Towards An Object-Oriented Framework For Multi-Domain Management

Bela Ban
IBM Zurich Research Laboratory
Saeumerstr. 4
8803 Rueschlikon

Dec 18, 1995

With the decentralization of computing resources into a variety of possibly heterogeneous networks and with the advent of distributed computing, management of networks has become both more important and complex. Today, network management (NM) is mostly conducted employing the Internet (SNMP) or the OSI (CMIP) network management standards. With the introduction of the Corba industry-standard [OMG], at least some of the network management tasks are likey to be done using Corba in the future. A possible scenario may be the case of Customer Premises Networks (CPNs) with the carrier employing OSI NM to manage the services provided to the customer and the customer managing his part of the network using Corba technology. Given the fact that the carrier in some cases needs to have access to some of the customer's management information and vice versa, both parties involved need to convert information to / from the other management model used. The task of conversion is very complex and requires knowledge of the other party's management model. Interchange of management information becomes even more difficult when more than two parties are involved, with each party using a different management model. In the worst case, each of the parties involved has to know the union of all management models used. Unfortunately, to know a management model doesn't yet mean to be able to program it since a lot of different APIs may exist for the same model. In OSI NM there is e.g. XOM/XMP [XOM] which is an X/Open standard, but is quite difficult to program and there are others such as the string-based value notation of cmipWorks [cmipWorks].
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, weakly-typed (runtime-checked) interface that is very flexible and suited to manage information distributed over several, disparate locations.
The structure of this paper is as follows. First, a short overview of the X.700 and Corba object models will be given. Then a summary of the work currently being done in inter-domain management will be presented. The main part deals with the presentation of the generic object model, which will be positioned against other approaches to inter-domain management.

Keywords:
Generic Object Model, X.700, Corba, Dynamic Invocation Interface, OSI Network Management, Proxy, Interoperability, Object System, Multi-Domain Management Framework

1. The X.700 Object Model

1.1. Architecture

OSI Network Management is defined by the joint ISO / ITU-T (formerly CCITT) standard documents X.7xx[1]. The approach taken by X.700[2] is to place an *agent* close to the information that is relevant to be managed and / or monitored (c.f. Fig. 1).

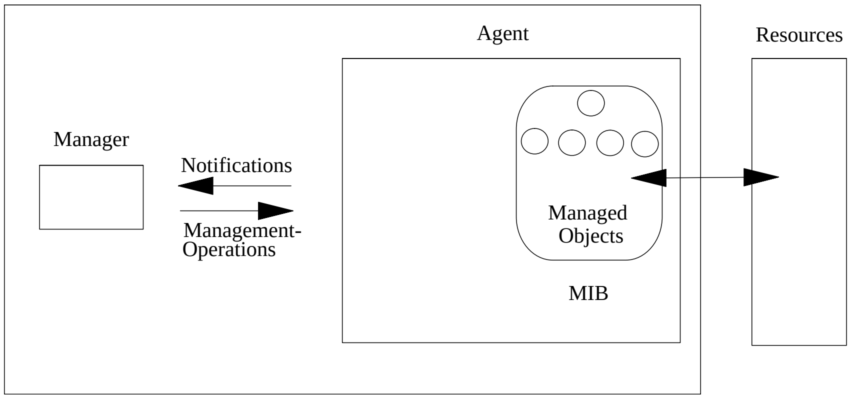

Fig. 1: Manager / Agent Paradigm

An agent[3] maintains information (in the form of *managed object instances*) about the state of a particular part of the network that it is responsible for. Queries about or changes of the state of the network can only be sent to the agent, not directly to the objects. The protocol used to interact with the agent is *CMIP* [CMIP]. A *manager*[4] can send messages and receive replies over this protocol to manage the agent. An agent may also send unsolicited information asynchronously to the manager e.g. to notify it of a problem in the case of an element failure. The Common Management Information Service Entity (CMISE) specifies 7 requests that define the interaction between manager and agent: M-CREATE is used to add management information to be maintained by the agent. The opposite is M-DELETE which removes management information from the agent. M-SET changes information in an agent whereas M-GET retrieves information from an agent. Since the amount of management information returned by an agent in response to an M-GET-request may be too large, this request can be canceled using M-CANCEL-GET. When the simple semantics of M-GET or M-SET to retrieve or change information are not powerful enough, it is possible to specify operations on managed objects which can be called using M-ACTION. If an error occurs in the network, an agent may at any time send an unsolicited message about the cause of the error to the manager using the M-EVENT-REPORT request.

1.2. Object Model

Information to be managed by an agent is modeled as managed objects. A managed object (MO) may represent either a logical resource such as a user account, or a real resource such as an ATM switch. A resource may be mapped to several managed objects, or several resources may map to one single managed object. A managed object contains *attributes*, *actions* and *notifications*. An attribute models some state in a resource (e.g. an IP-number of a router or the address of a customer). An action may for example be the addition of a new routing entry to the routing table (attribute) of a managed object that represents a router. Notfications are alerts that can be sent from the managed object to its agent which then forwards it to a manager. CMISE requests that can be sent to an agent map directly to methods applied to managed objects that the agent contains. When e.g. an M-GET request is sent to the agent, the agent will first find the

---

[1] An overview is given in the ISO/IEC documents 7498-4 (X.700) (5 functional areas) and ISO/IEC 10040 (X.701) (agent-manager architecture). The information model is defined in ISO/IEC 10165-1 (X.720).
[2] The terms X.700 and OSI Network Management are used interchangeably
[3] More well-known in the Distributed Computing community as a *server*
[4] More well-known in the Distributed Computing community as a *client*

managed object to which the request is directed, then retrieve the information from the managed object and finally return it to the sender.
The type of attributes contained within a managed object is defined using Abstract Syntax Notation One [ASN1]. ASN.1 defines atomic types such as integer, real or string and aggregate types such as lists, structs and unions. An example of ASN.1 is shown in Fig. 2:

```
BaseManagedObject ::= SEQUENCE {
     baseManagedObjectClass    ObjectClass,
     baseManagedObjectInstance ObjectInstance}

ObjectClass       ::= CHOICE {
     globalForm        [0] IMPLICIT OBJECT IDENTIFIER,
     nonSpecificForm   [1] IMPLICIT INTEGER}

ObjectInstance    ::= CHOICE
     distinguishedName [2] IMPLICIT DistinguishedName,
     nonSpecificForm   [3] IMPLICIT OCTET STRING,
     enumerateForm     [4] IMPLICIT INTEGER}
```

Fig. 2: ASN.1 example

The example defines 3 types: `BaseManagedObject` is a struct[5] which contains 2 members, namely an `ObjectClass` (another ASN.1 type defined below) named `baseManagedObjectClass` and an `ObjectInstance`. An `ObjectClass` is a union with the members `globalForm` of type OBJECT IDENTIFIER and `nonSpecificForm` of type INTEGER. An `ObjectInstance` is also a union with 3 members.
ASN.1 types are assigned to attributes within managed objects as will be shown later.
Managed objects are defined using GDMO [GDMO] notation. An example of GDMO is given in Fig. 3:

```
customer MANAGED OBJECT CLASS
    DERIVED FROM "CCITT Rec. X.721 (1992) | ISO/IEC 10165-2 : 1992":top;
    CHARACTERIZED BY
        customerPkg,
        "OP1 Library Vol. 1":contactListPkg,
    CONDITIONAL PACKAGES
        customerTypesPkg PRESENT IF ! an instance supports it !,
        "OP1 Library Vol. 1":opNetworkListPkg PRESENT IF ! an instance supports it !,
        "OP1 Library Vol. 1":serviceListPkg PRESENT IF ! an instance supports it !,
        "OP1 Library Vol. 1":typeTextPkg PRESENT IF ! an instance supports it !,
        "CCITT Rec. M.3100 (1992)":userLabelPackage PRESENT IF ! an instance supports i!;
    REGISTERED AS { iso member-body(2) 124 forum(360501) 3 46};

customerPkg PACKAGE
    BEHAVIOUR customerPkgDefinition, customerPkgBehaviour, commonCreationBehaviour;
    ATTRIBUTES
        customerID PERMITTED VALUES FORUM-ASN1-1.SystemIdRange GET,
        customerTitle GET;;

customerID ATTRIBUTE
    DERIVED FROM "CCITT Rec. X.721 (1992) | ISO/IEC 10165-2 : 1992":systemId;
    MATCHES FOR EQUALITY;
    BEHAVIOUR customerIDBehaviour;
    REGISTERED AS {iso member-body(2) 124 forum(360501) attribute(1) 228};
```

Fig. 3: GDMO example

Each managed object class is derived from at least one other class (multiple inheritance is allowed) and contains a number of mandatory and conditional packages. A package contains attributes, actions and notifications. The type of an attribute is not defined in GDMO directly, but the ATTRIBUTE clause always refers to an ASN.1 type defined in some ASN.1 file. A mandatory package (and therefore its attributes, actions and notifications) must be present in an instance of the managed object class, whereas the presence of conditional packages is determined at managed object instance creation time[6]. This implies that

[5] (in C/C++ terms). In the following discussion, C++ types are used to explain ASN.1 types
[6] Using the TMN Workbench's agent generator tool [Fer95], a developer may hard-code which conditional packages must be present before an agent is created.

although two instances may be of the same class, they may not have the same number of attributes, actions or notifications !
In the example above, the managed object class `customer` is derived from `top`. It has the mandatory packages `customerPkg` and `contactListPkg` and the conditional packages `opNetworkListPkg`, `serviceListPkg`, `typeTextPkg` and `userLabelPackage`. The `customerPkg` package defines the attributes `customerID` and `customerTitle`. `CustomerID` is subsequently defined to be derived from attribute `systemId`, which is defined in some other GDMO file. `systemId` itself will have its type defined somewhere else.

1.3. Naming

Instances of managed objects may contain other instances and may themselves be contained within other instances. The resulting structure is called a *containment tree*. Each instance within the containment tree has a *relative distinguished name* (RDN), which consists of the naming attribute of the instance and its value, e.g. customerID=(name IBM).[7] The concatenation of all RDNs from the root to an instance is called *distinguished name* (DN), e.g. netId=TelcoNet;customerID=(name IBM). A distinguished name uniquely identifies an instance within the context of its agent[8]. An instance within a containment tree can be for example accessed any time given its distinguished name.
The GDMO name binding clause defines which instances of managed object classes can be contained in an instance. If an instance of class `customer` is to be created under (i.e. ’contained in’) an instance of class `network`, then there has to be a name binding that specifies that customers can be created under networks. A name binding is also used to constrain the deletion of managed objects in that it can mandate that an instance cannot be deleted as long as it still contains other managed objects.
Using *scoping* and *filtering*, it is even possible to access more than one single instance at a time, e.g. to send a GET-request to a set of instances. Scoping selects instances starting with a base instance and specifies how many instances should be included. The scope is essentially the number of levels of children, or it can be the entire subtree starting from the base instance. The resulting set of instances can be further reduced by specifying a filter which is an expression that is applied to the attributes of each instance in the selected set of instances. If the evaluation of the filter is true, the instance is included in the set. Of the 7 CMISE requests, scoping and filtering can be used with M-GET, M-SET, M-ACTION and M-DELETE. It is thus possible to delete an entire subtree of managed object instances with a single request[9].

[7] Please note the ASN.1 value notation taken here is the one from [cmipWorks], which uses a short form for RDNs and DNs.
[8] The same naming scheme can als be used to ensure object identity across several agents.
[9] This may not be possible when the name binding specifies that contained instances have to be deleted before deletion of the parent.

2. The Corba Object Model

2.1. Architecture

OMG’s Common Object Request Broker [OMG] specifies the architecture by which instances in a heterogeneous environment can communicate with each other irregardless of whether they are local or remote. The architecture is shown in Fig. 4:

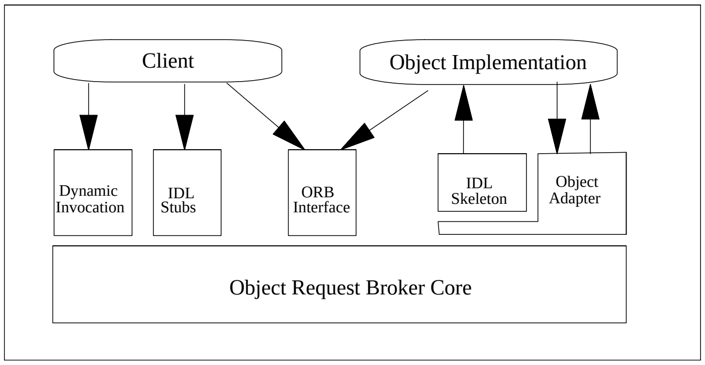

Fig. 4: The Common Object Request Broker Architecture

The Object Request Broker (ORB) is responsible for accepting requests from clients, finding the target object and its implementation, invoking the request on it and returning the result to the caller. The *interface* (c.f. below) that the client sees is completely independent of where the object is located or what programming language it is implemented in. The ORB maintains an *Interface Repository*, which is essentially meta information about the interfaces registered with the ORB and an *Implementation Repository*, which is information about where implementations of interfaces are located. The Interface Repository is needed by the ORB to marshal and unmarshal[10] requests to / from clients and the Implementation Repository is needed for locating the implementation in order to invoke requests on instances. Clients call methods of other instances using either IDL compiler-generated client stubs (c.f. 2.3), which have to be included at compile-time, or the Dynamic Invocation Interface, which allows clients to create requests to be sent to instances at runtime. Implementation *skeletons* are IDL compiler-generated implementations of the services offered by an interface. They will usually be called by a server to serve requests from clients.
When a client creates an instance (either remote or local), it receives a proxy in the form of an object reference, which is an opaque handle that uniquely identifies the remote instance. Any method invoked on an object reference will be forwarded to the remote instance and the result will be returned to the client.

2.2. Object Model

The Corba object model provides objects that isolate the requesters of services from the providers of services. An object encapsulates state (attributes) and behavior (operations) and offers access to these only through a well-defined interface. Objects interact sending messages (requests) to other objects.
Corba has atomic types such as long, string and boolean and constructed ones such as sequences, structs, or unions.
An interface defines the set of possible operations (i.e. services) that can be performed on an object. An object satisfies an interface if it can be specified as the target object in each potential request described by the interface [OMG91, p. 2.2.5]. An operation of an interface is an identifiable entity with a name, optional parameters and a return value that specifies a service that can be performed by the object.

[10] Marshalling flattens a data structure e.g. to save it in a file or send it across a network. Unmarshalling reconstructs the data structure e.g. from a data stream received over the network. In order to be able to work without the programmer having to write additional code, flattening / unflattening operations make use of meta-information.

2.3. Definition Language

Interfaces are specified using the Interface Definition Language (IDL) which is a language used to fully define all aspects of the interface, but no aspects of the implementation. An IDL example is given in Fig. 5:

```
interface Printer {
    attribute long printer_id;
    attribute Location loc;
    boolean Print(in Document doc, in long number_of_copies);
    void Shutdown();
};
```

Fig. 5: Interface definition using IDL

In the example, an interface for a printer is defined. It has the attributes `printer_id` (long) and `loc` which is an object reference to an interface `Location`. A printer offers 2 services, `Print`, which takes 2 parameters and returns a `boolean` and `Shutdown`, which has no parameters and no return value.
Since IDL is language-neutral, it has to be compiled to a specific language binding (e.g. C++). The IDL compiler takes IDL-code and generates client stubs and implementation skeletons for the specified language binding (c.f. Fig. 4). Client stubs don’t directly invoke methods on objects, but instead are essentially void methods that only forward the method to the ORB. The ORB finds the implementation of the object and sends the request to it. Then the implementation skeleton is called and the return result is sent back to the client[11].

3. Approaches To Inter-Domain Management

Inter-domain management deals with managing[12] physical and logical resources through objects of a different management paradigm (or for that matter, object model). Most approaches described below restrict themselves to the SNMP, CMIP and Corba domains, but, of course, any other object model such as e.g. COM [Brock] , ANSA [ANSA] or even C++ [Strou] can be seen as a domain. Inter-domain management is necessary in cases where one is forced to use more than a single object model. This may be the case when X.700 is predominantly used and when an SNMP-based MIB is to be accessed. Another case is the usage of Corba in a company. If the need arises to access an X.700 agent, it would be desirable to be able to access the agent using Corba so that no additional object model has to be learnt. Most approaches to inter-domain management strive for the greatest possible access transparency, i.e. if objects from domain B are to be accessed from domain A, the syntax and semantics of accessing the former should be the same as accessing objects from domain A. The advantage of this approach is that programmers only have to learn the syntax and semantics of a single domain, rather than of possibly several domains. The downside is that a mapping has to exist from A to B and possibly also from B to A if domain B is the preferred domain. This may result in a combinatorial explosion of mappings in the case where multi-domain management is desired.
Some of the work on inter-domain management deals with a CMIP manager being able to manage instances in an SNMP MIB. We will, however, concentrate on Corba managers being able to manage X.700 agents and X.700 managers being able to manage Corba agents.

3.1. X/Open Joint Inter-Domain Management (XoJIDM)

X/Open’s Joint Inter-Domain Management task force is working on the mapping from GDMO/ASN.1 to IDL [JIDM] and from IDL to GDMO/ASN.1, which shows that it is mainly concerned about the domains of Corba and X.700. Besides the translation mapping (specification translation), several people[13] are working on documents describing the use of the mapping (interaction translation) at runtime. Since most of

[11] The implementation skeleton is generated by the IDL compiler and has to be enriched with code by the programmer to implement the desired functionality (behavior).
[12] The semantics of management confine it to the area of network management. But nothing precludes the techniques presented below from being used in a more general meaning of the word; namely the creation and manipulation of instances from a different object model (domain).
[13] For further references c.f. [Soukouti1], [Soukouti2], [Mazum] and [Hierro].

the approaches are similar in nature, we will summarize the preliminary proposal submitted by Telefonica I&D [Hierro].
This approach focuses on transparently managing X.700 agents from Corba managers. Corba managers send requests to Corba servers which then communicate with the corresponding X.700 agents via CMIP. Fig. 6 depicts this approach:

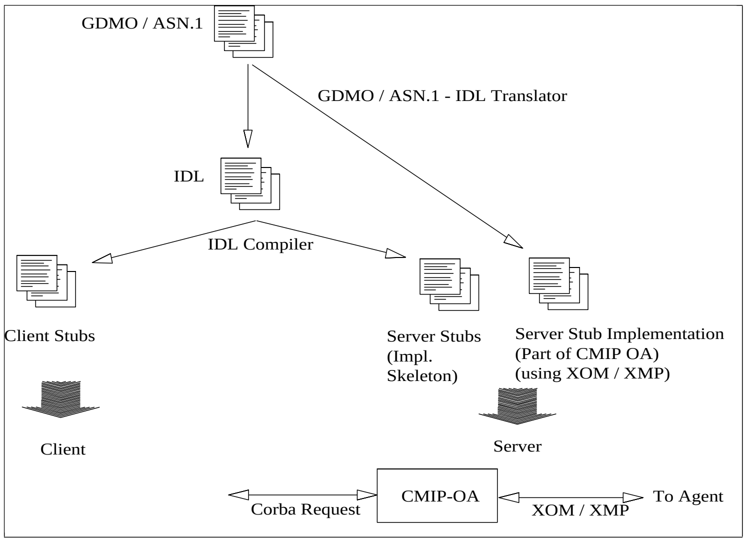
GDMO / ASN.1
GDMO / ASN.1 - IDL Translator
IDL
IDL Compiler
Client Stubs
Server Stubs (Impl. Skeleton)
Server Stub Implementation (Part of CMIP OA) (using XOM / XMP)
Client
Server
CMIP-OA
Corba Request
XOM / XMP
To Agent

Fig. 6: Specification and Interaction Translation

The GDMO/ASN.1 documents that describe the agent's MIB are translated to IDL according to the proposed XoJIDM mapping. These are subsequently translated to both client stubs and implementation skeletons in the desired language bindings (e.g. C++) using the IDL compiler. The previous GDMO/ASN.1 - IDL translation provided not only IDL interfaces, but also code (behavior) for the implementation skeletons which maps Corba requests into CMIP PDUs using XOM/XMP. The implementation skeletons merged with the generated implementation code represent fully functional proxies for managed objects in a Corba *object adapter* (CMIP OA) [OMG].
The manager includes the client stubs generated by the IDL compiler. A request invoked by the manager on a client stub will be sent to the Corba server. The proxy in the server's CMIP OA will then invoke the XOM / XMP interface and send a CMIP request to the agent. The response from the agent is subsequently converted to Corba and returned to the client. The manager looks at X.700 managed objects as if they were native Corba objects.

3.2. Other Approaches

Other approaches are proposed by [Mazum] and [Schuerf]. Both propose to add a Corba interface to an X.700 agent so that the latter can be managed from an X.700- *and* from a Corba-manager. The mapping between CMIP PDUs and Corba [OMG] calls is done entirely in the agent so that access to the agent is transparent, irregardless of which object model is used in the manager. The GDMO/ASN.1 to IDL mapping described in the previous chapter is used to generate IDL code which is included by a (DSOM) manager and which is used as implementation skeleton in the agent. Since this mapping is generated at compile time, a DSOM manager necessarily has to recompile whenever the mapping is regenerated, e.g. in case of changes in the GDMO/ASN.1 files.

3.3. Evaluation

Both approaches described above have some inherent limitations, but also some advantages. A disadvantage is that some features of X.700 cannot be adequately mapped to Corba using a static translation approach, but rather require a dynamic, meta-information based approach. In the case of X.700 conditional packages, the solution proposed by X/Open [JIDM] is to map all packages to IDL interfaces and, when mapping a GDMO template to an IDL interface, to make the latter inherit from all possible

packages that it may include. When an instance of the resulting class would be instantiated, a list of all conditional packages would have to be maintained, and when access to an attribute of a non-available package would be requested, a runtime Corba exception would be raised. This approach substantially increases the size of a class and incurs a penalty for users of that class if they need only a few conditional packages rather than all of them[14].
Another problem is the ASN.1 type ANY DEFINED BY which is a type that is only known at runtime and that cannot be statically determined.
All the above problems are due to the dynamic nature of some features of X.700 for which a dynamic, runtime-based rather than compile-time based approach seems to be better suited.
A problem of a more general nature is the fact that managers include generated code (e.g. by the GDMO/ASN.1-IDL compiler), so that whenever it is re-generated, e.g. in the case of a GDMO class template modification, the manager has to be recompiled as well. Also, sending a request to an agent via a Corba server incurs a time penalty. Finally, the XoJIDM mapping implies a substantial amount of classes and structs that are generated and included in the manager which may result in a bloated executable.
An advantage of [Hierro] is the fact that all the client stub implementation code is generated at compile time so that no runtime overhead exists on the manager side, but this is partly negated by the fact that most of the time used for a CMIP request is spent in the OSI stack anyway. Strong typing is another advantage, but this can be mitigated by runtime type-checking mechanisms[15].

4. The Generic Object Model

4.1. Introduction

The generic object model (GOM) is another approach for inter-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 the GOM is shown in figure 7 below:

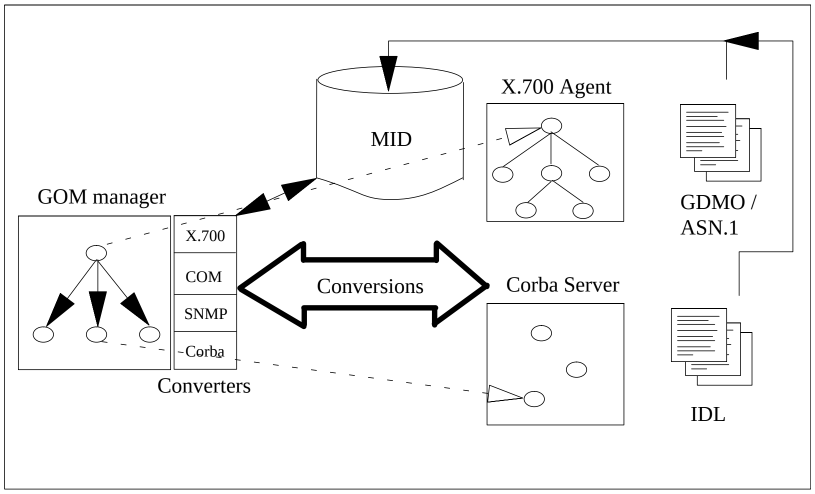

Fig. 7: The Generic Object Model

In most approaches to inter-domain management (c.f. section 3), code that converts information from one object model to another is generated at translation time. In [Hierro] for example, IDL code is generated from a GDMO/ASN.1 description. Subsequently, the IDL description has to be translated into a Corba language binding, e.g. C++. A manager then includes the generated C++ header files. The disadvantage of

[14] A similar problem exists with extensible attribute groups. These are collections of attributes to which attributes may be added or removed at runtime.
[15] Of course, runtime type-checking is bound to be slower than compile-time type-checking.

this approach is that the manager has to be recompiled whenever the original GDMO/ASN.1 files are modified or when new files are added.
The approach taken by GOM is to collect meta-information from GDMO/ASN.1 and IDL in a Meta-Information Database and access the instances in an X.700 agent or a Corba server using this meta-information rather than having to include generated header files. This makes the client independent from the server since the *contract*[16] offered by a class is retrieved dynamically at runtime from the MID rather than at compile-time from the 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 customizeable *converters* that make use of meta-information from the MID. A converter translates information back and forth 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[17].
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.

4.2. 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. In contrast to the X/Open approach [JIDM], which creates a class for each managed object template that is encountered in GDMO, in GOM there are only instances of these two classes. 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. Each value class has conversion operators to and from C++ and to and from other GOM values (e.g. Integer to Long). There is a finite number of GOM types available for usage by the programmer.

4.2.1. Example

In this section, some examples of how to manipulate GOM instances are given.
GOM instances are created using the MID as *factory*[18] object:

```
// network1 already exists, e.g. by retrieving it from a directory server
Location loc("X");
GenObj* cust=MID::Create("customer", network1, "netId=Telco;customerID=(name IBM)",
                         &loc, 0, 0);  // no attribute list given (c.f. next example)
```

This creates an instance of X700_Obj with the distinguished name given under the already existing instance `network1`. The creation of a GOM instance is the only time when a user needs to decide what type of instance should be created. In the case of an X.700 instance, a parent and a distinguished name need to be given, both of which would be optional with a Corba instance. If Customer occurs both as a managed object template and a Corba interface, the module in which it is defined would have to be provided as well. This has been omitted for simplification.
The creation request is processed as follows: The MID object looks up the definition of `Customer` and finds out that it is an X.700 object. It then creates the instance and sends a CMIP CREATE-request to the agent the location of which is determined by `loc`[19]. Subsequently the newly created instance is returned.
The following code creates a Router instance in the Corba server on host X as child of network1:

[16] A *service* is an operation offered by a class. The set of services offered by a class comprise the contract.
[17] In the case of X.700, a CMIP PDU is generated and sent to the agent.
[18] c.f. [Gamma]
[19] How the generic concept of location is mapped to a concrete location is not detailed here for space reasons.

```
Location loc("X");
AttributeList al("name", new String("Cisco_162-3"), "location", new String("Room S187"),
                 "routerNumber", new Integer(2445), 0);
GenObj*   cisco1=MID::Create("CiscoRouter", network1, "cisco1", &loc, 0, &al);
```

The MID looks up the definition of CiscoRouter and finds out that it is a Corba interface. It then creates a DII request (including the initial attribute values from the attribute list) and sends it to the ORB to create an object of class CiscoRouter. If successful, a GOM instance of class CorbaObj is instantiated and returned. If another manager needed a reference to the Cisco router instance, it would make the call:

```
GenObj* router=MID::Find("cisco1");
```

If an agent is already running and the manager wants to retrieve some instances in it, it would call the method Find of the MID:

```
Scope s(entire_subtree);
GenObj* cust=MID::Find("netId=Telco;customerID=(name IBM)", &s, 0);
```

If found, an instance of class X700_Obj is returned which contains all children (possibly other customers, or equipment under customer). It is important to note that, since objects in GOM have a pointer to their children, a Corba object can be created under an X.700 object, i.e. an X.700 object can contain Corba objects and vice versa[20].
The following example shows how values can be retrieved from GOM instances:

```
// cust is already known
Union* custId=cust->Get("customerID");
cout << *custId << endl;  // prints '(name IBM)'
```

If the attribute `customerID` was not present in the GOM instance, a NULL pointer would be returned. This would happen in the case where an attribute was defined in an optional GDMO package that was not instantiated. The following example shows how to manipulate the router instance:

```
Long* in=cisco1->Get("in_pkts");
if(in) {
  *in=0;
  cisco1->Set("in_pkts", in); // equivalent to cisco1->Set("in_pkts", new Long(0));
}
```

The Set()-method performs type-checking using the information of the type member of the attribute. If e.g. a string was to be assigned to the attribute `in_pkts` of `cisco1` (which is a long), the Set()-method would fail:

```
cisco->Set("in_pkts", new String("foo"));  // fail, string cannot be assigned to long
cisco->Set("in_pkts", new Integer(0));     // o.k., integer can be converted to long
```

4.3. 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 loose as in the 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 the 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

[20] Since a Corba object cannot contain an X.700 object, this is meant in the sense of the first having a reference to the latter. The X.700 object must have an X.700 parent.

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. On 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 with 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 the backend of which has been modified to create entries for the MID database. 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. The MID is a primitive trader [X.903] in the sense that services are exported by the IDL or GDMO/ASN.1 compilers and can be created or imported by the clients using the MID::Create() or MID::Find() operations.

4.4. Attributes and Operations as Instances

Having a loose binding between client and server and having attributes and values as instances has a number of advantages, especially in the area of X.700 which is considerably more flexible and necessarily also more complex than Corba[21]. It is for example possible to accomodate X.700 conditional packages (c.f. section 1.2) which are included at instance creation time depending on the user’s decision. The approach taken by GOM is to initially include all attributes and operations of only the mandatory packages of a managed object class in the resulting GOM instance[22]. When a request is sent to the instance referring to an attribute or operation residing in a conditional package, the attribute or operation is created using the MID and added to the attribute- or operation-list respectively[23]. This is in contrast to other approaches, where all attributes and operations are made part of the resulting (C++) class and where a precondition before access to attributes/operations checks whether the corresponding conditional package has really been instantiated.
Since it is easy to add or remove attributes from the attribute-list of any GOM instance, the problem of X.700 extensible attribute groups (c.f.[GDMO]) can be easily solved. Whenever an attribute needs to be added, it is just added to the attribute-list. The problem of ASN.1 ANY-DEFINED BY ([ASN1]) types which are essentially types that are determined at runtime, is also easily solved by GOM since it keeps meta-information. As soon as the referred-to type is known, the ANY type is made to point to the same type as the referred type. This is a problem in stongly typed languages such as C++ where the type system is fixed after compilation and cannot be modified or extended.

4.5. Transparency

One of the goals of the GOM is to hide the complexity of the underlying specific object models on one hand and to offer a uniform, object-oriented programming model on the other hand. Having to program against a single API rather than against multiple different APIs is an advantage because programmers have to learn the syntax and semantics of an API only once and may subsequently use it to program different specific object models. This is similar to the situation of the graphical user interface (GUI) market where many different GUIs exist. To ease the problem of having to develop applications that run unmodified on different GUIs, virtual GUI-toolkits have been introduced that offer a single (generic) API and converters (libraries) for each GUI. Code written against such an API is inherently more portable and costs of programmers having to learn several GUI APIs are reduced to learning only a single API.

5. Conclusion

Using the GOM offers significant amount of flexibility to programmers. First, the binding between client and server is not tight[24]; servers may be modified without the client having to be regenerated 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.

[21] For a detailed comparison of the X.700 and Corba models see [Rutt] or [Quinn]
[22] The list of mandatory and conditional packages is available from the MID.
[23] Of course, the attribute or operation may not be present in the managed object. In this case, the newly created elementwill not be added to the attribute- or operation-list and an error message will be returned.
[24] That is, information about available classes and their services is not compiled into the client

Providing this flexibility increases the chance that future object models may be integrated with the 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 to GOM would not require a programmer to learn a new syntax or semantics since GOM can be used to manipulate instances of the new model.
GOM currently only has 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 generated.
The provision 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. 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.

A prototype including a subset of the functionality of GOM has been created at the IBM Zurich Research Laboratory using an implementation of Corba [DSOM] and an OSI agent develoment platform [Fer95]. The prototype includes creation and deletion of GOM instances using the MID, GET- and SET-requests and a modified ASN.1 and GDMO compiler. Currently, there exists a graphical user-interface for the manipulation of GOM. It uses a modified HTTP-server and CGI-scripts for communication between a GOM manager and the frontend.
Future work will include the addition of operations and events (notifications) to the GOM. Also, ways of dynamically (i.e., at runtime) loading user-defined customization code such as converters, OSI stacks or policy objects[25] will be investigated. Some significant amount of work has already been done in that direction, c.f. [Deri].
Another important enhancement will be the integration of other object- or information-models such as COM [Brock] or SNMP [SNMP].

6. Acknowledgements

The author would like to thank Luca Deri for encouragement and numerous valuable discussions on the generic object model.

[25] Policy objects (called *strategies* in [Gamma]) are currently used in the prototype to customize caching of attribute values.

7. References

[ANSA] *The ANSAware 4.1 manual set*. Architecture Projects Management. Poseidon House, Castle Park. Cambridge, 1993.

[ASN1] ISO/IEC 8824. *Specification of Abstract Syntax Notation One (ASN.1)*.

[Brock] Brockschmidt, K. *Inside OLE2*. Microsoft Press. Redmond, 1994.

[Challa] Challa, Siva. *Towards an Interoperable, Reflective Common Object Model for Statically-Typed Object-Oriented Languages*. Dept. of Computer Science. Blacksburg, Virginia, 1994.

[CMIP] ISO/IEC 9596. *Common Management Information Protocol (CMIP)*.

[cmipWorks] Geiger, G., W. Allen, A. Majtenyi, P. Reder. *IBM cmipWorks Technical Paper*. IBM, March 1994.

[Deri] Deri, Luca. *Breaking Monolythic Applications Apart*. Internal Report. IBM Zurich Reasearch Laboratory. Rueschlikon, 1995.

[DSOM] *SOMobjects Developer Toolkit Programmer's Reference*. IBM. New York, 1993.

[Fer95] Feridun, M., L. Heusler and R. Nielsen: *Implementing OSI Agents for TMN*. IBM Research Division, Zurich Research Laboratory. Rueschlikon, Switzerland.

[Gamma] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading 1995.

[GDMO] ISO/IEC 10165-4. *Guidelines for the Definition of Managed Objects*. 1992.

[Hierro] Hierro, Juan: *Architectural Issues For Using Corba Technology in OSI Systems Management*. Append of draft to XoJIDM forum.Aug 1994.

[JIDM] *Translation of GDMO/ASN.1 Specification into CORBA-IDL (Draft 0.0)*. A Report of the Joint X/Open / NM Forum Inter-Domain Management Taskforce. Editor: Subrata Mazumdar, IBM T.J. Watson Research Center (mazum@watson.ibm.com).

[Mazum] Mazumdar, Subrata: *Design of DSOM based OSI Network Management Agent*. IBM T. J. Watson Research Center. Hawthorne NY, 1994.

[OMG] Object Management Group: *The Common Object Request Broker: Architecture and Specification*. OMG, Framingham 1992. Doc. 92-12-1

[Quinn] Quinn, Peggy and George Preoteasa. *Reconciling Object Models for Systems and Network Management*. UNIX System Laboratories, Inc. Summit, NJ 1993.

[Rutt] Rutt, Tom. *Comparison of the OSI management, OMG and Internet management Object Models*. A Report of the Joint X/Open/NM Forum Inter-Domain Management Task Force, 1993.

[Schuerf] Schuerfeld, Ute and Dieter Gantenbein: *Bilingual Agent. DSOM Access To X.700 Agent*. IBM Zurich Research Laboratory, March 31, 1994.

[SNMP] RFC 1157. Case, J., M. Fedor, M. Schoffstall, C. Davin. *The Simple Network Management Protocol*. May 1990.

[Soukouti1] Soukouti, Nader: *Managing OSI Objects Using A Corba Manager*. Append of draft to XoJIDM forum.

[Soukouti2] Soukouti, Nader: *Toward Managing Corba Objects Via OSI Network Management Mechanisms*. Append of draft to XoJIDM forum.

[Strou] *The C++ Programming Language*. 2nd edition. Addison Wesley, 1992.

[XOM] *X/Open OSI Abstract Data Manipulation API*. CAE Sepcification.

[X903] ISO/IEC JTC1/SC21/WG7 N807 / X.903. *The ODP Trader*.

8. Biography

Bela Ban received his degree in computer science from the University of Zurich in 1993. He worked for IBM Switzerland for 3 years and is now working towards his Ph. D. at the IBM Zurich Research Laboratory. His interests include distributed programming, object-oriented technology, weakly-typed languages and systems management. He can be reached at bba@zurich.ibm.com.