages 38–45, September 1986.

[162] Alan Snyder. Modeling the C++ Object Model: An Application of an Abstract Object Model. In *Proceedings ECOOP '91*, pages 1–20, July 1991.

[163] Richard Mark Soley and Christopher M. Stone, editors. *Object Management Architecture Guide, Revision 3.0.* John Wiley and Sons, New York, NY, 1995.

[164] Bjarne Stroustrup. An overview of C++. *ACM SIGPLAN Notices*, pages 7–18, October 1986.

[165] Bjarne Stroustrup. What is Object-Oriented Programming? *IEEE Software*, pages 10–20, May 1988.

[166] Bjarne Stroustrup. *The C++ Programming Language, 3rd Edition.* Addison Wesley, Reading, MA, 1997.

[167] Ichiro Suzuki and Tadao Kasami. A Distributed Mutual Exclusion Algorithm. *ACM Transactions on Computer Systems*, 3(4):344–349, November 1985.

[168] BBN Systems and Technologies. Cronus technical summary, Release 2.0, March 1991.

[169] Clemens A. Szyperski. Import is not inheritance - Why we need both: Modules and Classes. In *Proceedings ECOOP '92*, pages 19–32, July 1992.

[170] David Taenzer, Murthy Ganti, and Sunil Podar. Object-Oriented Software Reuse: The Yoyo Problem. *Journal of Object-Oriented Programming*, pages 30–35, September/October 1989.

[171] David Taenzer, Murthy Ganti, and Sunil Podar. Problems in Object-Oriented Software Reuse. In *Proceedings ECOOP '89*, pages 25–38, July 1989.

[172] S. Tucker Taft. Ada 9X: From abstraction-oriented to object-oriented. In *Proceedings OOPSLA '93*, pages 127–136, October 1993. Printed in SIGPLAN Notices, 28(10), October 1993.

[173] Antero Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438–479, September 1996.

[174] BBN Technologies. Corbus. http://www.bbn.com/products/dpom/corbus.htm.

[175] IONA Technologies. Orbix 2 Programming Guide. Technical report, IONA, Dublin, Ireland, November 1995.

[176] Chris Tomlinson and Vineet Singh. Inheritance and synchronization with Enabled Sets. In *Proceedings OOPSLA '89*, pages 103–112, October 1989.

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

[178] Mladen A. Vouk. On the Cost of Mixed Language Programming. *ACM SIGPLAN Notices*, 19(12):54–60, December 1984.

[179] Mitchell Wand and Daniel P. Friedman. The Mystery of the Tower Revealed: A Non-Reflective Description of the Reflective Tower. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 111–134. North-Holland, 1988.

[180] T. Watanabe and A. Yonezawa. An Actor-based Metalevel Architecture for Group-wide Reflection. In *Foundations of Object-Oriented Languages*, pages 405–425, may/june 1990. Printed in LNCS 489.

[181] Takuo Watanabe and Akinori Yonezawa. Reflection in an Object-Oriented Concurrent Language. In *Proceedings OOPSLA '88*, pages 306–315, September 1988.

[182] P. Wegner and S. B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In *Proceedings ECOOP '88*, pages 55–77, July 1988.

[183] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings OOPSLA '87*, pages 168–182, October 1987.

[184] Peter Wegner, William Scherlis, James Purtilo, David Luckham, and Ralph Johnson. Object-Oriented Megaprogramming. In *Proceedings OOPSLA '92*, pages 392–396, October 1992.

[185] Mark Weiser, Alan Demers, and Carl Hauser. The Portable Common Runtime Approach to Interoperability. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 114–122, December 1989.

[186] Mario Wolczko. Encapsulation, Delegation and Inheritance in Object-Oriented Languages. *IEEE Transactions on Software Engineering*, pages 95–101, March 1992.

[187] Th. Wolff and K. P. Lohr. Transparently Programming Heterogeneous Distributed Systems. In Alexander Schill, Christian Mittasch, Otto Spaniol, and Claudia Popien, editors, *Distributed Platforms*, pages 399–411. Chapman & Hill, 1996.

[188] Thomas Wolff. Transparent Object Distribution and Remote Inheritance. In Amr Zaky and Ted Lewis, editors, *Tools and Environments for Parallel and Distributed Systems*, pages 279–303. Kluwer Academic, 1996.

Index

S

T

U

VITA

Siva Challa was born in 1967 in a beautiful small town of Nagarjunasagar on the banks of the river Krishna, in the Telugu speaking state of Andhra Pradesh in India. He never stayed at any place for more than 5 years until he started his Ph.D. at Virginia Tech! He received a bachelors degree in Computer Science from Andhra University, India, in 1988. His interest in higher studies made him join the graduate program in Artificial Intelligence and Robotics at the University of Hyderabad, India. He received a masters degree in 1990. He then wanted to get a doctoral degree in Computer Science. He joined Virginia Tech in 1991 after a brief stay at the University of Minnesota, Duluth. After his initial futile attempts to work on database research, he thought that object-oriented programming is a way to go. While pursuing the doctoral degree, he got married to his lovely wife, Madhavi Latha, in 1996. He would like to show his wife several interesting places in the United States, which he deprived her of while working on his Ph.D. He wants to work in the industry for some time before deciding about the eventual future plans.