Towards A Generic Object Model For Multi-Domain Management

Bela Ban, IBM Zurich Research Laboratory[1], University of Zurich

Abstract

Today, management of heterogeneous distributed networks is a non-trivial task requiring sophisticated management tools. In the telecommunications area SNMP [7] and X.700 [3] are predominantly used while distributed object-oriented programming toolkits such as CORBA [6] or TINA [8] are emerging as alternatives. Therefore it would be beneficial to have a tool that lets programmers transparently access instances from several different object models. This paper presents a uniform generic object model (GOM) [2] that can be employed to transparently manipulate instances from various specific object models.

1 Introduction

This paper proposes a management framework that is based on a uniform generic object model (GOM) that can be used to transparently manipulate instances of various specific object models. The object model consists of classes, operations, attributes, values and a type hierarchy which are modeled using the C++ language as host system. By using and manipulating only instances of these classes, a uniform and transparent interface is offered which allows to disregard the specific underlying object model(s) used and the idiosyncracies they feature. In addition, the generic object model has a Meta Information Database (MID) which maintains information about the classes, types etc. used in the specific object models. Using the MID, the generic object model is able to offer a highly dynamic, runtime type-checked interface that is very flexible and suited to manage information distributed over several, disparate locations.

2 The Generic Object Model

1. IBM Research Division, Zurich Research Laboratory, Säumerstrasse 4, 8803 Rüschlikon, Switzerland. Email: bba@zurich.ibm.com, WWW: http://www.zurich.ibm.com/~bba/.

2.1 Introduction

The generic object model (GOM) is our approach to multi-domain management. Among the features it offers are an object-oriented programming model, transparent access to instances of other object models, meta-information and a flexible, runtime-based typing and method binding. The architecture of GOM is shown in figure 1 below:

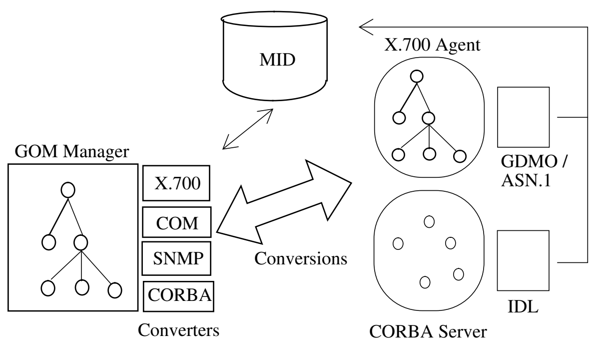

Figure 1: Architecture of GOM

The approach taken by GOM is to collect meta-information from the specification of object models such as X.700's GDMO/ASN.1 [3] or CORBA's IDL [6] in a Meta-Information Database (MID) and access the instances in an X.700 agent or a CORBA server using this meta-information. This makes the client independent from the server since the contract[1] offered by a class is retrieved dynamically at runtime from the MID rather than at compile-time from statically included header files.

A GOM manager creates a proxy instance in its address space for each remote instance in a CORBA server or X.700 agent. Requests sent to the proxy are forwarded to the remote instance using customizable converters that make use of meta-information from the MID for their task. A converter translates information between GOM and a specific object model. In the case of the CORBA converter, GOM information is first converted to CORBA in the form of a DII request. The result is then converted back to GOM[2].

GOM instances are manipulated through the interface offered by an abstract base class. X.700 or CORBA instances are created from classes derived from the abstract base class. Using the virtual method dispatch mechanism of C++, the user need not know whether X.700 or CORBA instances are actually accessed.

The information in the MID is created by GDMO/ASN.1 and IDL compilers, the backend of which was modified.

1. A *service* is an operation offered by a class. The set of services offered by a class comprise the contract.
2. In the case of X.700, a CMIP PDU is generated and sent to the agent.

2.2 Object Model

In the generic object model, each class, be it a CORBA interface or an X.700 managed object class, is represented by either the C++ class *CorbaObj* or *X700_Obj*. The abstract interface through which they can be accessed is defined in the abstract base class *GenObj*. Each GOM class has a list of attributes and operations, which are instances of the classes *Operation* and *Attribute* respectively. An attribute has a type and a value, represented through the classes *Type* and *Val*. The type essentially contains meta-information about the attribute, which was retrieved from the MID when the instance was created. The value is an instance of one of the subclasses of *Val* that are provided by GOM. There are simple values such as *Boolean*, *Long*, *String* and *Integer* and aggregate values such as *Struct*, *Sequence* and *Union*. There is a finite number of GOM values available for use by the programmer.

2.3 Meta Information

In distributed applications, processing is usually divided among many disparate instances, distributed across the entire network. Each instance offers one of more services and many services may even be offered by more than one instance, e.g. for redundancy purposes. Instances (and therefore services) may be added or removed from the network any time without disrupting the operation of the applications. If the contract between a client and server is based on header files that are included by the client at compile time, then it is impossible for the client to use new or modified services without recompilation. In contrast, if a contract between client and server is based on the high-level description of the services (e.g. IDL or GDMO/ASN.1) and if the binding between client and server is highly dynamic as in GOM, then it is possible to add or modify services without disruption of the application. Consider e.g. the case where a new service (i.e. operation) is added to a class. Using a static approach, the client would have to be recompiled even if it did not use the new service because the headers files are regenerated. Using the dynamic approach, the client could continue using the old services and would only have to be recreated if it wanted to use the new service explicitely. If the client was programmed to use the type information available for operations, it would not even need to be recompiled. This may be the case with a graphical user interface where, on a mouse-click, all attributes and operations of a GOM instance are listed. Upon a further click on an operation, all parameters would be shown and could be filled in and the operation could be performed.

The MID essentially keeps all descriptions of either IDL interfaces or GDMO templates containing all their attributes and operations in a database. It is used to query e.g. the type of an attribute or a description of an IDL interface. The information in the database is initially created by IDL and/or GDMO/ASN.1 compilers that parse specifications and provide input to the MID. Any modification or addition of IDL or GDMO/ASN.1 files would have to be fed into the MID as well. Once information in the MID has been updated/added, it is usable immediately.

2.4 Converters

Converters have the task of adapting ‘foreign’ object models to the generic object model. This involves the conversion of values sent to the foreign model and the conversion of results received from the other model. A converter must be a subclass of GenObj and has to implement the parent class protocol dictated by GenObj’s pure virtual methods. In addition it can be used to contain object model specific information such as the distinguished name and the managed object class OID in the case of X.700. Converters are implemented using shared libraries and demand-loaded at runtime. Thus applications employing GOM only have to pay for the converters they really use.

3 Conclusion

Using GOM offers a significant amount of flexibility to programmers. First, the binding between client and server is not tight[1]; servers may be modified without the client having to be recompiled or even shut down. Modeling attributes and their values as separate C++ classes offers additional flexibility with respect to X.700 conditional packages, extensible attribute groups and the ASN.1 ANY DEFINED BY type. Providing this flexibility increases the chance that future object models may be integrated with GOM as well. A requirement of an object model to be included is the provision of a dynamic API such as CORBA’s DII or the CMIP protocol. To integrate an object model with GOM, only a compiler that adds type information from the model’s specification files to the MID and a converter from the specific model to GOM and vice versa have to be written.

The second advantage is the uniform and simple API offered by GOM and the fact that the host language is C++, a language widely known and used. The learning effort for GOM is not long, and once, the syntax and semantics are grasped, instances from any specific object model for which a GOM converter exists, can be manipulated. The integration of yet another model into GOM would not require a programmer to acquire a new syntax or semantics since GOM can be used to manipulate instances of the new model.

GOM currently provides classes for X.700 (1), CORBA (1), attributes (1), types (1) and values (14). Since this is a rather small number of classes, the syntax and semantics of accessing these are learnt quickly. (A user need only know the *GenObj*-interface and the value classes derived from *Val*). As well, the size of the executable is not bloated through a possibly huge number of classes that may be included by the client at compile time.

The availability of meta-information is an advantage in cases where flexible applications should be written, e.g. in the case of a network management browser, where information of instances of hitherto unknown classes is to be graphically displayed.

1. That is, information about available classes and their services is not compiled into the client.

In addition, meta-information allows GOM to offer built-in persistence for all instances. This means that GOM instances can be saved in a database or marshalled / unmarshalled to be sent across a network.

Having a generic model is especially useful for applications that need to handle a non-finite set of classes (extent of metaclasses not known at compile time of the client) and / or need to transparently access instances from several 'foreign' object models.

A prototype including a subset of the functionality of GOM has been created at the IBM Zurich Research Laboratory using an implementation of CORBA [4] and an OSI agent develoment platform [5]. The prototype includes creation and deletion of GOM instances using the MID; GET-, SET- and ACTION-requests for both CORBA and X.700 and a modified ASN.1 and GDMO compiler. Currently, there exists a graphical (WWW-based) user-interface for the manipulation of GOM. It uses an HTTP-server and CGI-scripts for communication between a GOM manager and the frontend. A C++-like interpreter [9] has been written on top of GOM that allows to create and manipulate instances either interactively or by the use of scripts. The interpreter has been used to provide a proof-of-concept implementation of a toolkit for 'roaming agents'[1].

Future work will include the provision of a generic event handling mechanism and the porting of the GOM toolkit to CORBA [1].

References

1. Ban, Bela. Extending CORBA For Multi-Domain Management. IBM Research Division. Zurich Research Laboratory, 1995. http://www.zurich.ibm.com/~bba/corbagom.ps
2. Ban, Bela. Towards an Object-Oriented Framework for Multi-Domain Management. Research Report RZ 2789 (89267). IBM Research Division. Zurich Research Laboratory, 1996. http://www.zurich.ibm.com/~bba/GOM.ps
3. ISO/IEC 9596. Common Management Information Protocol (CMIP).
4. SOMobjects Developer Toolkit Programmer's Reference. IBM. New York, 1993.
5. Feridun, M., L. Heusler and R. Nielsen: Implementing OSI Agents for TMN. IBM Research Division, Zurich Research Laboratory. Rueschlikon, Switzerland.
6. Object Management Group: The Common Object Request Broker: Architecture and Specification. OMG, Framingham 1992. Doc. 92-12-1
7. RFC 1157. Case, J., M. Fedor, M. Schoffstall, C. Davin. The Simple Network Management Protocol. May 1990.
8. Telecommunications Information Networking Architecture Consortium: Overall Concepts and Principles of TINA. Version 1.0, Feb. 17 1995.
9. GOMscript. An Interpreter For GOM. http://www.zurich.ibm.com/~bba/gomscript.html

1. See http://www.zurich.ibm.com/~bba/agent.html