Filter Objects for Distributed Object Systems

G. Sriram Reddy, Rushikesh K. Joshi [1]
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Powai, Mumbai-400076, India

Abstract

In this paper, we describe a design and implementation of *filter objects* for CORBA based distributed object systems. Filter objects are intermediate objects which transparently intercept client requests. Filter objects can intercept both *upward* messages from clients to servers and *downward* messages that are return results to clients. We introduce filters in CORBA systems through a standard IDL interface called `Filterable`. A filterable server interface can be built using this standard interface via inheritance. Implementation of a filterable server inherits a library implementation of interface `Filterable`. Filter objects are full fledged CORBA objects which can be plugged and unplugged from their corresponding servers at run time. Filter objects have been implemented as an add-on facility for the MICO C++ Corba implementation working on LINUX operating system.

Keywords: CORBA, distributed objects, extensions to objects, filtered messages, filter objects, design and implementation

1 Introduction

In the conventional distributed object oriented systems, messages from sources are directly delivered to their destinations. We describe in this paper a filtered delivery model for CORBA [5] based distributed systems in which intermediate filter objects are allowed to transparently intercept messages. This property of transparency gives filter objects the ability to carry out additional tasks with the messages when they are on their way to destination and back without changing the client or the server code. Some examples of the tasks that can be carried out by filters are transparent security checks, routing of requests, load balancing of replicated servers, maintaining server side caches, masking of server functionalities and providing replacement code for servers.

The filter objects that we propose in this paper are based on the model of message filters of Joshi et al. [4]. Other interesting models of message filters that have been described in the literature are the composition filters of Aksit et al. [1] and the per-process and per-object filters in IONA's Corba product, Orbix [3].

Filter objects can manipulate the message arguments, can perform intermediate actions and either pass the request on to the destination or bounce it to the client on behalf of the server with a return result. Filter objects can be dynamically created and plugged to sever objects. They can filter requests in the upward direction and their

[1] *Corresponding Author. Email:rkj@cse.iitb.ernet.in*

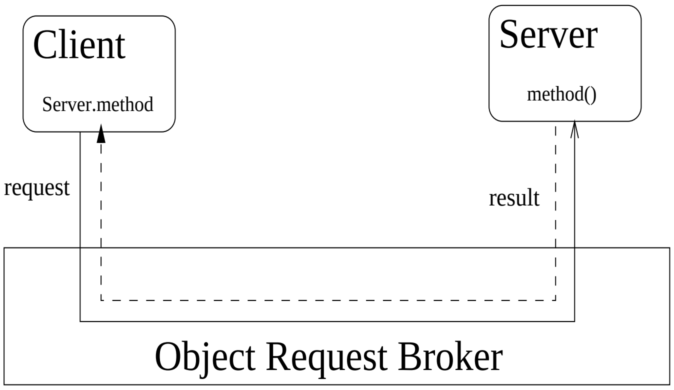

Figure 1: Distributed direct delivery model

return results in the downward direction. The distributed filter object model that we describe in this paper is realized through an IDL interface called `Filterable`. The *Filterable* interface contains member functions `attach()` and `detach()`, which are used to attach and detach filter objects with their server objects at run time. A server object makes itself filterable at the IDL level by inheriting the *Filterable* IDL. An implementation of the server IDL inherits from its library implementation called *Filterable_impl*. The IDL compilation phase generates a *filter base interface*, an IDL, through which the filter objects are implemented.

The implementation of filter objects is built over an existing IDL compiler by adding a pre-translation and a post-translation stage. A pre-translator generates a *filter base interface* for a filterable server interface and a *post-translator* modifies the dispatcher in the server skeleton. The generated *filter base interface* defines filter methods that are invoked transparently during interception of their corresponding methods in the server object. This interface is split into *upfilter* and *downfilter* interfaces. The methods of the *upfilter* interface can intercept the upward messages to the server, whereas the *downfilter* methods can intercept the return values from the server. Filters objects derive their IDL interfaces from the *filter base interface*.

The paper is organized as follows. Section 2 distinguishes the distributed filtered delivery model from the conventional direct delivery model. In Section 3, the methodology of design and development of filterable servers and filter objects is illustrated. The implementation of filter object support for the MICO C++ CORBA implementation [6] is described in Section 4.

2 Distributed Filtered Messaging

Figure 1 shows the conventional distributed direct delivery message passing model of CORBA. A *client* sends a request to a remote *server* which gets directly delivered to the server that is chosen by the client itself. The corresponding method is invoked at the remote destination object.

In the *distributed filtered delivery model* as shown in Figure 2, the intermediate transparent filter object intercepts a method invocation while client still sees only the server object. Design patterns such as *Proxy* and *Adapter* [2] also employ intermediate objects.

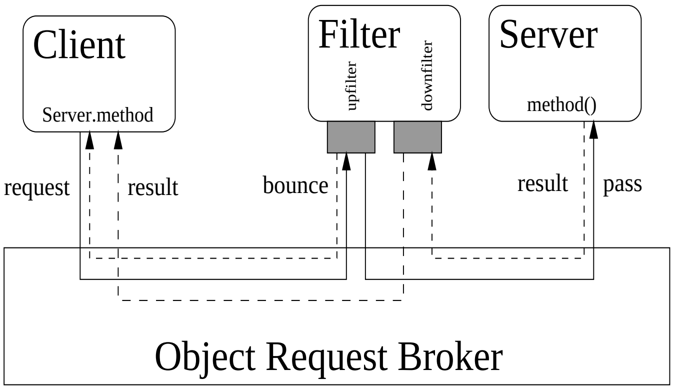

Figure 2: Distributed filtered delivery model

However, the intermediate objects in these patterns are known to the client and the client makes explicit invocations on them. In the case of a filter object, the client code is unaware of its existence.

A filter object is a full fledged CORBA object, and in addition to its properties of being a distributed object, it also possesses the transparent filtering capabilities. A filter object thus may export an interface and make itself available to other objects. A filter object may be located at any node irrespective of the location of the message source and message destination objects.

A filter object performs bidirectional filtering. Upward messages from `Client` to `Server` are intercepted by an *upfilter* interface and downward return results are intercepted by a *downfilter* interface. The IDL-translator generates a filter IDL that consists of its upfilter and downfilter member functions. Implementation of the filter interface is provided by the filter programmer. An instance of the filter implementation can be plugged to the corresponding server during runtime to activate filtering actions. A filter implementation can manipulate the arguments of the message that it intercepts and decide either to forward the request to the actual destination through a *pass* action or to bounce the results to the client through a *bounce* action.

3 Design Methodology

The design methodology for filter objects is based on the *need-to-filter* principle. A server object which anticipates or is aware of a need for filtering is specified as a filterable server. A filterable server is thus a *filter-aware* object. A filter object cannot be plugged to a server that is not filter-aware. Filter-awareness is granted to servers through a standard IDL interface during interface definition stage and through an implementation of this IDL during the server implementation stage.

Similarly, a filter object has to be server specific. However, the implementation of filter objects is fully decoupled from the implementation of its corresponding server object. Once a server is specified as a filter-aware server during the interface definition stage, its corresponding IDL that functions as a filter base interface is generated. This filter base interface is used subsequently during the filter implementation.

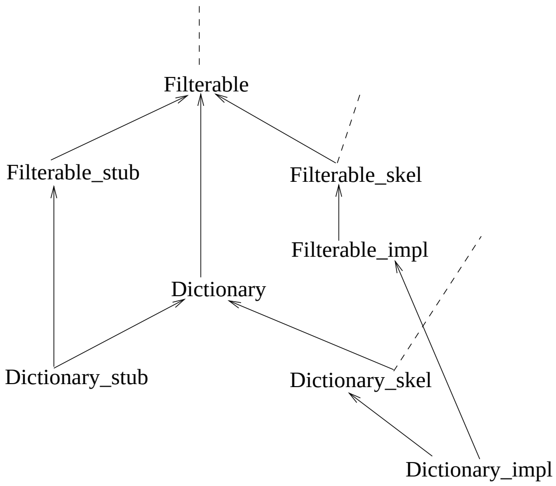

Figure 3: Class hierarchy for filterable server Dictionary.

3.1 Filterable Servers

A standard IDL interface `Filterable` contains operations *attach* and *detach* which are common to all filterable servers. Class `Filterable_impl` which is a library implementation of IDL `Filterable` implements the two member functions *attach* and *detach*. A server IDL is made filterable by inheriting class `Filterable`. As an example, Figure 3 shows the class hierarchy for a filterable server IDL *Dictionary*. The hierarchy depicts a view of the filterable-server implementor. The code below shows a filterable server `Dictionary`, which is made filterable through inheritance.

```
interface Filterable {
    void attach (in Object filter);
    void detach ();
};
interface Dictionary: Filterable {
    string searchWord (in string Word);
    void addWord (in string Word, in string Meaning);
};
```

The server implementation `Dictionary_impl` inherits `Filterable_impl`. The user needs only to implement server methods since both the methods `attach()` and `detach()` are inherited from `Filterable_impl`. Operations *attach* and *detach* can be called on an instance of *Filterable_imple* to plug or unplug a filter object during runtime. The following code shows the method of implementing the filterable server object in this example.

```
class Filterable_impl : virtual public Filterable_skel {
    // library implementations of attach() and detach()
};
class Dictionary_impl : virtual public Filterable_impl, virtual public Dictionary_skel {
    // filter implementation of searchWord() and addWord()
};
```

3.2 Filter Objects

A filterable server IDL is used to generate its corresponding filter base IDL. A filter base IDL consists of filter member functions. A filter object is implemented by providing implementations of these filter member functions. It is possible to inherit the filter base IDL to provide additional conventional member functions.

3.2.1 Filter Base Interface

Filter base interface defines filter methods that are invoked automatically when their corresponding methods in the server are invoked by clients. Consider the following example of an IDL `Dictionary`, which is a filterable server interface:

```
#include "Filterable.idl"
interface Dictionary : Filterable {
    string searchWord (in string Word);
    void addWord (in string Word, in string Meaning);
};
```

Its corresponding filter base interface called `Dictionary_Filter` is generated by the IDL-translator as follows:

```
interface Dictionary_Filter {
    string searchWord_Up (inout string Word, out int message_passed);
    string searchWord_Down (inout string Meaning);
    void addWord_Up (inout string Word, inout string Meaning, out int message_passed);
};
```

A filter base interface is split into upfilter and downfilter interfaces. The methods of the upfilter interface can intercept the upward messages going towards the server, whereas the downfilter methods can intercept the return values from the server. A downfilter member is not generated for for the methods that return a *void* value. For example, the methods `searchWord_Up()` and `searchWord_Down()` in IDL `Dictionary_Filter` are the upfilter and downfilter methods for the method `searchWord()` as defined in IDL `Dictionary`.

The types of the input and return arguments in an upfilter member match with its corresponding argument types in the server method. An upfilter member also requires an additional output argument to specify whether the filter member function has taken a *pass* action or a *bounce* action.

A downfilter method receives the return value from the server and has to return a value of the same type to the source of the message. Hence the type of the input and return argument in a downfilter method is the same as that of the return argument in the server member function. Note that each input argument of every method in *filter base interface* is specified as a read-write argument by the access right specifier **inout**. This gives filter objects the capability to change the argument values before passing them on to the server.

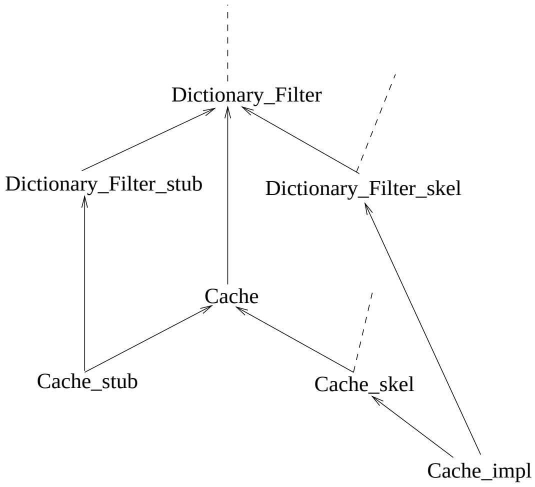

Figure 4: Class hierarchy for filter Cache.

3.2.2 Implementing Filter Objects

Filter objects can be implemented in two ways. In the first alternative, the user can directly implement the *filter base interface* generated by the *IDL-Translator*. The second alternative derives a filter interface from a *filter base interface*. In this case, the filter implementor has the flexibility to add new public member functions to the filter object. Using the second scheme, the implementor can also give a convenient name to the filter interface.

The example below shows an implementation of a filter interface `Cache` through inheritance of the filter base interface of filterable server `Dictionary`.

```
interface Dictionary_Filter {
    string searchWord_Up (inout string Word, out int message_passed);
    string searchWord_Down (inout string Meaning);
    void addWord_Up (inout string Word, inout string Meaning, out int message_passed);
};
interface Cache : Dictionary_Filter { ... };
```

Figure 4 shows the class hierarchy for filter implementation, a view of the filter implementor. The implementation of the filter IDL `Cache` is derived from class `Dictionary_Filter_skel` in the usual way of implementing CORBA IDLs. The programmer needs to implement all the methods of `Dictionary_Filter`. The code for the implementation of `Cache` is shown below.

```
class Cache_impl : virtual public Dictionary_Filter_skel, virtual public Cache_skel {
private: // cache implementation
public:
        ...
    string searchWord_Up (inout string Word, out int message_passed)
    { // return from cache if hit }
```

```
    string searchWord_Down (inout string Meaning)
    { // update cache with new result if miss}

    void addWord_Up (inout string Word, inout string Meaning, out int message_passed)
    { // update cache }
};
```

In this example, the filter object provides a cache implementation as its state. The upfilter and downfilter members access the local state of the filter object. The `searchWord()` invocation on the server is intercepted by the `searchWord_Up()` in the cache object. It can specify a bounce action upon cache hit. The downfilter member is used to carry out a cache update.

3.3 Attaching Filter Objects to Servers

A filter object can be attached to a filterable server object by calling an invocation of *attach()* on the server object as shown below.

```
int main( int argc, char *argv[] ) {
    ...
    _server- >attach(filter_obj);
    ...
}
```

This code runs as a client for the filterable server. It plugs `filter_obj` to `_server` object in filter relationship. The filter object can be detached similarly by invoking `detach()` on the filterable server. A client code such as the above can be treated as a code running in the system administrator's console. A system administrator may thus plug and unplug desired filtered objects to their corresponding servers at runtime.

3.4 Security

It is possible to integrate a security mechanism in the design presented. In the current implementation of filter objects, a password based security mechanism has been provided for attach and detach operations. These operations can only be performed by authorized users. The server sets the *filter password* which has to be used subsequently to attach or detach a filter object to the server object. Similar mechanism can be employed for filter member function invocations.

4 Implementation

We have implemented filter objects for the MICO C++ Corba implementation [6] as an add-on facility. In this section, we discuss the implementation mechanism in detail.

An *IDL compiler* generates client stubs, server skeletons and other code from a CORBA IDL as per the language mappings. A pre-translator written in Lex and Yacc and a post-translator written is C++ generate the necessary IDLs and dispatcher implementation. The implementation scheme is shown in Figure 5.

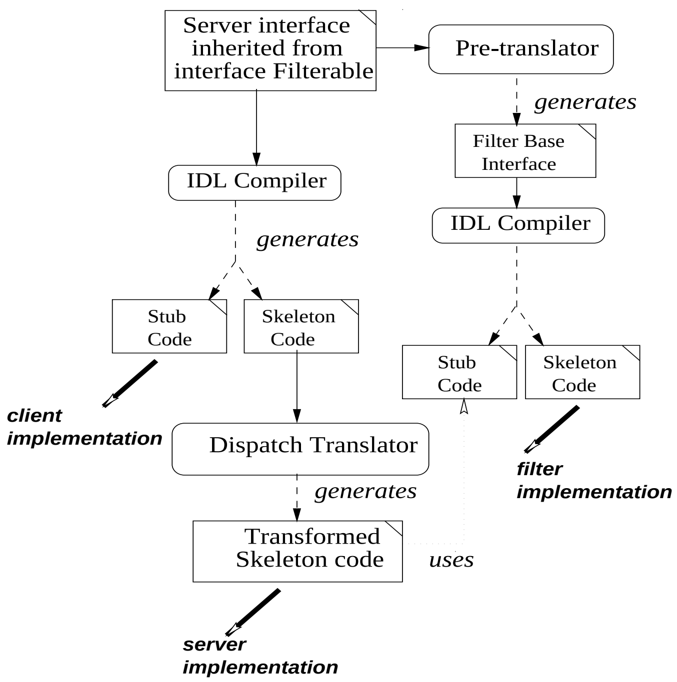

Figure 5: The Implementation Scheme

4.1 Pre-translation stage for IDLs

The pre-translator takes a filterable server IDL as its input and generates the corresponding *filter base interface*. The necessary upfilter and downfilter members with appropriate type specifications are generated. The generated filter base interface can subsequently be compiled to obtain the filter skeleton and the filter stub as shown in the figure.

4.2 Post-translation stage for Dispatcher

A *dispatch* function in the skeleton class is responsible for method dispatch. For example, in the following code, the dispatcher schedules the invocation of `searchWord()` method in the Dictionary_skel class.

```
bool Dictionary_skel::dispatch( CORBA::ServerRequest_ptr _req,
                                CORBA::Environment &_env ) {
  if( strcmp( _req->op_name(), "searchWord" ) == 0 ) {
    ...
    try { _res = searchWord( Word ); } ...
    return true;
  } ...
}
```

The dispatcher as generated by the IDL-complier directly invokes the methods of the server. An add-on post-translator stage injects the required *filtering code* into the dispatch function to enable filtering. A snapshot of the transformed dispatch function for the above example is shown below.

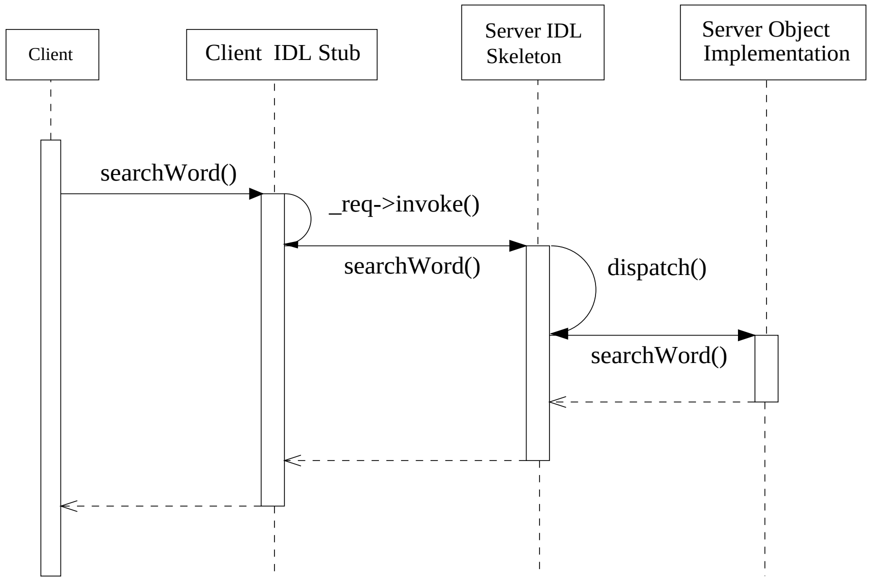

Figure 6: Method Invocation on a Filterable Server without a Filter Object

```
bool Dictionary_skel::dispatch( CORBA::ServerRequest_ptr _req,
                                CORBA::Environment &_env ) {
  if( strcmp( _req->op_name(), "searchWord" ) == 0 ) {
    ...
    try {
      if(filter_tag) {  // set by ‘attach’ method.
       Dictionary_Filter_ptr _fil = Dictionary_Filter::_narrow( _filter );
          _res = _fil->searchWord_Up( Word, message\_passed );  // upfilter.
          if(Message\_Passed()) {   // checks the value set by upfilter.
             _res = searchWord( Word );  //  pass to server.
             _res = _fil->searchWord_Down( _res );  // downfilter.
          }
      }
      else  // filter is not attached.
         _res = searchWord( Word );  // directly invokes server method.
     }
     ...
}
```

An *Object* pointer `_filter` maintains a pointer to the currently attached filter object and a *Boolean* variable `filter_tag` specifies the plug/unplug status of the filter object during runtime. These variables are set and reset by `attach()` and `detach()` members. If the attribute `message_passed` is observed to be true after interception, the message is dispatched into the server.

4.3 Interaction Diagrams

The interaction diagrams for method invocations on filterable server objects are presented in Figures 6 and 7. The first scenario in Figure 6 represents an invocation of

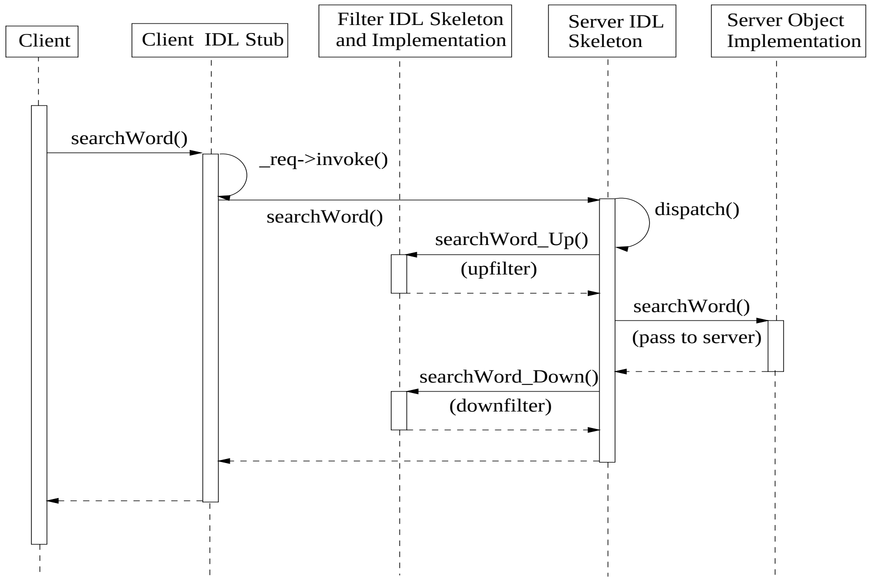

Figure 7: Method Invocation on a Filterable Server with Filter Object

method *searchWord()* on a filterable server object without a filter object or with an unplugged filter object. The second scenario as shown in Figure 7 represents an invocation that is intercepted by a filter object.

It can be seen that this implementation incurs the *runtime overheads* of redirection of requests to filter objects from server objects. If a support for filtering is built into the ORB, these overheads can be reduced.

4.4 Implementation of Attach and Detach

The `attach()` and `detach()` methods in class `Filterable_impl` modify a filter object pointer and a plug/unplug status flag. For example, the implementation of the attach operation is provided below.

```
class Filterable_impl : virtual public Filterable_skel {
public:
    . . .
    void attach (CORBA::Object_ptr Filter) {
        // check for authorized user.
        . . .
        _filter = CORBA::Object::_duplicate( Filter );
        filter_tag = TRUE;
        . . .
    }
    ...
};
```

Similarly, the detach operation sets the `_filter` object to a NULL value and resets the variable `filter_tag`. It can be seen that the filter pointer is an *Object* pointer.

The dispatcher *narrows* the pointer in order to invoke filter methods on an instance of a particular filter implementation class. This scheme was chosen since attach and detach methods are implemented as a library implementation, whereas the dispatcher is generated for a particular filterable server.

5 Conclusions

We presented the design and implementation of a filter object mechanism for CORBA based distributed object systems based on an existing model of first class filter object for C++. A filter object can be plugged during runtime to transparently intercept message invocations on a CORBA server object. A distributed filter object is a full fledged CORBA object. Filterable servers are developed based on the *need-to-filter* principle. A filterable server is specified at the IDL level through inheritance of a standard interface called `Filterable` and is implemented by inheriting a standard implementation called `Filterable_impl`. A filter object for a filterable server is implemented by implementing the filter base interface for the intended server. This scheme was implemented as an add-on facility for the MICO C++ CORBA implementation running on LINUX.

References

[1] Mehmet Aksit, K. Watika, J. Bosch, L. Bergmans, A. Yonezawa, *Abstracting Object Interactions Using Composition Filters*, Proceedings of ECOOP'93, pp. 152-184, LNCS-791, Springer Verlag, 1993.

[2] Erich Gamma, Richard Helm, Ralph Josnson, John Vlissides, *Design Patterns*, Addison-Wesley, 1995.

[3] IONA Technologies Ltd., *Orbix Advanced Programmer's Guide*, 1991.

[4] Rushikesh K. Joshi, N. Vivekananda and D. Janaki Ram, *Message Filters for Object-Oriented Systems.* Software-Practice and Experience, Vol.27(6), June 1997, pp.677-699.

[5] Object Management Group, *CORBA Specifications*, http://www.omg.org

[6] Kay Romer and Arno Puder, *MICO Reference Manual,* Computer Science Department, University of Frankfurt, Germany.