Using Reflection for Implementing ICOM, An Interoperable Common Object Model

Siva Challa and Dennis Kafura

Department of Computer Science
McBryde Hall, Virginia Tech
Blacksburg, VA 24061-0106
Phone: (540) 231-6931, Fax: (540) 231-6075
E-mail: {siva,kafura}@cs.vt.edu

Abstract

Reuse of already developed object-oriented software components is reduced when software is written in different object-oriented languages, or when it is not easy to relocate the software components from one machine to another machine. Software development in a distributed and heterogeneous environment is a solution to this problem. Several common object models have been defined for software development in a distributed and heterogeneous environment. Most of the existing common object models do not agree upon a common set of object-oriented features. These models are weak (i.e., they have only a modest subset of object-oriented features) because they support both object-oriented and non object-oriented languages and mapping of the model into non object-oriented languages is a complex problem. In this paper an interoperable common object model (ICOM) that supports statically typed object-oriented languages is presented. The ICOM model has an extensive set of object-oriented features along with a framework that uses reflection techniques to support the model. The architecture of ICOM framework is described and a detailed account on how reflection is used is given. A prototype involving dynamic method binding feature is explained in detail.

1 Introduction

With the increasing complexity of software, reuse is becoming a critical aspect of the software development process. Reuse of software by relocating the software components from one machine to another is proving to be a difficult task. Software development in a distributed environment is a solution to this problem. With ever-growing hardware technology, heterogeneity is increasing among the available machine architectures. The challenge then becomes one of developing software in a distributed, and heterogeneous environments.

Software development using object-oriented technology in distributed and heterogeneous environments has been found to be effective because of concepts such as message passing for communication, encapsulation for hiding implementation details, and inheritance for specializing default system behavior with application specific semantics. Most of the existing object-oriented software is developed in different object-oriented languages, since there is no universal object-oriented language for software development. To integrate the software components developed in different languages, several common object models have been defined [Som93] [Omg91] [Open93] [Mano93] [Broc94]. A typical common object model is defined by the features contained in an Interface Definition Language (IDL). The object model defined by the IDL can be used for interoperability of components from different languages.

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 [Chal94]. These common denominator models are limited: 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) [Omg91] since 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. To provide a manageable representation in non object-oriented languages, the features of the common object model are restricted. The limitations on the interoperability and component reuse implied by such weakened object models restricts the availability of services to end-users, and restrains software productivity as it forces the rebuilding of existing software components.

In this paper an interoperable common object model (ICOM) that has an extensive set of object-oriented features (divided into two groups: core set and extended set, see sections 3.1 and 3.2) is presented. A framework that supports the model in a distributed heterogeneous environment is presented. Uniform mapping techniques are used to map the model into object-oriented languages. For this purpose reflection techniques [Ferb89] [Maes87] [Kicz91] are used. These techniques help eliminate minor semantic differences between similar features of the model and the language. Reflection is also used to map the features that are not present in the languages onto the languages.

Statically typed object-oriented languages are frequently used in building systems for interoperability. 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. ICOM model concentrates on statically typed object oriented languages. The languages selected are C++ [Stro94], Modula-3 [Nels90] and Java [Java95]. Though these languages fall within the same family of statically typed languages, they similar enough to yield a powerful object model, different enough to make the problem interesting, and widely enough used to create interesting opportunities for reuse (e.g., a Java applet to a C++ digital library).

In the rest of the paper, the architecture for the proposed model is discussed in section 2, use of reflection in the context of a distributed heterogeneous environment is explained in section 3, a prototype for mapping dynamic method binding feature on to the languages C++, Modula-3 and Java is discussed in section 4, work in progress is detailed in section 5, and conclusions and future directions are provided in section 6.

2 Architecture of the ICOM Framework

The architecture of the ICOM framework has three major layers: communication layer, intermediate layer, and translation layer. The communication layer (also called ICOM layer, because it is a critical component of the ICOM framework) is responsible for all communication mechanisms required between a client and a server. An entity requesting a service is called a

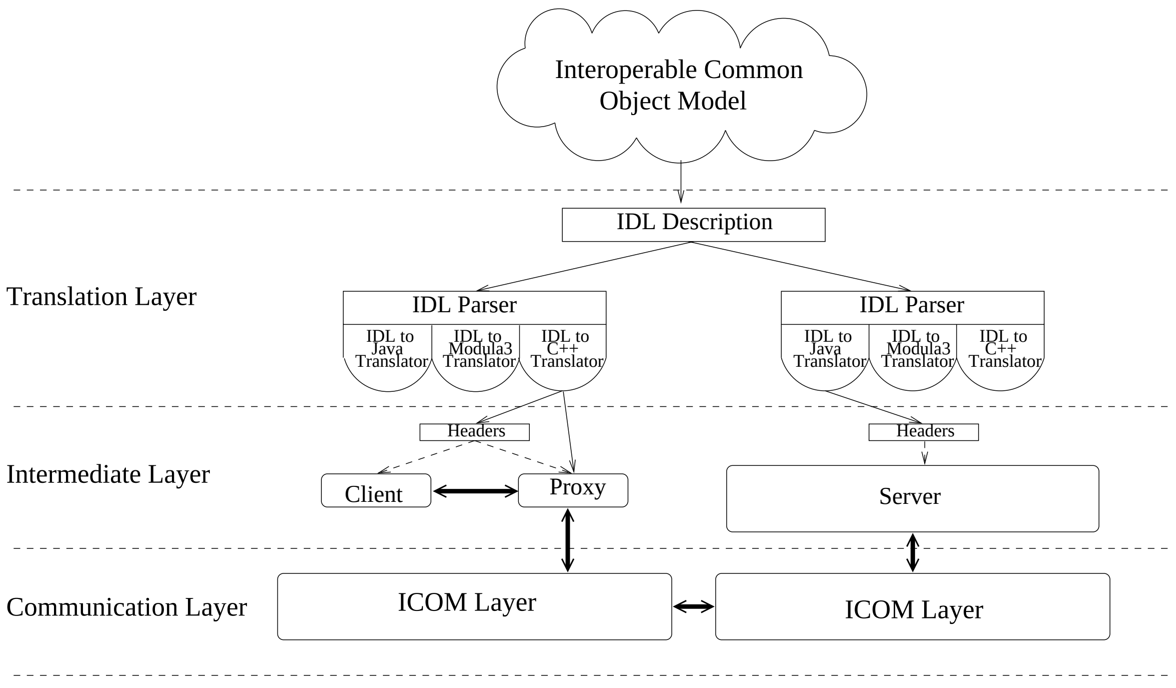

Figure 1: Architecture

client and an entity providing a service is called a server. The communication between various entities is a peer to peer communication, i.e., a server can be a client of another server. The communication between a client and a server is shown using thick and solid bi-directional arrows in Figure 1. It is the ICOM layer's job to prepare a server for communication, deliver an invocation message from a client to a server and return results. Encoding and decoding of method invocations and results is automatically handled by the ICOM layer. The translation layer contains an IDL parser and IDL to language translators. For a client to know the services offered by a server, the server's interface must be expressed in IDL. Only the features defined by the IDL's object model can be used to express the interface. The intermediate layer exists between the communication layer and the translation layer. The client code, translator generated code and the server code are present at this layer. An important aspect of ICOM framework is that the client and the server can be developed independently and linked (shown with the dotted arrows in Figure 1) to the code generated by the translators at the client and server locations respectively. The translators at the client side generate header files, and an entity called the proxy for each server object; they are shown as solid arrows from the translation layer into the intermediate layer. A proxy is a representative of the server at the client side. As far as the client is concerned the proxy acts as a real object. In reality, the proxy delegates all messages to the server via the ICOM layer and return results back to the client.

2.1 A Walk-through With An Example

A digital library example is given below to understand the functionality of the framework. Typically a server implementor advertises the services offered by the server object by providing an IDL description. An IDL description for the Library interface is given below (CORBA IDL is used to express the interface information).

```
struct Book {
   string author;
   string title;
   string call_no;
};

interface Library {
   boolean add_book(in Book book_info) raises(CapacityExceeded);
   boolean search_by_title(in string title, out Book result);
   boolean search_by_author(in string author, out Book result);
   boolean search_by_call_no(in string call_no, out Book result);
};
```

The above declaration defines a structure (Book) consisting of three string variables author, title and call_no. It also defines an interface to add books into a database and search for the database entries using different keys.

In this example a C++ client and a Modula-3 server are assumed to be present in a distributed and heterogeneous environment. The IDL translator, when invoked at the server location, parses the IDL description and generates Modula-3 header files, stubs to handle incoming messages, and a metaclass for the Library interface. The header file contains the server object's interface described in Modula-3. The stubs contain unmarshaling and marshaling routines that accept requests, interact with the server object implementation (via metaclasses) and send back results in the form of output parameters or exceptions. The metaclasses intercepts the operation invocations on the server Library objects and sends appropriate messages to the server Library objects. The server object interface is implemented by the server developer in the same language (Modula-3) as the server stubs, and linked to the IDL generated Modula-3 code.

At the client side the IDL translator generates a C++ specific "proxy object" and C++ header files. The header files contain the Library's interface rendered in C++.

The proxy objects represent the server objects at the client side. All requests to the proxy objects are marshaled, sent to the server and when received, the results are unmarshaled and sent back to the client. The client code should be parsed with the IDL translator, compiled with the language compiler and linked with the stub code generated by the initial invocation of the IDL translator.

3 Use of Reflection

Extending the definitions of reflection and reification [Maes87], object-oriented reflection and object-oriented reification with regard to object-oriented programming in a distributed and heterogeneous environment are defined. *Object-oriented reflection* is defined as the ability of an object-oriented system to make its own object model's features the subject (or domain) of its computation in a causally connected way. *Object-Oriented Reification* is the representation of the specific aspects of an object-oriented program as objects within that program.

There are two approaches to object-oriented reflection: the metaclass approach and the metaobject approach [Ferb89]. In the context of distributed and heterogeneous environment a third approach metacommunication [Ferb89] can be used. In the metaclass approach, the class of an object is considered as its metaobject. Metaclasses are metaobjects of classes. Metaclasses can have metaobjects for themselves, thus forming an infinite reflective tower [Maes87] that is normally terminated by defining a root metaclass which is metaobject of itself. If the behavior of the class needs to be changed, then the application programmer has to replace the default behavior the root metaclass by defining a specialized class from the root metaclass and assigning it to the class. In the metaobject approach, an object can have a metaobject by itself. This metaobject is an instance of a special class, typically called the "metaobject". Metaobject defines the default behavior of all objects. To change the default behavior of an object, the metaobject can be specialized and assigned to the object. In the metacommunication approach, the communication itself is reified. Each message is treated as an object. Messages are represented as instances of a special class, typically called Message, that includes a method send for transmitting the message to the receiver. It is the responsibility of a message to interpret itself. The default interpretation information is provided in the Message class which may be subclassed to provide specialized behavior for particular messages.

In the ICOM framework reflection is not used in the traditional way. Reflection techniques are used to map the common object model onto an object-oriented language. Here behavior of the model or the language is not changed. A missing feature is mapped or the semantics of a feature of the common object model is interpreted in terms of an existing corresponding feature in the language (in case of minor semantic differences between the feature of the model and the feature of a language).

As mentioned earlier, the IDL translator, when invoked at the server location, parses an IDL description and generates header files, language specific stubs, and metaclasses. The translator generates a metaclass for each class of the server. A metaclass controls the behavior of its instance, a class. Metaclasses contain method tables that have entries for the signatures of the operations supported by its instance. 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. When an invocation is received by the server, the metaclass intercepts the operation invocation and sends appropriate message to the server object.

3.1 Core Set of Features of ICOM

Among the existing common object models [Omg91] [Som93] [Cutt93] [Broc94], there is no agreement upon a common set of object-oriented features. The object model of CORBA takes a common denominator approach to find a common set of features. Since the number of features in the resulting common object model is limited, useful features that are not part of the model can not be used for interoperability. This problem is addressed by ICOM by identifying a *core set* of features and an *extended set* of features. The core set of features along with the extended set of features constitute the ICOM common object model. The ICOM framework provides a flexible mechanism that combines the existing core set of features to incorporate the extended set of features into the model.

The core set of features that must be present in a statically typed language for it to be supported in the framework are:

- object identifier - it is a unique identifier for the whole object,

- object interface - an object can have a private interface and a public interface,
- single inheritance - classes define objects state and interface. New classes can be derived by subclassing from previously existing classes through inheritance and specialization,
- object construction/destruction - an object can be constructed and destructed explicitly.
- encapsulation - it prevents direct access to the state of an object by other entities,
- method/operation/request - a service provided by an object that can be invoked by others.
- communication (synchronous) - an entity requesting a service waits for the results to arrive,
- object type (class) - a class defines the implementation shared by a group of objects, i.e., it characterizes the behavior of is instances by describing the operations that can be applied to those objects, and
- object state - it consists of a set of attributes of the object.

Almost all of the object-oriented languages support these features along with a few advanced features.

3.2 Reified Entities

Features listed here are advanced features (extended set) that can be constructed from the core set of features using reflection techniques. They are:

- message - it contains a request, possibly along with the details about a server location, client location, class name and object identifier of the server object,
- multiple inheritance - allows a class to inherit from more than one parent,
- dynamic method binding - allows binding of methods at runtime as a response to a method invocation,
- method overloading - allows more than one operation to exist with the same name in the interface of an object,
- parameterized types - types that take other types as parameters to define new types,
- message queues - allow queuing of messages intended for an object and ensure their processing,
- extensibility - ability to provide introspective reflective capabilities to the model.

- concurrency - ability to run multiple tasks at the same time; it has two aspects: a mechanism that executes a server's methods concurrently without the methods being aware of it, and a mechanism that reifies the message queueing of an object and allows the object to make decision about which message to accept.

Some of the above mentioned advanced features are supported by object-oriented languages.

3.3 Reflected Entities

The reflected entities depend on the language on to which the model is mapped. If a feature of the ICOM model is missing from the language then it gets mapped onto the language, and thus becomes a reflected feature. For example, Modula-3 does not support multiple inheritance. Since, this feature is supported by ICOM, it is reflected onto Modula-3. Here, the behavior of Modula-3 is not changed, but the feature supported by the model is mapped onto the language. All of the reified entities mentioned in the previous subsection are candidates for reflected entities.

4 Prototype

The approach followed here is an incremental prototype development. The basic ICOM framework requires implementation of the ICOM layer. Once the basic framework is in place, the features of the ICOM model can be added to the prototype one at a time. One of the important features that needs to be added first is the dynamic method binding. It is explained in the following sub-section.

4.1 Dynamic Method Binding

An experimental prototype has been developed to provide method binding between objects in a distributed, multi-language environment along with message queuing at the server objects. This prototype is motivated by the desire to provide a uniform implementation mechanism for translating the features of ICOM into statically typed object-oriented languages.

There are at least three possible ways to provide the method binding capability.

1. Modify the compiler: this is not a good solution because it does not support portability.

2. Let the programmer take care of the dynamic method binding code: this is not a good solution either, because it puts additional responsibilities on the programmer.

3. Use reflection techniques: this is a clean and elegant method because it does not have the drawbacks of previous two methods.

The prototype follows a reflection technique based on the combination of the metaclass approach and the metacommunication approach. The metaclasses control the behavior of the server objects whereas the messages between the interacting objects are reified as message objects which carry the necessary information required for communication.

The prototype is different from the existing systems in the following ways. Programming conventions such as using directives in comments [Chib93] are not required. The prototype uses a combination of metaclass and metacommunication approaches whereas the existing systems use either metaclass or metaobject approaches. The prototype is intended to support the idea of reifying most of the object-oriented features as opposed to the existing systems where only a few object-oriented features are reified.

The prototype has been implemented in C++, Java, and Modula-3. A simplified diagram of the system architecture is shown in Figure 2 to explain the prototype.

An actor-like entity is used to provide the queuing facility at the server side. Queuing is an essential feature to have in applications that involve a large number of messages being sent to the server. Actor-like entities wrap around the server objects and act as message handlers. They receive the messages intended for the server objects and deliver them one at a time.

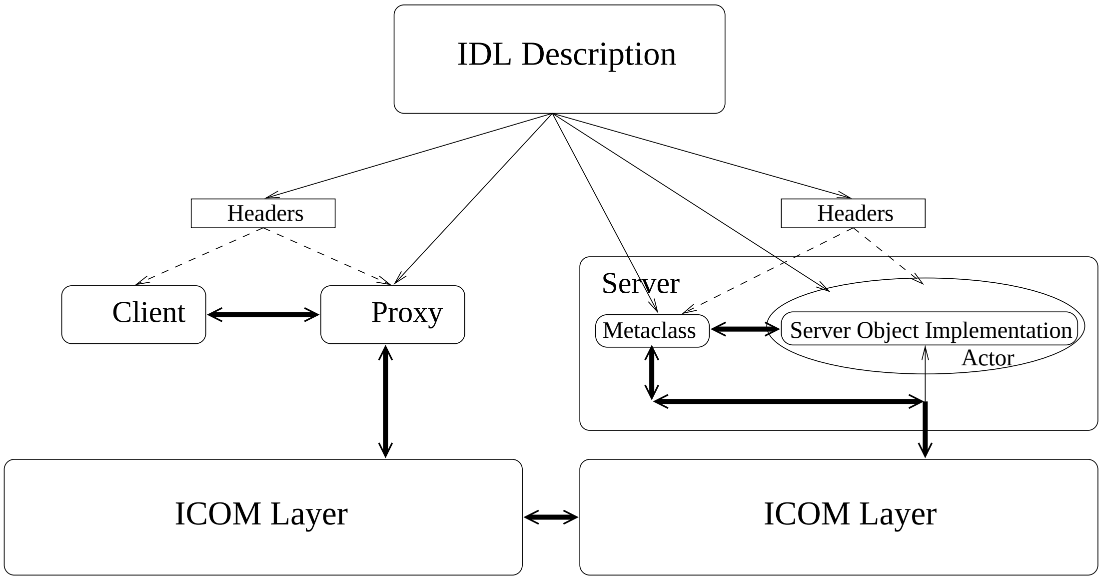

Figure 2: Prototype

The client side primarily consists of headers and a proxy object implementation. The headers contain basic actor element definitions (e.g., actor, message, behavior etc.) and the interface of the proxy object which is a representative (of the server object) at the client side. The proxy object's interface is the same as that of the server object, but its implementation is different. Each operation of the interface is represented as a class that has *encode* and *send* operations in its interface. Any invocation of an operation X_{op} of the proxy object results in the invocation of the three operations: new(X_{obj}), X_{obj}*.encode()*, and X_{obj}*.send()*. The execution of *new* operation creates an instance X_{obj} of the class X_{class} which is defined in the interface of the proxy object. The execution of *encode* operation translates the signature of X_{op} into a standard form understandable by ICOM. The execution of *send* operation transports the encoded operation (also referred to as reified message) to the ICOM at the server location.

The communication between the client and the server is shown using thick, solid bi-directional arrows in Figure 2. The translator generated components are connected to the IDL description with solid arrows. The linking of translator generated components with the user/developer code is shown with dotted arrows.

The server side consists of headers defining the basic actor elements, a metaclass and a server object template[1]. A metaclass contains a method table to hold the signatures of operations of the server object and pointers to their implementations. The Metaclass' interface has two operations *add* and *bind*. The *add* operation enters into the method table a signature and a pointer to the implementation of the corresponding operation. The *bind* operation searches for a signature and a corresponding pointer to the implementation and initiates its execution. The server object template defines a generic server object interface that can be instantiated with an actual server object's class. This interface consists of classes representing the operations of the server object. A *new* operation on the instantiated template creates a method table and registers the signature of each operation of the server object interface along with the pointer to the implementation of that operation.

To invoke an operation on a server object, a client invokes the operation on the proxy object. The proxy object creates an instance of the class representing that operation (with appropriate initial values) and invokes *encode* and *send* operations on it.

At the server side the ICOM layer interprets the message and creates a new message object that holds the signature and the values and supports *invoke* operation in its interface. ICOM performs the *invoke* operation on the new object. The execution of the *invoke* operation results in the invocation of a relevant operation on the server object. The metaclass of the server intercepts this invocation and executes appropriate methods.

A digital library is used as an example in this prototype. The Library interface supports: *add_book*, *search_by_title*, *search_by_author*, and *search_by_call_no* operations. The add_book operation is used to add book records that have author, title and call number information in them and the remaining operations are used to search books based on the author name, book title, and call number. When an *add_book* invocation is received by the server, it is delegated

[1]The server object template is mapped into the languages using the existing features of each one of them. C++ directly supports templates, Modula-3 supports generics, and Java supports interfaces that can be used to emulate templates

to the metaclass. The metaclass performs the *lookup* and *bind* operations for the message. An actor-like entity gets created as a result of the binding of *create* message. The actor-like entity can receive messages and execute the methods of the Library object. The server then receives the other messages, and delegates them to the metaclass. The server object implementation receives the messages through the actor-like entity wrapped around it, after lookup and bind operations at the metaclass and executes appropriate methods.

5 Work in Progress

5.1 Inheritance

A prototype enhancement is underway to map single inheritance feature onto the object-oriented languages C++, Modula-3, and Java, in a distributed and heterogeneous environment. An important issue is the location of the state of the parent portion of an instance of a child class. For simplicity it can be kept at the parent's location. When invoked, the IDL translator at the client cite generates a metaclass for the child class, based on the inheritance hierarchy. This metaclass maintains a method table that contains the method signatures defined in the child class. It delegates method calls whose methods are not defined in the child class, to the parent class location. These method calls are handled by the metaclass of the parent class.

5.2 Multiple Inheritance

The single inheritance feature can be extended to map the multiple inheritance feature onto the selected languages. The process is straight forward for languages that support multiple inheritance. Multiple inheritance is faked in languages that do not support it, using metaclasses. The lookup method of a metaclass delegates (after failing to find a method entry in its method table) a method call to the metaclasses of the parents successively until a method entry is found in their method tables. A child class and its parent classes are loosely connected using metaclasses to map the multiple inheritance feature.

5.3 Method Overloading

Method overloading involves overloading of methods within a class and overloading of inherited methods in a class. Mapping of method overloading feature is straight forward in languages that support method overloading. The multiple inheritance feature will be used to map the method overloading feature onto the languages that do not support it.

5.4 Parameterized Types

Parameterized types introduce an interesting idea of "distributed compilation". Assume three locations A, B, and C in a distributed heterogeneous environment, Let A have a parameterized type, A_Type, its interface definition and its implementation. Let B have a class (B_Class) definition and its implementation. Let C have an interface declaration, A_Type(B_Class), that instantiates the parameterized type at A with the class defined at B. At the time of IDL translation, the translator at C, invokes the IDL translator at A, by sending the interface description of B_Class. The translator at A instantiates the parameterized type, A_Type, with the interface of B_Class. Proxies will be generated at C that delegate method invocations to A which in turn delegates them to B when needed.

The initial prototype that supports dynamic method binding will be extended to include support for the above mentioned features.

6 Conclusions and Future Work

In this paper a powerful common object model for interoperability of object-oriented languages was introduced. A framework was designed that supports this model in statically typed object-oriented languages. Reflection techniques were used to provide a uniform translation of the model into the participating languages. Prototype explaining the basic framework that supports dynamic method binding was explained. Actor-like entities were used to provide message queues. Currently work is in progress to enhance the existing prototype to include inheritance, multiple inheritance, method overloading, and parameterized types. Inclusion of parameterized involves a new concept called "distributed compilation" that needs to be explored further. Once the real framework is completed, a real world application that uses software components written in C++, Modula-3, and Java will be developed.

Acknowledgements

The authors would like to thank the group members of Act++ and Distributed Act++ groups for their help. They would like to acknowledge the Department of Computer Science at Virginia Tech for providing the computing facilities. They would also like to thank Gregor Kiczales and Satoshi Matsuoka for their encouragement.

References

[Agha86] Gul A. Agha, *ACTORS: A Model of Concurrent Computation in Distributed Systems*, The MIT Press, Cambridge, Massachusetts, 1986.

[Atta89] Guiseppe Attardi, Cinzia Bonini, Maria Rosaria Boscotrecase, Tito Flagella, and Mauro Gaspari, Metalevel Programming in CLOS, *ECOOP 89*, July, 1989, pp 243-256.

[Broc94] K. Brockschmidt, *Inside OLE 2*, Microsoft Press, Redmond, 1994.

[Chal94] Siva Challa, Towards an Interoperable, Reflective Common Object Model for Statically Typed Object-Oriented Languages, *OOPSLA '94 Workshop on Multi-Language Object Models*, Portland, Oregon, October 24, 1994.

[Chib93] Shigeru Chiba, *Open C++ Release 1.2 Programmer's Guide*, Technical Report no. 93-3, University of Tokyo, 1993.

[Cutt93] Doug Cutting, Bill Janssen, Mike Spreitzer and Farrel Wymore, *ILU Reference Manual*, Xerox Corporation, Palo Alto, 1993.

[Ferb89] Jacques Ferber, Computational Reflection in Class Based Object Oriented Languages, *OOPSLA '89 Proceedings*, October, 1989, pp 317-326.

[Java95] *The Java Language Specification*, Sun Microsystems, CA, 1995.

[Kafu92] Dennis Kafura and Manibrata Mukherji and Greg Lavender, ACT++: A Class Library for Concurrent Programming in C++ Using Actors, *Journal of Object-Oriented Programming*, October, 1993, pp 47-62.

[Kicz91] Gregor Kiczales, James des Rivieres, and Ganiel G Bobrow, *The Art of the Metaobject Protocol*, MIT Press, Cambridge, MA, 1991.

[Kons93] Dimitri Konstantas, Object Oriented Interoperability, *Visual Objects*, D. Tsichritzis (Ed.), University of Geneva, 1993.

[Malh93] Jawahar Malhotra, Dynamic Extensibility in a Statically-Compiled Object-Oriented Language, *Object Technologies for Advanced Software*, November, 1993, pp 297-314.

[Mano93] Frank Manola and Sandra Heiler, *A "RISC" Object Model for Object System Interoperation: Concepts and Appl ications*, Document No. TR-0231-08-93-165, GTE Laboratories Inc., Waltham, Massachusetts, December, 1993.

[Mano94] X3H7 Object Model Features Matrix, Frank Manola (Editor), x3h7-93-007v8, Waltham, Massachusetts, May, 1994.

[Maes87] Pattie Maes, Concepts and Experiments in Computational Reflection, *OOPSLA '87 Proceedings*, October, 1987, pp 147-155.

[Nels90] Greg Nelson (Editor), *Systems Programming with Modula-3*, Prentice Hall, Englewood Cliffs, NJ, 1991.

[Omg91] Object Management Group, *The Common Object Request Broker: Architecture and Specification* OMG Document Number 91.12.1, Rev. 1.1, 1991.

[Open93] *Reference Model for Open Distributed Processing*, WG7 N. 885, International Standards Organization, New South Wales, Australia, November, 1993.

[Som93] *SOMobjects Developer Toolkit Technical Overview*, Version 2.0, IBM, New York, November, 1993.

[Stro94] Bjarne Stroustrup, *The C++ Programming Language*, Second Edition, Addison Wesley Publishing Co., Reading, MA, 1994.