Extending CORBA For Multi-Domain Management

Bela Ban
IBM Zurich Research Laboratory
Saeumerstr. 4
8803 Rueschlikon
bba@zurich.ibm.com
http://www.zurich.ibm.com/~bba

13. August 1996

1 Abstract

The CORBA architecture [OMG] will be of significant importance in the future for building distributed systems. Nevertheless there are other systems (object models) that exist for the purpose of distributed computing such as COM [Brock], TINA [TINA], ANSA [ANSA] and for the purpose of management such as the OSI [X700] and SNMP [SNMP] network management standards. The goal of this paper is to show that the CORBA architecture can be enhanced to allow transparent access to those systems. The approach chosen is based on a generic object model [Ban95] that uses CORBA as distribution mechanism and provides a generic API to clients, the *dynamic object model*. Another component is the concept of adapters that take care of bridging requests between object models; their main use is the conversion of requests from one system to another. Their functionality can be compared to CORBA's object adapters, however, contrary to those, they are located at the client rather than the server side. The third component is the extended interface repository (EIR). It is essentially CORBA's interface repository (IR) enhanced to accomodate not only meta information about CORBA interfaces, but about other models as well. The intention is to make the IR a distributed meta-information service for all object models[1].
This work complements the NMF - X/Open Joint Inter-Domain Management task force [Telefonica, JIDM] approach for inter-domain management. While JIDM's model of specification translation is of a static and strongly typed nature, our proposal focuses on a dynamic, runtime-based approach. While this proposal uses the DII, located on the client side, JIDM's work is based on the concept of a CMIP object adapter, located at the server.

1.1 Definitions

- object, instance: used synonymously
- domain: used synonymously with object model (and system). Although domain is traditionally used to denote an administrative boundary (e.g. accounting- or security-domain), it is used here as defined by JIDM.
- proxy object, underlying object: a proxy object refers to a (possibly remote) underlying object
- underlying class: class in a specific object model that is mapped to a class in the dynamic object model
- classes / interfaces: used synonymously, a CORBA interface denotes the IDL definition of a CORBA class
- object reference: logical pointer to a CORBA instance (local or remote)
- agent, server: an OSI agent is still mostly used for management services, but conceptually, it may offer *any* service (such as a database server). Used more or less synonymously.
- manager, client: used synonymously.

[1] In this respect it is similar to the concept of trader [ODP].

2 Overview of the Architecture

The architecture is shown in Figure 1:

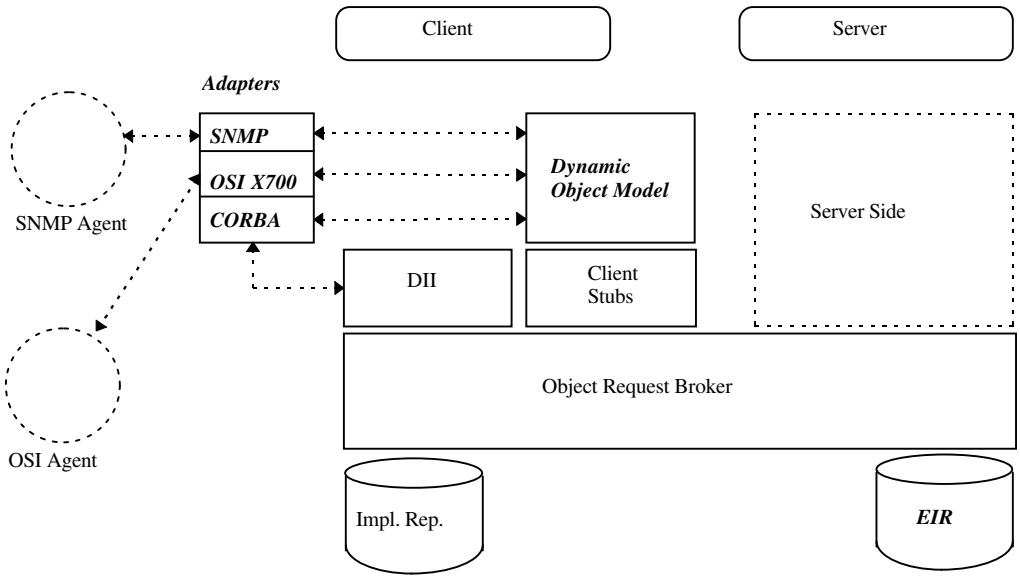

Figure 1: Architecture

Modified / extended components of the CORBA architecture are shown in bold italic font. The dynamic object model will be described in section 2.1, the concept of adapters in 2.2 and the extended interface repository in 2.3.

2.1 Dynamic Object Model

The dynamic object model provides a generic API for the uniform handling of multiple object models. Among its goals are:

1. To provide an object-oriented, easy-to-use model that encapsulates idiosyncrasies of different object models in one single generic object model.
2. The ability to handle instances of classes that were not known at compile time. A prerequisite is the availability of runtime composition of instances through addition of attributes and methods to an instance. This feature is not available in statically typed languages such as C++ where the layout of a class has to be known at compile time.
3. Transparent manipulation of foreign models through the use of meta-information and adapters.

The proposed model maps classes from all possible models to instances of a single generic class called *GenObj*. *GenObj* is a CORBA interface and has attributes and methods. Attributes are instances of the CORBA class *Attribute* and methods are instances of the CORBA class *Method*. An attribute has a value which is modelled as a CORBA class as well[2] . The model looks as follows:

[2] With respect to the mapping of classes and its contained elements as classes themselves, the model proposed here bears similarities to the generic aspects of the GDMO++ and ASN1++ APIs proposed in [TMN++].

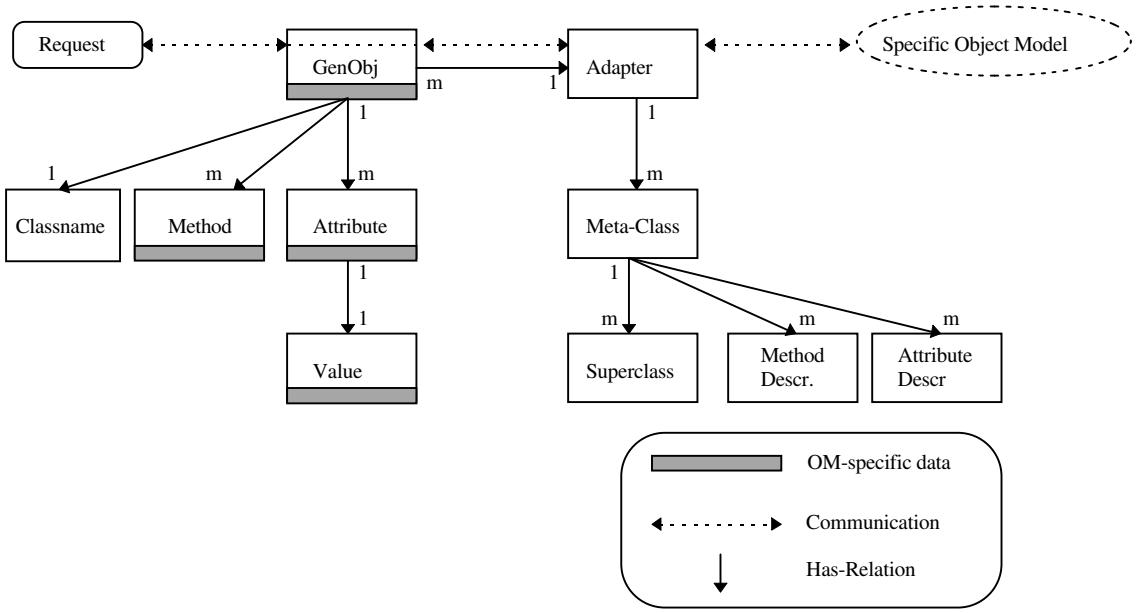

Figure 2: Dynamic Object Model

GenObj is a *proxy object* in the client's address space that refers to an *underlying object* which may be located in another process or even at another location. Requests sent to it are transparently *forwarded* to the underlying object and responses are returned to the client.
Conceptually, each proxy object has a list of attributes and methods. Each list only maintains the names of attributes and / or methods that it contains; the description of the attribute- / method-definitions is kept in the metaclass. Each attribute has a value which is modelled as an instance of a subclass of the CORBA class *Value*. The dynamic object model has a pre-defined set of *Value* classes such as *Integer*, *String*, *Boolean*, *List*, *Struct*, *Union* etc.
Modelling the concepts of attributes and methods as classes of their own[3] offers the ability to construct proxy instances at runtime. Having only meta-information about the class available, *and not the class itself (as a CORBA interface, e.g. 'interface user')*, this ability is a pre-requisite for runtime class instantiation. It is important to note that the underlying classes must have a definition in the server (e.g. a CORBA interface named "user" in a CORBA server or a GDMO class template named "circuit" in an OSI agent), whereas in the dynamic object model, the classes "user" and "circuit" would each be mapped to an instance of class *GenObj*, having a reference to their metaclass. This scheme allows the dynamic object model to represent all classes, that will ever be encountered, as instances of *GenObj* and clients need not be recompiled when new classes are added or existing ones modified. The only prerequisite for manipulating instances of underlying classes is that their meta-information is available and that an adapter exists for that specific object model.
Modelling all classes of the dynamic object model as CORBA interfaces has a number of advantages. First, they may be created at any location (local or remote). Second, they may be accessed from any language for which a language binding exists (e.g. Java or Smalltalk). And third, their implementation may be written in any language with a CORBA binding. This allows the creator of an adapter to choose which language should be used to implement it rather than being tied to a single language.
Since the classes in the dynamic model represent classes of several specific models, the problem of mapping each model's idiosyncrasies to the dynamic model exists. The approach for handling this is to include in the dynamic model only concepts that can be found in most models and to introduce in each class an attribute that can be used to store object-model specific information[4] (see `GenObj::specific_data` attribute in Appendix A: Corba Classes). The alternative would have been to create subclasses of `GenObj` with model-specific information. This was rejected because a comparison of some models (CORBA, X.700, COM) showed only minor differences between them. Also, the addition of subclasses for each new model that was to be integrated would have led to an explosion of classes which, in most cases, would only have contained a single new attribute anyway. Also, addition of subclasses would have led to the recompilation of clients of the dynamic object model each time a new model was introduced.

[3] This concept of representing aspects of an object-oriented system as objects themselves is called *reification* [Challa].
[4] This may for example be the distinguished name or the class object identifier (OID) in the X.700 object model.

The concept of a model-specific attribute is *not* used in the extended interface repository (see section 2.3) because we expect the meta-models to be more different than the instance models. However, if it is seen that they are closer to each other than anticipated, we will reconsider handling the differences by adding the same model-specific attribute to all meta-model classes. In this case, the addition of subclasses to the EIR doesn't lead to recompilation of clients since EIR (sub-)classes are only used in adapters.

2.2 The Concept of Adapters

Adapters are the bridges between the dynamic CORBA-based object model and the specific models and are the key component of this proposal. They take care of conversion between the dynamic and a specific object model. As a matter of fact, adapters are the only contact points between the dynamic object model and the outside world (i.e. foreign model). The author of an adapter has to know both the dynamic and the foreign model.
An adapter is implemented as a CORBA interface and has to be derived from the `Adapter` interface (c.f. Appendix A: Corba Classes). Each adapter has to implement the abstract protocol dictated by the parent interface. It may add additional methods and attributes, but those are not visible to clients since the latter can only see the abstract interface offered by `Adapter`. They may, however, be used by the adapter, e.g. for storing state related to the foreign model such as the connection to an OSI agent.
The tasks of adapters is to create foreign objects[5] , access them and return results to the clients. Since adapters are CORBA interfaces themselves, they may be created at a remote location, but for performance reasons, they will usually be created locally. The ability of an adapter to be created at any location offers an important advantage: they can be instantiated on the machine on which they were implemented. For example, if the dynamic object model implementation resides on a UNIX machine and a COM[6] adapter is needed to access OLE-instances [Brock], then the COM-adapter can be implemented and instantiated on a Windows NT server on which OLE is available. This would not be possible on the UNIX machine on which the OLE libraries are not available[7] . Employing this scheme, COM instances in the NT-server can be transparently manipulated by the manager on the UNIX-machine using the dynamic object model.
Figure 3 shows how adapters are employed in the dynamic object model:

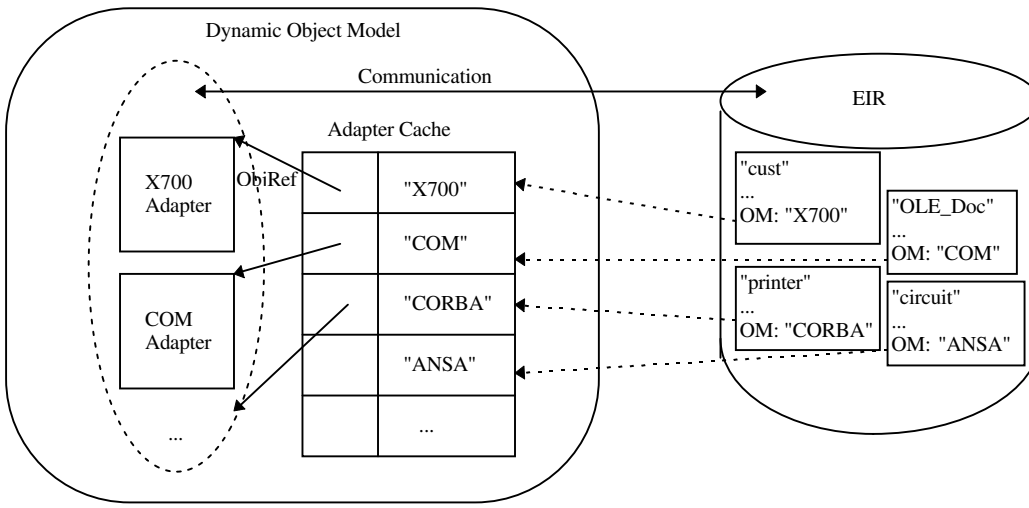

Figure 3: Links Between Adapters And Contracts

A *contract* in the EIR is the definition of a class (i.e., a metaclass) and describes its methods, attributes, inheritance relations etc. Each contract has an (inherited) attribute that contains the name of the adapter interface that is to be used for conversion to the foreign object model. In the example above, the contract "cust" specifies that an adapter named "X700" should be used for communication with the foreign object model. It is of course possible to determine the adapter to be used with more than just a string attribute containing the adapter's interface name; one could think of specifying an attribute-set of properties similar to concepts found in the ODP trader [ODP].

[5] There will be a default adapter which handles CORBA objects (using the DII).
[6] Component Object Model. Microsoft's object model underlying OLE.
[7] Of course, a CORBA implementation has to be running on the Windows NT machine.

An adapter for each object model is kept in the object model's adapter cache so that the EIR only has to be accessed the first time an adapter is needed. All accesses thereafter will use the cached adapter[8] . Since adapters are CORBA interfaces themselves, their definition is kept in the (CORBA-part of the)EIR as well.
When adding information to the EIR, each contract that is added has to specify which adapter is to be used[9] . Thus, a logical link between a contract and its adapter is created. This allows for easy integration of other object models since whoever *adds* information to the EIR also has to know how to *convert* it to the dynamic object model and vice versa and therefore also has to furnish an adapter and provide the link between meta-information and adapter. Integration of another specific model therefore involves the generation of meta-information from the model specification (e.g. through the means of a translator), the addition to the EIR and the creation of an adapter.
Figure 4 shows in detail how an adapter works:

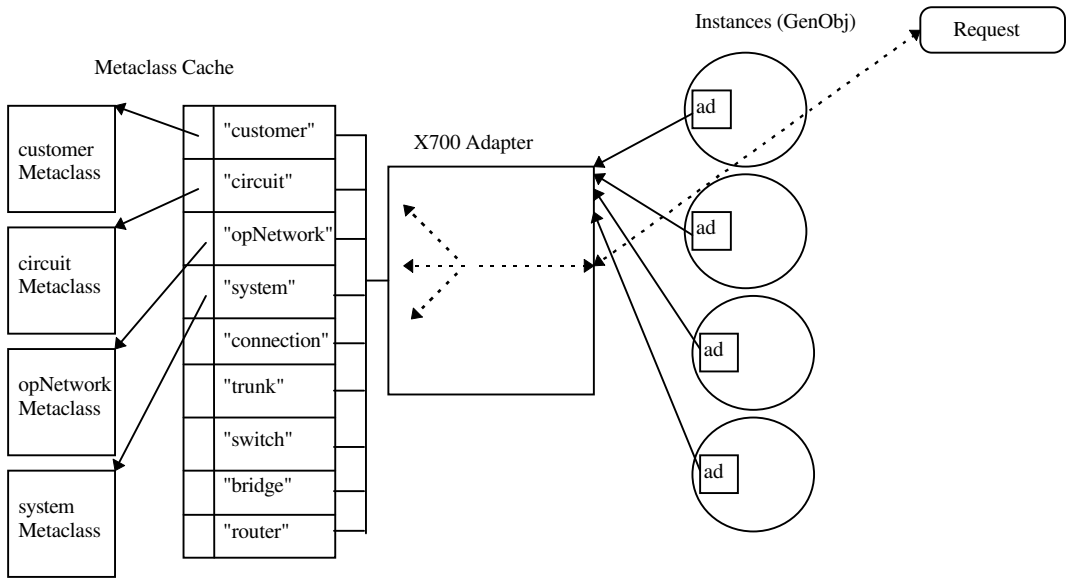

Figure 4: A Closer Look At An Adapter (X.700)

Each adapter has a metaclass cache that is used when accessing the underlying objects. Each proxy instance has as one of its attributes a reference to the adapter that created it. This attribute is used to forward requests sent to the proxy to the adapter for processing. Thus the entire functionality (i.e., conversion knowledge) is concentrated in the adapter. The next section describes two use cases: the creation of objects and the processing of requests sent to objects.

2.2.1 Use Cases

Preliminary definitions for some of the CORBA interfaces mentioned below are given in Appendix A: Corba Classes.

2.2.1.1 Creation of Objects

1. A create request for a new object is received by a factory[10] object. It contains the name of the class to be created, the type (optional) of the object model (e.g. "X700", "CORBA") and the location[11] .
   Ex: `GenObj* psp30=factory->Create("printer", "CORBA", "adlerhorn", 0);`
2. If the name of the object model is not given, then the name of the adapter to be used is looked up in the EIR using the classname as key. Every class in the EIR contains the name of the adapter that is to be used (e.g. "printer" -> "CORBA", "OLE_Doc" -> "COM" etc, c.f. Figure 3).
3. The adapter is created (normal CORBA object) if not yet present in the dynamic object model's adapter cache. (In most cases, adapters will already be in the cache, c.f. Figure 3). The constructor of the adapter

[8] Of couse, the adapter cache may be flushed at any time, so that adapters have to be re-created. This is needed when adapters have been modified and should be reloaded without the client having to be restarted.
[9] Alternatively, a client may specify the adapter to be used at instance creation time.
[10] A factory is a well-known object that creates instances given their class name.
[11] The location is only conceptual and eventually maps to a specific location such as the name of a CORBA server or the AE-title of an OSI agent [X700].

may initialize environment-specific data now (such as the establishment of a connection to an OSI stack or the initialization of an ORB-runtime).

4. The create request is forwarded to the adapter: the class of the object that is to be created (i.e., its metaclass) is looked up in the adapter's metaclass cache or, if not found, retrieved from the EIR[12] and subsequently added to the cache (c.f. Figure 4)[13] .
5. Using this metaclass, the real (underlying) object (e.g. an OSI managed object or a COM object) *and* its proxy (an instance of GenObj) are created. The attribute 'ad' of the proxy is set to point to the adapter that created it. The 'ad' attribute is important for the handling of further requests sent to the object (see 2.2.1.2). Finally, the object reference to the proxy is returned to the client. Note that the proxy instance contains only a list of attributes (and their values) and methods. Their definition is located in the metaclass that resides in the adapter's metaclass cache. Note that the attribute 'any' of the proxy object may be used to store instance-specific data such as the distinguished name in the X.700- or the ObjRef in the CORBA model. This solution makes the creation of subclasses of *GenObj* for the storage of instance-specific data needless.

2.2.1.2 Sending requests to Objects

1. A Get-request is received by the proxy object. If the attribute_name is not a member of *attributes*, the request is rejected.
2. Otherwise, the request is forwarded[14] to the adapter that created the object using the 'ad'-attribute. Arguments are the name of the class (e.g. "cust") and the objref of the underlying object. Ex (in C++):
   `ad->Get(this, class_name, attribute_name);`
   The object reference is needed in the adapter to retrieve information from the instance, for example for a CMIP-request to be sent to an OSI agent the distinguished name of the instance (residing in the `GenObj::specific_data` attribute) is needed. The classname is needed to lookup the metaclass in the adapter's metaclass cache. The reason for choosing to add this indirection and not to directly include in each object a reference to its metaclass was for the purpose of flexibility. This scheme allows for runtime meta-information modification. Through flushing the metaclass cache and subsequent re-population, modified meta-information can be accessed and used immediately without having to restart the client.
3. The adapter handles the request on behalf of the object using the meta-information in its adapter class cache. This involves type-checking, conversion of the request and forwarding to the underlying object (e.g. sending a CMIP PDU or a DII request), receiving the result, converting and returning it to the proxy object.

2.3 The Extended Interface Repository

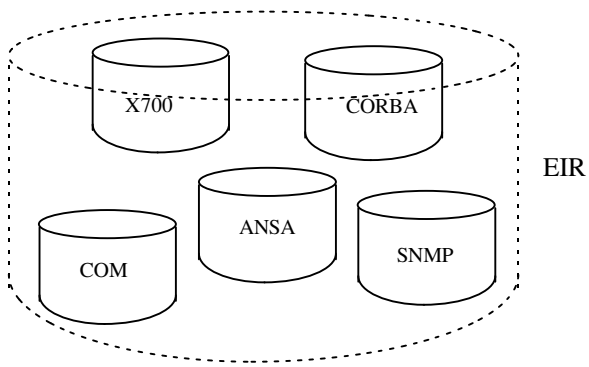

Figure 5: The Extended Interface Repository

In this paper, we propose to extend the CORBA interface repository (IR) towards a global multi-domain meta-information service. This means essentially that the IR becomes a (distributed) repository for meta-data from several object models (not just CORBA). Figure 5 shows the conceptual model of the extended interface repository (EIR). It contains a database for each object model which has meta-model instances that describe the elements of the object model, such as classes, operations, attributes etc.
Figure 6 shows how the EIR is structured internally:

---

[12] The metaclass, including all attribute- and operation-descriptions, is essentially copied to the adapter's class cache.
[13] Note that there are two caches: one for adapters in the dynamic object model and one for the metaclasses in *each adapter*.
[14] That is, the corresponding method of the adapter object is called. Each method of a proxy is therefore just a stub that calls the corresponding adapter-method.

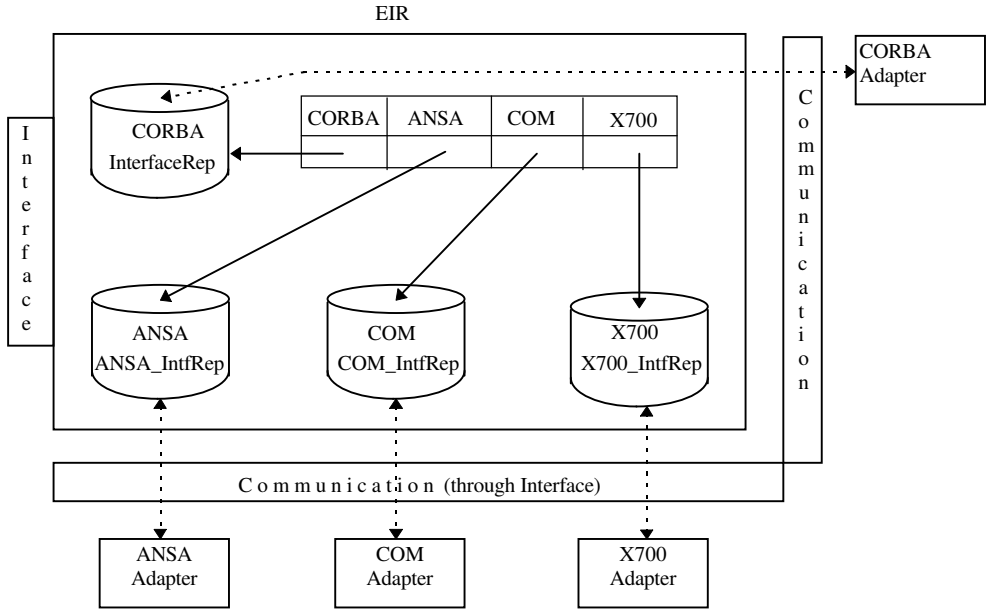

Figure 6: The Structure Of The Extended Interface Repository

When retrieving meta-information from the EIR, if the name of the object model is given, then the corresponding database is searched directly for the metaclass entry. Otherwise all databases are searched sequentially (CORBA db first) until the metaclass is found.
The CORBA interface repository, as it exists now, is basically a CORBA interface (`InterfaceRepository`) and can therefore be subclassed to accomodate the needs of other meta-models. It is proposed to add an attribute (e.g. `other_models`) to `InterfaceRepository` that contains a list of instances of subclasses of `InterfaceRepository` [15]. These subclasses essentially represent the various other object models. Whenever a new object model is to be integrated into the EIR, all that has to be done is to define a subclass of `InterfaceRepository` and add an instance to `other_models`. Thus the interface to the EIR can remain stable even if new models are added.
The CORBA `InterfaceRepository` instance in the EIR is the same as in the original IR and contains the same elements as before. An instance of `InterfaceRepository` contains instances of interfaces that describe concepts such as interfaces (`InterfaceDef`), operations (`OperationDef`), attributes (`AttributeDef`), types (`TypeDef`), exceptions (`ExceptionDef`) and so on. Since those concepts may vary among models, each model may subclass some or all of the above classes and implement the desired structure (e.g. additional attributes) or behaviour (e.g. additional operations) where needed to express its idiosyncrasies. Figure 7 gives an example:

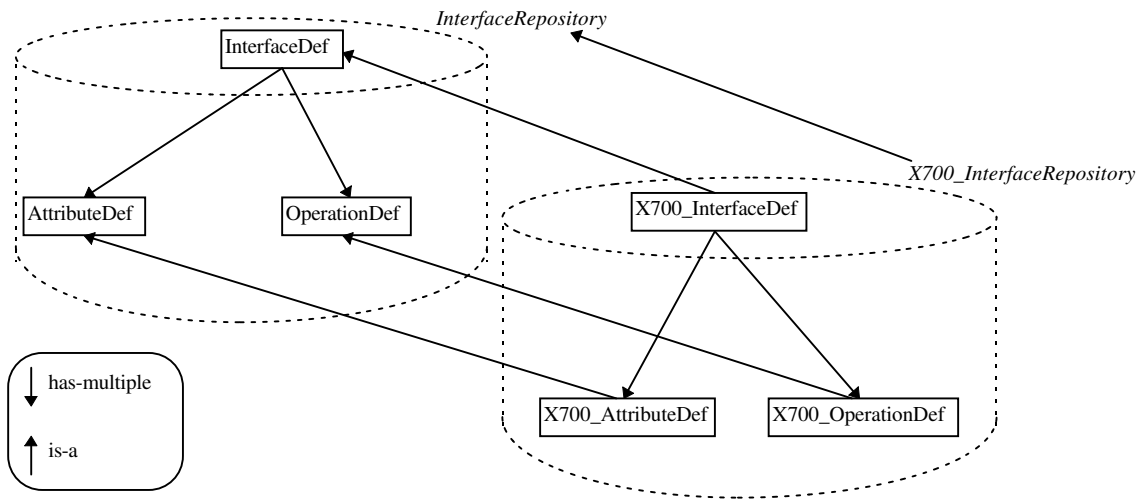

Figure 7: Relation Of Object-Model Specific Meta-Data To Dynamic Object Model Meta-Data

[15] It is not necessary to add this attribute to the IDL definition of `InterfaceRepository`; it can be added to the implementation code that results from IDL-translation. Therefore only the implementation, not the definition of `InterfaceRepository`, has to be modified.

In this example, the class `X700_InterfaceDef` is derived from `InterfaceDef` which describes an interface. The latter models by default a CORBA interface, whereas the first represents a GDMO class template. The subclass is necessary because it needs to contain X.700-specific information such as packages or OIDs. The same holds true for X.700 attributes; they are more complex than their CORBA counterparts and therefore cannot be represented using CORBA `AttributeDefs`. In cases where concepts of an object model can be represented with normal IR elements (classes), of course no subclasses are needed.
It is important to note that, although most CORBA classes will be subclassed to accomodate other object models, the interface to the `InterfaceRepository` remains unchanged. It e.g. always returns metaclasses in the form of `InterfaceDefs`, regardless of which model is used. However, the actual object that is returned may be an instance of `InterfaceDef` or a subclass (e.g. `X700_InterfaceDef`).
Adapters that retrieve metaclasses from the EIR must, of course, know the subclass definitions. That means that an X.700 adapter has to include the classes `X700_InterfaceDef`, `X700_AttributeDef`, `X700_OperationDef` etc. An adapter can always indicate in which database the EIR should search for meta-information since it knows which object model it serves. When retrieving meta-information from the EIR, all classes have to be narrowed (e.g. cast down) to their actual classes in order to manipulate them (e.g. `InterfaceDef` -> `X700_InterfaceDef`). This is no problem either, since the adapter can be sure that the classes really *are* of the actual type. Since an adapter includes the definitions of a specific database in the EIR, it must be recompiled whenever the *structure* of the meta-information changes. When for example the structure of a `COM_ConstantDef` is modified, the corresponding `COM_Adapter` class has to be recompiled as well.

3 GOMscript: An Interpreter for the Dynamic Object Model

In order to handle instances of the dynamic object model, an interpreter has been written which lets users create, access and delete instances either interactively or by running scripts. With the current set of adapters it is possible to handle CORBA [DSOM] and X.700 [CCI] instances.
GOMscript[16] has simple values such as numbers, booleans, strings and aggregate values such as structs, unions and lists. It has the usual control statements for repetition (`for`, `while`) and conditional branching (`if`, `else`) and is object-oriented in the sense that it implements (single) inheritance, polymorphism and encapsulation. Its core is very small and can be extended (additional functions and classes) through user-written *components* [Deri95] which are located in shared libraries.
GOMscript can be started in *daemon mode* in which it waits for (potentially remote) clients to send scripts to be executed. This allows for implementation of simple *roaming agents* [Mage96] which are essentially scripts moving from machine to machine and taking their state with them. The architecture is shown in Fig. 8 below:

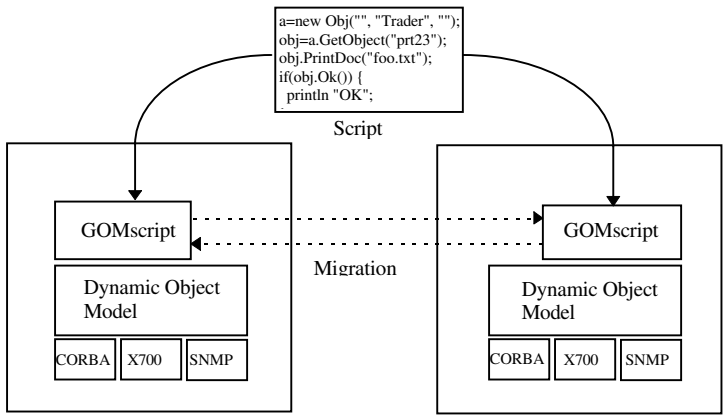

Figure 8: GOMscript as an Execution Platform for Roaming Agents

At any time during its execution, a script may decide to migrate to a different machine. To do so it simply calls a GOMscript function which has as parameters the name of the target host and the port number. This will pack the currently running script and the state (a dump of the symbol table) into a TCP message which is subsequently sent to the target machine. There, another interpreter receives the message, sets its initial state from the dumped symbol table data and starts execution of the script in a separate process.

[16] A prototype for AIX (3.2.5, 4.1.X) and Linux (2.0.0) is available at http://www.zurich.ibm.com/~bba/gomscript.html

Since CORBA *object references* are values in the symbol table that can be dumped to a data stream as well, it is possible for roaming agents to have access to some 'global' CORBA instance which they can refer to during their trips[17] . This allows for schemes where agents are sent to different locations to perform data collection and filtering tasks the results of which are periodically sent back to the central CORBA instance. A sample script that does this is shown below:

```
if(hosts == NULL) {  //only assigned initially since state is kept on migration
  hosts=new List("adl", "saz", "mut", "kis");
}

if(collector == NULL) {  // global CORBA instance (only created once)
  collector=new Obj("", "Collector", "adl");   // "adl" is central location
}

if(hosts.Length() > 0) {
  target_host=hosts.At(0);
  hosts.Delete(0); // otherwise we loop since the script is always sent to the same loc. !
  SendAgent(target_host, 10000, "", ""); // will know 'hosts' and 'collector' at target loc.
}


/* Now start the assigned task */
snmp_obj=new Obj("", "::Liaison::DSOMSNMPObj", "");  // create local SNMP obj
snmp_obj.SetSnmpAgentAddress("", 160);               // local snmpd is used
hostname=GetHostname();
while(true) {
  out_requests=snmp_obj.GetAttribute("ifOutRequests.0");
  // get more interesting data ...
  collector.UpdateAttr(hostname,           // primary key
                       "ifOutRequests.0",  // secondary key
                       out_requests);      // data
  Sleep(60); // sleep 1 min.
}
```

First the script is sent to a set of machines determined in `hosts`. As long as this list is not empty, the script will propagate a copy of itself (and its current state) to the next member of the list. Having done this, the assigned work can be started. An SNMP object is created and a value retrieved (`ifOutRequests.0`). This value is then sent to the central CORBA instance where it may be accessed by other clients for examination. Since values can be summarized, correlated or filtered by an agent before sending them to the global CORBA instance, a sort of event filtering mechanism can be easily implemented.

Since different sets of classes and functions may be available on various machines in the network, GOMscript allows to check for the availability of certain classes and functions before usage. Also, since GOMscript is based on the dynamic object model, it is possible for agents to dynamically find out what classes are available on a certain system and use their services. Consider the case of a `Printer` class which offers a service named `Print`: an agent can always invoke this service on *any* object that offers a service of the same signature irregardless of whether the class is derived from some common base class or the client (agent) has (compiled-in) knowledge of this specific class.

[17] Of course, only a *proxy* to the real instance is migrated. CORBA's `object_to_string` and `string_to_object` functions are used.

4 Conclusion and Outlook

The CORBA object model is gaining rapid acceptance for building distributed systems. However, existing (object) models such as SNMP and X.700 for network management and TINA and ANSA for distributed computing have to be integrated to take advantage of existing systems and code.
This paper tried to show that it is possible to use CORBA to access other object models by enhancing it with a dynamic object model, by creating adapters for bridging to other models and by modifying the IR towards a distributed, multi-model meta-information service which adapters can use for dynamic bridging. Having concepts such as attributes or methods as classes of their own enables flexible runtime creation of instances and increases the likelihood of easy integration of future object models. Basing the dynamic object model on dynamic adapters makes clients more independent from agent / server changes and offers a more abstract interface as well. The dynamic object model offers the ability to write generic clients e.g. for the purpose of management. These needn't include the specifications of all potential classes whose instances they might be managing at some future time, but can rather load the specification on demand (when used) contacting the EIR. This results in very small client applications (managers).
A prototype implementing a subset of the proposed architecture has been written at the IBM Zurich Research Laboratory [Ban95]. It is currently based on C++ rather than CORBA. The EIR is implemented as a UNIX daemon accessing an object-oriented database and using TCP/IP for communication. The database is populated / modified using a GDMO and an ASN.1 compiler that generate requests for a proprietary protocol based on TCP/IP. The CORBA IR is accessed directly to retrieve meta-information related to CORBA objects. Adapters are located in shared libraries, are loaded at startup-time and can be dynamically reloaded at runtime[18] . Two adapters are available, one for CORBA [DSOM] and one for X.700 [CCI].
Current work includes the porting of the prototype to CORBA according to the proposal. This basically means to define IDL interfaces for all C++ classes of the dynamic object model that are visible to clients. Also, adapters have to be defined as CORBA interfaces. The EIR will be revamped to become a CORBA service for multi-domain meta-information.

[18] The idea and implementation was influenced by the concept of *droplets* [Deri95]. This concept has turned out to be quite efficient for development turnaround-times; all the (sometimes) large libraries for specific object models need not be linked with the client, can be shared by multiple clients in the system and can be modified and reloaded on the fly which is especially useful in the development stage.

5 References

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

[Ban95] Ban, Bela: *The Generic Object Model: Towards An Object-Oriented Framework For Multi-Domain Management*. IBM Research Report RZ 2789. IBM Zurich Research Laboratory, Rueschlikon, 1995. http://www.zurich.ibm.com/~bba/GOM.ps

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

[CCI] Feridun, M., L. Heusler and R. Nielsen: *Implementing OSI Agents For TMN*. IBM Research Division, Zurich Research Laboratory. Rueschlikon, 1995.

[Challa] Challa, S. and D. Kafura: *Using Reflection for Implementing ICOM, An Interoperable Common Object Model*. Department of Computer Science, Virginia Tech. Blacksburg, VA, 1995.

[Deri95] Deri, Luca: *Droplets: Breaking Monolithic Applications Apart*. Internal Research Paper. IBM Zurich Research Laboratory. http://www.zurich.ibm.com/~lde

[DSOM] *SOM Developer's Toolkit: An Introductory Guide To The System Object Model And Its Accompanying Frameworks*. IBM, 1994.

[IR RFP] Digital Equipment Corporation, Hewlett-Packard Corporation, SunSoft Inc: *OMG RFP Submission. Interface Repository*. OMG TC Document 94-11-7, Version 1.0. Available at ftp.omg.org:/pub/docs.

[JIDM] *Translation of GDMO / ASN.1 Specification into Corba-IDL.* A Report of the Joint X/Open / NM Forum Inter-Domain Management Taskforce. Editor: S. Mazumdar, IBM T.J. Watson Research Center.

[Mage96] Magedanz, T. and T. Eckardt: *Mobile Software Agents: A New Paradigm for Telecommunications Management*. In: Proceedings of 2nd Intl. IEEE Workshop on Systems Management, Toronto, Canada, 1996.

[ODP] ITU-T X.901, ISO / IEC 10746-1: *Reference Model For Open Distributed Processing Part 1*.

[OMG] Object Management Group: *The Common Object Request Broker: Architecture And Specifcation*. Revised Edition, OMG document # 95-3-31, March 31, 1995.

[SNMP] *A Simple Network Management* Protocol. Internet RFC 1157

[Telefonica] Hierro, Juan: *Common Facilities for OSI Management*. Telefonica I+D, TMN OS Platform Division, 20.6.1995.

[TINA] Telecommunications Information Networking Architecture Consortium: *Overall Concepts and Principles of TINA*. Version 1.0, Feb. 17 1995.

[TMN++] Network Management Forum / OMNIpoint: *TMN++ Application Programming Interface*. Issue 1.0. Morristown NJ, Jan. 1996.

[X700] ITU-T X.720: *Information Technology Open Systems Interconnection - Structure Of Management Information: Management Information Model*.

6 Appendix A: Corba Classes

```
#include <somobj.idl>

module GOM {

  interface GenObj;
  interface ArgList;
  interface Adapter;
  interface Value;
  interface Attribute;

  interface Factory {
    GenObj   Create(in string  class_name,     /* Will be forwarded to an adapter */
                    in string  object_model,
                    in string  location,
                    in ArgList args);
  };


  interface GenObj {                           /* Proxy object class              */
    attribute Adapter             ad;
    attribute string              class_name;
    attribute string              instance_name;
    attribute sequence<Attribute> attributes;
    attribute sequence<string>    methods;
    attribute any                 specific_data;
    Value    Get(in string attribute_name);  // ad->Get(this, class_name, attribute_name)
    boolean  Set(in string attribute_name,
                 in Value new_value);
    Value    Call(in string method_name,
                  in ArgList args);
  };

  interface Adapter {
    GenObj   Create(in string class_name,     /* Creates an X.700 managed object */
                    in ArgList args);         /* args may e.g. contain the DN    */
    Value    Get(in GenObj objref,
                 in string class_name,
                 in string attribute_name);
    boolean  Set(in GenObj objref,
                 in string class_name,
                 in string attribute_name,
                 in Value new_value);
    Value    Call(in GenObj objref,
                  in string class_name,
                  in string method_name,
                  in ArgList args);
  };


 interface X700_Adapter : Adapter {
    /* override Create, Get, Set and Call */
  };


};
```