A Generic Management Model For Corba, Cmip and Snmp

Dissertation
der Wirtschaftswissenschaftlichen Fakultät
der Universität Zürich

zur Erlangung der Würde
eines Doktors der Informatik

vorgelegt von
Bela Ban
von
Kreuzlingen TG

genehmigt auf Antrag von
Prof. Dr. L. Richter
Prof. Dr. K. Bauknecht

DEZEMBER 1997

Die Wirtschaftswissenschaftliche Fakultät, Abteilung Informatik, gestattet hierdurch die Drucklegung der vorliegenden Dissertation, ohne damit zu den darin ausgesprochenen Anschauungen Stellung zu nehmen.

Zürich, den 10. Dezember 1997

Der Abteilungsvorsteher: Prof. Dr. L. Richter

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;
Then took the other, as just as fair,
And having perhaps the better claim,
Because it was grassy and wanted wear;
Though as far that the passing there
Had worn them really about the same,
And both that morning equally lay
In leaves no step had trodden black.
Oh, I kept the first for another day !
Yet knowing how way leads on to way,
I doubted if I should ever come back.
I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I –
I took the one less traveled by,
And that has made all the difference.

Robert Frost (1874–1963)
The Road Not Taken

Contents

List of Figures

List of Tables

Acknowledgments

My thanks go to the following people (in alphabetical order):

Bauknecht, Kurt	For being co-advisor for this thesis and for his support getting me started as a pre-doc.
Deri, Luca	For many interesting discussions, for sometimes having extreme opinions, for his invaluable help with SNMP and OSI-related issues, for always introducing new and exciting ideas, for trying and installing everything new on the planet, for proof-reading my thesis and correcting my English ...
Domenig, Marc	For getting me to learn CLOS and for being an excellent diploma thesis advisor.
Dreyer, Werner	For proof-reading and valuable comments.
Gantenbein, Dieter	For many interesting discussions and for proof-reading the thesis.
Genilloud, Guy	For interesting discussions during DSOM'95, ECOOP'96 and afterwards.
Kleinöder, Wolfgang	For providing me with the job at IBM.
Maffeis, Silvano	For the many discussions we had on Electra and CORBA in general, for always being interested in all sorts of things and for giving me the tip about Cornell ...
Niksch, Michael	For his guru-style advise on things ranging from AFS cache problems to questions about the firewall etc.
Pavka, Lilli	For very detailed proof-reading.
Rageth, Beat	For his willingness to help with problems with the infrastructure at IfI since my diploma thesis.
Richter, Lutz	For his support in being the main advisor for this thesis, for his willingness to hold frequent meetings with me and for his advise on many aspects of the 'famous' *Promotionsordnung*.
Riviere, Anne-Isabelle	For valuable comments on the draft version.
Schürfeld, Ute	For the long discussions on DSOM/OSI integration.
Steiger, Patrick	For proof-reading and valuable comments.

Most of all I would like to thank **Janet**: for her patience with the 'perpetual student', for supporting me the entire time, and for the past 8 years !

Abstract

The predominant models to perform network and systems management have traditionally been SNMP and CMIP. However, SNMP is beginning to reach its limits when more complex management tasks are to be performed, and CMIP – owing to its complexity and slow standardization process – has not yet gained the same acceptance as SNMP.
The advent of object-oriented distributed processing models has brought forward a third candidate, CORBA, originally not targeted specifically at management tasks, but in many respects nevertheless suitable for managing both local and wide area networks.
CORBA is more powerful than SNMP and less complex than CMIP. Its affiliation with C++, a widely used programming language, lends it to immediate use by a huge number of programmers and allows them to introduce distribution into their programs without a too drastic change of philosophy.
It is therefore assumed that CORBA will become important in the network and systems management domain as well as in the distributed systems domain. To be more precise, CORBA will be used to implement management applications (managers) and managed entities (agents).
The author assumes that, in the near future, all models will have to coexist, because investments in older models have been made, because CORBA may not yet be ready for certain specialized management tasks (e.g. embedded management agents), or simply for political reasons.
Therefore, the need arises to manage a system written in *one* model from a system written in a *different model*, for example to manage an OSI agent using a CORBA-based manager. Assuming that CORBA will achieve broad acceptance in the management world, it would be desirable to be able to manage other models transparently from CORBA. The benefits would be investment protection of existing managed entities, the opening of the SNMP- and CMIP-dominated management world to management-inexperienced (CORBA) programmers, and the unification of management in one common, simple, and yet powerful model.
The goal of this work is to examine how CORBA can be used for network and systems management. The focus is on the client side; that is, how CORBA can be used to implement management applications that *access* managed entities, rather than how CORBA can be used to *implement* managed entities.
Work in this area already exists; an overview will be given and it will be shown that most work focuses on *compile-time static translation* of one model to another. This generates a number of problems.
Therefore a *dynamic runtime-based* approach is proposed that eliminates these problems and has several advantages over static approaches. The proposed model is a combination

of a generic object model, metadata, and adapters that convert between the generic and specific target models. Although the elements of the proposed models are established and well known, their combination and their application to the domain of management is novel and makes for the issues of interest in this thesis.

Zusammenfassung

Im Netz- und Systemmanagement werden traditionellerweise SNMP und CMIP eingesetzt. Es häufen sich jedoch die Anzeichen dafür, dass SNMP langsam an seine Grenzen stösst, und CMIP hat – aufgrund seiner Komplexität und des langwierigen Standardisierungsprozesses – bisher noch nicht die gleiche Akzeptanz wie SNMP gefunden.
Die Verbreitung von objekt-orientierten verteilten Objektmodellen lässt einen dritten Kandidaten, nämlich CORBA, erkennen, welcher ursprünglich eigentlich nicht zum Zwecke des Netzmanagements entwickelt wurde, aber trotzdem in vieler Hinsicht ideal für das Management von lokalen und Weitverkehrsnetzen geeignet scheint.
CORBA ist mächtiger als SNMP und weniger komplex als CMIP. Seine Nähe zu C++, einer weit verbreiteten Programmiersprache, öffnet es gegenüber einer grossen Anzahl von Programmierern, welche ihre Applikationen auf CORBA-Basis auf relativ einfache Art und Weise um den Verteilungs-Aspekt erweitern können, ohne eine wesentlich andere Philosophie lernen zu müssen.
Die vorliegende Arbeit basiert deshalb auf der Annahme, dass CORBA zunehmende Bedeutung im Netz- und Systemmanagement erlangen wird, welche es im Bereich Verteilte Systeme schon hat. Präziser gesagt: CORBA wird eingesetzt werden, um *Management*- (Manager-Rolle) und *Managed*-Applikationen (Agenten-Rolle) zu entwickeln.
Der Autor nimmt an, dass mittelfristig alle drei Modelle koexistieren werden, sei es, weil bereits eine beträchtliche Menge von Mitteln in die existierenden Modelle investiert wurde, sei es, dass andererseits CORBA noch nicht für alle anfallenden spezialisierten Managementaufgaben bereit ist, oder dass Unternehmen einfach aus politischen Gründen (noch) nicht ausschliesslich auf die Karte CORBA setzen wollen.
Deshalb stellt sich die Frage, wie Systeme, welche in *einem* Modell entwickelt wurden, durch ein *anderes* verwaltet werden können. Basierend auf der Annahme, dass CORBA eine grössere Durchdringung der Managementwelt gelingen wird, wäre es wünschenswert, andere Modelle ebenfalls transparent via CORBA zu manipulieren. Vorteile wären Investitionsschutz bestehender Agenten, das sich Öffnen der SNMP- und CMIP-dominierten Managementwelt gegenüber Management-unerfahrenen (CORBA) Programmierern und die Zusammenführung des Managements unter ein gemeinsames, einfaches und dennoch mächtiges Modell.
Das Ziel der vorliegenden Arbeit ist die Beantwortung der Frage, wie CORBA im Netz- und Systemmanagement eingesetzt werden kann. Ich beschränke mich dabei auf die Manager-Seite; d.h. wie CORBA eingesetzt werden kann, um Managementapplikationen zu entwickeln, welche auf unterschiedliche Agenten *zugreifen*, und nicht, wie Agentenfunktionalität auf Basis von CORBA entwickelt werden kann.
Nach einem Überblick über die Forschung auf diesem Gebiet werde ich zeigen, dass die

meisten Ansätze eine *statische* Überführung eines Modells in ein anderes wählen, ein Vorgehen, das einige Probleme aufwirft.
Deshalb schlage ich einen *dynamischen* Ansatz vor, welcher einige dieser Probleme löst und ausserdem gegenüber der statischen Vorgehensweise gewisse Vorteile beinhaltet. Er besteht aus einem generischen Objektmodell, Metadaten und Adaptern, welche zwischen dem generischen und jeweils einem spezifischen Zielmodell konvertieren. Die verwendeten Elemente sind bekannt; jedoch ist es ihre Kombination im Gebiet Netz- und Systemmanagement, welche den Beitrag dieser Arbeit ausmacht und interessante Fragestellungen aufwirft.

Chapter 1

Introduction

1.1 Motivation

Prior to the seventies, computing was performed primarily on mainframes with proprietary operating systems and centralized administration. Attached to the mainframes were dumb terminals, which possessed no computational capability other than the firmware code that initially connected them to the mainframe, and their operation and management was performed from a central place.
With the rise of the UNIX (formerly: MULTICS) operating system in the early seventies and the personal computer in the eighties, decentralization of computation ensued, and computers were interconnected in the form of local area networks (LANs) and wide area networks (WANs) [Tan92, Bac86]. Parallel to the decentralization of computational resources, their management had to become decentralized as well. At that time, various proprietary schemes existed or were created to perform management of the computing infrastructure (e.g. TL1, Telnet etc.).
In 1978, the International Standards Organization (ISO) began work to establish a standard for network management, the Open Systems Interconnection (OSI) Network Management standards, intended to create a common management protocol, the Common Management Information Protocol (CMIP) [CMI].
Because of the slowness of the standardization process, the complexity of the proposed new standard, and the urgent need for management tools, the Simple Network Management Protocol (SNMP) [CFSD90] was devised. It was originally regarded as a provisional means for management until the OSI standard would be completed, but subsequently became a de-facto standard because of its dissemination, partly due to its simplicity. Today, although SNMP is predominantly used, it is beginning to reach its limits. Therefore a new version (SNMPv2) has been created [CMRW96], but its deployment has been rather low so far [Ros90a].
For historical reasons there has always been a division between tools for *operating* (e.g. TCP/IP, RPC) and *managing* (e.g. SNMP, CMIP) a networked system. With the advent of object-oriented distributed computing models (CORBA [OMG95], ANSA [ANS93], Java [RMI96], DCOM [BK96]), there are efforts to make the operational and management 'faces' the same, i.e. to manage *and* operate the network using the same model. Moreover, *distributed* systems management tasks cannot themselves be centralized, but

require *distribution of management* as well. In this respect, a distributed computing model such as CORBA can be considered a *distribution enabler* for management tasks.
There are indications that CORBA is becoming an important technology for implementing both management- and managed entities. The success of Tivoli's management framework [Tiv95], which is based entirely on CORBA, is a strong indicator of this. Genilloud [Gen96] reinforces this argument by stating that "distributed objects are thus the paradigm that will allow systems management to converge towards one single, homogeneous and global management architecture".
Still, there exist a variety of legacy systems for management purposes. Especially companies involved in telecommunications and carrier business have invested significantly in OSI network management standards and are therefore unlikely to move towards emerging new technologies such as CORBA without preserving their existing investments. It is therefore unlikely that there will be a *single* predominant management model in the near future. For companies having a heterogeneous system environment this means that resources represented through a number of differing object models will have to be managed.
Assuming that CORBA[1] will achieve increasing prominence in the network and systems management business, without, however, fully dominating it, we can expect to see an increasing need to build and employ *bridges* between CORBA and other models, e.g. to manage legacy systems from CORBA and in turn to access CORBA from legacy systems. Bridges are syntactic and semantic translators that allow applications in one model to access another model as if the latter were of the same model, thus hiding differences between object models.[2]
The present work focuses on how a heterogeneous system environment can be managed from CORBA-based applications.
A number of approaches exist for managing OSI- or SNMP systems from CORBA. An overview of these will be given in chapter 3.
This thesis proposes a novel approach called *Generic Object Model (GOM)*[3] for CORBA-based network management consisting of a *metadata repository* as its central piece, a *generic object model* capable of representing *any* target objects, and *adapters* constituting bridges between generic and specific target systems.
The expected advantages of such a model can be summarized as follows:

Uniform Programming Model A uniform programming model allows transparent manipulation of a *set* of object models without having to deal with the underlying target model. This would enable new systems built using CORBA to integrate and make use of existing (and tested) legacy SNMP and CMIP building blocks without the need for re-implementation of functionality, thereby saving a considerable amount of time, cost, and knowledge. This scheme avoids the need for a complete re-writing of legacy systems, but

[1] The reason for choosing CORBA is given in section 1.3.7.

[2] The concept of CORBA's *inter-ORB bridges*, which convert requests between different ORBs, is discussed in section 2.1.4.

[3] The model is called GOM after one of its three main components, the generic object model (cf. section 4.2). To refer to that particular component, the term *generic object model* (in lowercase letters) is used; to refer to the entire proposal, the capitalized version *Generic Object Model* (or its acronym, GOM) is used.

instead allows immediate use and integration in the new system with the future option of gradual replacement by new, CORBA-based components.

Dynamic Aspect There are also benefits for applications that cannot predict at compile time the extent of classes they will encounter, instances of which they have to handle. This could be especially important in the area of mobile agent applications [ME96] where an agent travels from machine to machine and has to manipulate instances of classes that it did not know about when it was compiled (cf. section 5.2).
GOM allows the dynamic discovery of classes at runtime and enables management applications that take advantage of this capability to manipulate instances in a dynamic manner.

Small Clients Because of size limitations it is not feasible for an agent to have compiled-in static knowledge of a large number of classes. It is more economic both in terms of size and speed to send small agents around in the network rather than large ones (cf. section 5.2) [MGG96].
GOM allows a client to contain initially no knowledge of classes of the target system to manipulate. When starting to manage the target system through GOM, classes that are not yet known are dynamically loaded into GOM's runtime environment, enabling client applications to grow dynamically (in terms of size) with the number of classes they need to know in order to perform management tasks.

The structure of the thesis is as follows: the remainder of this chapter will define common terms and concepts. Then we will examine several aspects of computational systems that are important for this work. Also, the scope of the thesis will be identified. The goals pursued, the contribution, as well as underlying concepts and prerequisites will be discussed at the end of the chapter.
In chapter 2, an overview of the prospective *target object models* (CORBA, CMIP and SNMP) to be managed from the generic model is given. No in-depth treatment of the three models will be given, but the objective is rather to make the reader aware of the similarities, but also the differences, between the various models.
The objective of chapter 3 is threefold: first the author presents research done in the area of inter-domain management between CORBA, CMIP and SNMP. It will be shown that most approaches deal with static compile-time translation and therefore have certain deficiencies, which will be described. Second, the scope of this work will be further restricted by precisely defining on which area of inter-domain management the work focuses. Finally, one example of work on inter-domain management will be presented in more detail.
Chapter 4 is the main chapter and will present the author's model of CORBA-based dynamic network management. It will be shown that using a dynamic, metadata-based model has certain benefits over static compile-time based approaches. After giving an overview of the architecture, the three main components will be described: (1) the generic object model, which is a reified object model capable of handling any target object model without need to be extended, (2) the metadata repository, central to the generic model as it contains metadata descriptions of all target models to be managed and (3) adapters

which perform conversions between the generic and the target models. A section on a generic approach for event handling complements the generic object model and finally, some issues that are relevant, but not discussed in-depth, will be presented.

Chapter 5 validates the ideas proposed in this thesis. It discusses two applications of the proposed model, an interpreter and a toolkit for roaming agents. Both are dependent on the dynamic, runtime-based features offered by GOM and could not have been implemented using static compile-time based approaches.

Chapter 6 will summarize the proposed model, compare the results to the goals initially set and to the related work, and draw conclusions. The work will be completed with an outlook into an area in which dynamic models such as the one presented might have an impact in the future, namely the World Wide Web.

1.2 Definitions

This section defines terms and concepts commonly used throughout this thesis. All terms and abbreviations will also be listed in the glossary (p. 189).

An *object model* is essentially the description of a system by means of objects and their interactions. Information (data) is represented by objects and information flow is achieved by interaction between objects.

An *object* is a collection of operations sharing a common state which is hidden from the outside and can be accessed/modified only by invocation of the object's operations. The set of operations offered by an object is called its *interface* [Weg90].[4]

A *class* is a template from which objects (instances) can be created. It contains the definition of operations and a number of instance variables. When an object is created from a class, it receives its own set of instance variables according to the class' definition. Operations, however, are shared by all instances of a class and are therefore conceptually located in the class.

A *type* is a mechanism for classifying values into categories of common properties and is used, among other things, for type- checking to enforce syntactic constraints on expressions in order to ensure operator/operand compatibility. The main difference between types and classes is that types are specified by predicates over expressions that classify values into categories, whereas classes are specified by templates used for generating instances with uniform properties and behavior. Every class is a type, defined by a predicate that specifies its template. However, not every type is a class, since predicates do not necessarily determine object templates [Weg90].

The difference between an *information model* and an *object model* is the abstraction level used. An information model deals with information (data) and the flow of information in a system, whereas an object model defines information to be in the form of objects, thus being more concrete in terms of modeling. An information model may also be expressed using a structured (functional) model [ODP95].

One of GOM's goals is to manage a number of heterogeneous systems from its generic model. As will be explained in section 4.2, for each instance in the system to be managed,

[4] To prevent a name clash between CORBA interfaces, which are classes in object-oriented parlance, the terms *IDL-* or *CORBA interfaces* are used in ambiguous cases.

a corresponding *proxy instance* will be created on the GOM side. The system to be managed is called *target system* (or *target model*) and its instances will be called *target instances*.
A *management*- or *manager* system denotes the entity that performs management while a *managed* system is the entity on which management is performed. The terms *manager* and *client* on the one hand and *agent* and *server* are used synonymously.

1.3 Scope

The goal of this section is to present certain aspects of computational systems that are relevant for this work and to show how these affect and/or contribute to the topic. For each aspect, the non-relevant parts will be excluded from the scope of this thesis.

1.3.1 Distributed vs. Non-Distributed Paradigm

An important aspect of computational systems is whether (or to what extent) they are interconnected. Non-distributed, centralized systems are easier to maintain than their distributed counterparts because they do not have to deal with a variety of operating system architectures, protocols and data formats. Distributed systems, however, have a better cost-performance ratio, scale better, and are more flexible and reliable than centralized, single-point-of-failure mainframes. Distributed systems also make certain aspects of computing transparent. *Location transparency* means that users of such systems deal with computational entities regardless of where the latter are located physically and without the need to know. *Migration transparency* allows computational entities to be moved to different physical locations without their clients being aware of it. *Concurrency transparency* allows many clients to access the same entities without being aware of the other users in the system and *replication transparency* allows to have multiple replicas[5] of an entity in a transparent way for the user [Tan92].
This work is concerned with how (object) models for management and / or operation of distributed systems (CORBA, CMIP, SNMP) can be handled from a single generic model and therefore does not focus on centralized, non-distributed systems.

1.3.2 Protocol-Centric vs. Interface-Centric

The client view of a distributed system may be on the protocol level (CMIP, SNMP) or the protocol may be hidden and only *object interfaces* (cf. 2.1.2) may be exposed to the client. The latter approach defines objects regardless of the underlying protocol whereas the former defines a protocol that clients have to observe in order to communicate with a peer entity.
Of course, interacting with a remote object through its interface will make use of a protocol to send the request, but this fact is hidden from the client, thus offering a higher level of abstraction.

[5] E.g. for performance improvement or security reasons.

The *generic object model* presented in this work is affected in two ways by this aspect: first, it will encounter object models of both aspects such as the protocol-centric CMIP and SNMP models on the one hand and interface-centric models such as CORBA on the other hand. It will have to be able to handle all of the above models (and more) regardless of whether a model is protocol- or interface-centric. Second, the generic object model will *itself* have to expose an API to the client. This API will be interface-centric, which means that clients do not have to be concerned about protocols, but see only computational objects of the elements constituting the generic object model.

1.3.3 Procedural vs. Object-Oriented Model

Procedural (or structured) programming makes a clear separation between the program logic (functionality, code) and state (structure, data). With the advent of object-oriented programming, this separation was eliminated and data and code were encapsulated to form a computational unit. Major features of object-oriented programming are [Mey88]:

Encapsulation Data and code are comprised into a unit of computation (i.e. an object). Data within the object is shielded from the outside and can only be modified by invoking operations provided by the object at a (public) interface.

Inheritance Objects can extend other objects by inheriting their data and code and by adding new data or operations or by modifying existing ones. Inheriting behavior and state is called *implementation inheritance* while inheriting the protocol (interface) dictated by the superclass is called *interface inheritance*.

Polymorphism Depending on the type of the object the same message can produce different behavior.

The focus of this work will be on object-oriented models (CORBA, CMIP), but will also take into account non-OO systems such as SNMP because of its wide dissemination in the management domain. The API of the generic object model exposed to clients will be in the form of *object interfaces* (cf. 2.1.2).

1.3.4 Dynamic vs. Static Functionality

Functionality in a distributed system can be static or dynamic with regard to its location. A CORBA-based system is static in the sense that functionality is offered to clients in the form of services exposed by implementation objects located in a server. Although implementation objects may be located in multiple servers and may even be migrated between servers, they are passive and migration must be performed, for example, by a system administrator.

Dynamic functionality, on the contrary, is active (roaming) in the sense that it may decide itself to migrate to a different location. The concept of *agents* [ME96] or *mobile code* for example has dynamic functionality.

1.3.4.1 Dynamic Code Paradigm

Code is sent from one location to another to be executed. Examples are Java *applets* [Sun95] or TeleScript *intelligent agents* [Whi94].
A Java applet is essentially byte code for a *virtual machine* [ASU86, Kam90] that is sent to a location running a virtual machine interpreter, which then executes it. Thus, code is always sent from a server to a client.
In TeleScript, however, intelligent agents may be sent in both directions since it is the agent which decides where to travel.

1.3.4.2 Static Code Paradigm

Static code is functionality that is in a certain location (e.g. in a CORBA server) and can be accessed by clients using proxies (method-call-forwarding substitutes for the real remote objects).
Static and dynamic functionality can complement each other profitably. Taking the example of Java, is does not seem sensible to send large amounts of code (e.g. for a database server) to the client due to latency and economic reasons. However, it does make sense to send code for example for an intelligent form that has to be filled in by a client and that subsequently sends itself to a certain location (e.g. back to where it came from). A potential application of Java applets might be in the area of (portable) graphical user interfaces as front end to static functionality. An example is a CORBA server that implements a database management system. Rather than sending the entire DBMS code to the client, it makes more sense to send down a Java applet that implements the GUI functionality which subsequently accesses the CORBA DBMS as a CORBA client.
There are some projects that deal with implementing CORBA access from Java via IIOP [Joe96, IBM96, Sat96, Ion96].
In this thesis, the author is concerned mainly with the management of *static* functionality, i.e. with the creation and manipulation of instances located in servers[6] via proxies in the client's address space.
Research into how the dynamic code paradigm can be exploited in the management domain is underway, but is beyond the scope of the thesis.

1.3.5 Operation vs. Management

The model used to implement a system is often different from the one used to manage it. A distributed system may for example be implemented using remote procedure calls (RPC) but be managed using SNMP or CMIP agents. Recently, with the advent of object-oriented distributed systems such as CORBA, operations (i.e., implementation) and management models tend to become the same, e.g. CORBA may be used to implement *and* manage a system. This may be achieved by adding CORBA objects to the system whose task is to manage other objects, or by adding operations to existing CORBA interfaces to make them manageable. In the TINA [TIN95] model, management

[6] The term *server* is used here in the sense of *a location that implements object interfaces* and is not constrained to CORBA servers.

capabilities are usually injected into an object by adding specific interfaces to the object which are used exclusively for management purposes.
Here, we assume that object-oriented distributed processing (OODP) systems are increasingly used to operate and manage DP systems and will therefore be based on such concepts (CORBA) while at the same time offering capabilities for integrating pure management systems (SNMP, CMIP).
As the motivation for this thesis comes from the management world, the focus is on management. However, nothing precludes the model from being used to create distributed systems.

1.3.6 Interface vs. Implementation

Languages for building OODP systems can be classified into two main categories: pure *interface description languages* and languages that allow both *description* and *implementation* of distributed systems.
Interface description languages typically specify a system by defining a number of classes, the services they offer (operations), and their instance variables (having a certain type). These languages are not computationally complete in that they do not offer constructs for implementation of the specification. Typically, a specification written in an interface specification language will be translated into an implementation language for implementation.
Advantages of pure specification languages are that they are language-independent (any language can be used for implementation) and that the division between interface definition and implementation may account for a cleaner design without too much regard for implementation details. A disadvantage is that they have to be mapped to an implementation language which may involve conversion / loss of information.
Examples of interface specification languages are remote procedure calls [BN84], CORBA Interface Definition Language (IDL) [OMG95], GDMO and ASN.1 [GDM92, ASN90] and TINA ODL [TIN95].
An implementation language is computationally complete and can be used for both system specification *and* implementation. Examples are Smalltalk [GR89], C++ [Str91] or Java [Sun95].
Many object models for distributed systems are based on specification languages to define the system interfaces and then perform a mapping to an implementation language.
In this thesis, the focus will be on *interface description languages* such as OMG IDL or GDMO.

1.3.7 CORBA vs. DCOM vs. Java

Sun's *Java* [Sun95] language is similar in syntax to C++. Programs are compiled into byte-code interpreted by a virtual machine (VM). Porting the virtual machine to a number of architectures enables Java programs to run unmodified on every platform for which a VM exists. The same mechanism also allows Java code to be sent across the network to a different machine, which subsequently executes the byte-code (Java *applets*).

Sun has recently added *Remote Method Invocation (RMI)* [RMI96] to the language, enabling objects on one machine to invoke methods of objects on remote machines. Whether a method call is local or remote is transparent to the programmer.
Microsoft's *Distributed Component Object Model (DCOM)* [BK96] is another model for writing distributed applications. It has gained a strong foothold especially in the PC market as it is part of the Windows95 and WinNT operating systems. DCOM has only recently (1996) evolved from a non-distributed model (COM).
Both Java and DCOM are currently proprietary models developed solely by single companies, whereas CORBA is a standard. However, the author believes that both DCOM and Java will achieve wide dissemination and will compete with CORBA.
The reason for choosing CORBA as distribution mechanism of this work is a pragmatic one: both Java/RMI and DCOM were not yet available when this work started.
However, it is the author's belief that the model underlying all of these three object-oriented distributed processing architectures is fundamentally the same and therefore the ideas presented in this thesis could be applied equally well to Java or DCOM.

1.4 Goals

Network management systems have traditionally used SNMP or CMIP to manage resources. With the advent of object-oriented distributed systems such as CORBA it is envisaged that an increasing part of management tasks will taken over by CORBA-based management applications. These require either *managed* entities to be *CORBA-based* as well, or, if these are not available, e.g. if written in SNMP or CMIP, need *bridges* that convert between the CORBA and SNMP or CMIP models respectively. The advantage of using bridges to access existing managed systems of heterogeneous models would be that only CORBA needs to be used on the client side while still being able to access various other models on the server side. An important additional argument is that time, money and know-how invested in building managed systems using SNMP and CMIP is not lost.

> The major goal of this thesis is therefore to propose a common CORBA-based network management integration model for CORBA, CMIP and SNMP exploiting dynamic runtime- and metadata-based mechanisms to access target models.

Maybe surprising at first glance, the *CORBA* model is also among the target models to be managed. However, the author assumes that in the future a significant number of *managed entities* (e.g. agents) will be written using CORBA. Thus the need arises to manage CORBA as well.
The benefits of such a model are

1. A uniform programming model that hides the differences between heterogeneous managed systems,

2. Greater flexibility, thus reducing client-server dependencies, and

3. Lower memory requirements for client management applications.

Work done in the area of bridging between CORBA management applications and SNMP- and CMIP-based managed resources will be presented in chapter 3. Most of this work employs *static* translation of object models to CORBA, bringing with it certain disadvantages such as insufficient flexibility (strong interdependencies between management- and managed applications through inclusion of translation-generated code), bloated management applications and problems mapping certain idiosyncrasies of CMIP due to limitations of the compile time translation approach.
In chapter 4, a novel approach will be presented (GOM). It in fact comprises well-known concepts such as metadata, a generic object model, and adapters, but these have not yet been applied in this combination to the area of CORBA-based management. It will be shown that some of the disadvantages of static approaches can be eliminated using this model.
The focus of this work is on management aspects, but nothing precludes the model and concepts presented from being applied to the broader area of distributed applications in general. As management of a distributed environment is distributed by definition, concepts of distributed computing apply equally well to distributed management.

The contribution made by this thesis consists of

1. A proposal of the Generic Object Model (GOM), which is a novel model for CORBA-based management based on *existing and well-known* concepts such as metadata, a generic reified object model and adapters, but
   (a) applied in a novel combination
   (b) and to an area (network management) in which they have not been used before.
2. A prototype validating the ideas of GOM.

This thesis provides a model for CORBA-based management applications to transparently handle a number of heterogeneous target systems such as managed entities implemented in CORBA, CMIP or SNMP. It also provides an extensible mechanism through which other (management) models may be integrated.

1.4.1 Concepts

This work is based on some of the major concepts assumed to be present in an object model to be integrated into the generic model. Absence of one or all of the concepts in a target model does not preclude the model's inclusion, but integration will be easier if the following concepts are present:

1. Concept of distributed information represented by distributed objects (not necessarily of the same object model). Any information in a system is modeled as an instance of a class (object).

2. Concept of always available description of objects (metadata), e.g. replicated on every machine. As described in 1.4.2, meta information of the classes available in a system is essential for the functioning of the generic object model.

3. Concept of handle (local proxy) to an object that finds the corresponding target object wherever the latter may be located. The assumption is that in most distributed systems clients receive a local handle – called a *proxy* [Sha86] – to a remote object. Any operation invoked on the proxy will be forwarded to the *real* or *target* instance; clients need not care whether the target instance is local or remote (distribution and location transparency).

1.4.2 Prerequisites

As will be discussed in detail in chapter 4 there are a few requirements on an object model that is to be integrated into the generic model.
The first requirement is that there is runtime information available about the classes and their attributes and operations which *adapters* (cf. 4.4) can query. This requires some transformation of a model specification into an electronic form suitable to be maintained by a *type repository* (cf. 4.3). Typically this transformation is performed by a translator/compiler.
Second, the model must offer certain dynamic capabilities such as creation of instances given the class name (as a string) and invocation of operations on instances given the name of the operation. Some models already offer such an API, e.g. CMIP [ITU92a], CORBA's DII or IIOP [OMG95], COM's IDispatch interface [Box95] and Java's Core Reflection API [Sun96].

Chapter 2

Target Object Models

This section will present the three target (object) models dealt with in this thesis: CORBA, CMIP and SNMP.[1]
Its objective is first to familiarize the reader with the models (and their idiosyncrasies) that need to be managed from GOM, and second, to enumerate commonalities and differences between them to foreshadow problems to be overcome when defining a generic model that integrates all three target models.
A basic knowledge of CORBA, CMIP and SNMP is helpful because the concepts underlying these models will be explained only briefly.

2.1 CORBA

2.1.1 Architecture

OMG's Common Object Request Broker [OMG95] specifies the architecture by which instances in a heterogeneous environment can communicate with one another regardless of whether they are local or remote. The architecture is shown in fig. 2.1.

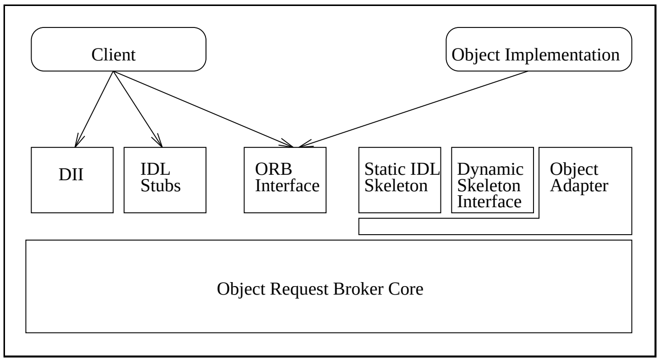

Figure 2.1: The CORBA architecture

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* (cf. 2.1.2) seen by the client is completely independent of where the object is located or in which programming language it is implemented. 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

[1]SNMP is an exception because, although it is not exactly an object model, it is included in the discussion because of its usefulness as an example of how non object-oriented systems can be mapped to object-oriented ones, and because of its dissemination in the management domain.

needed for example by the ORB to marshal and unmarshal[2] requests to / from clients and the Implementation Repository is needed for locating the implementation to invoke requests on instances.

Clients call methods of instances using either compiler-generated client stubs, which have to be included at compile-time, or the *Dynamic Invocation Interface (DII)*, which allows clients to create requests to be sent to instances at runtime. *Implementation skeletons* are compiler-generated implementations of the services offered by an interface, enriched with programmer-added functionality. They are usually called by a server to handle 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.[3] Any method invoked on an object reference will be forwarded to the remote instance and the result will be returned to the client.

The CORBA architecture includes a set of common object services (CORBAservices [COS95]), which are reusable services such as Naming, Event, Persistence etc.

2.1.2 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 only through a well-defined interface. Objects interact by 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 [OMG95]. 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.

2.1.3 Specification Language

Interfaces are specified using the Interface Definition Language (IDL). IDL allows to define interfaces, together with their attributes and operations. However, it cannot be used to *implement* an interface. For this purpose, IDL code has to be translated into an implementation language (e.g. C++, Smalltalk), which can subsequently be used to complete the implementation. IDL was designed to be used only as specification language to allow multiple implementation languages to be used, resulting in language independence. It is therefore possible for a client to access an object from a language

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

[3]Object references are *reliable* because they are never reused. An object reference will always point to the object for which it was created or, if the object has been deleted, to a NIL object.

different than the one in which it was implemented as long as a *language binding*[4] is available.
An IDL example is given in fig. 2.2:

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

Figure 2.2: 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 two services, `Print`, which takes two 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 accepts IDL code and generates client stubs and implementation skeletons for the specified language binding (cf. fig. 2.1). Client stubs do not 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[5].

2.1.4 Inter-ORB Bridging

This section will discuss the concept of *inter-ORB bridges* [OMG95, ch. 11] and how they relate to adapters as defined in this work.
Inter-ORB bridges have the task of converting requests between ORBs of different vendors. They can also be used to control access to ORBs within different domains, such as security (e.g. firewall functionality [CB94]) or accounting domains.
Inter-ORB bridges are classified into *in-line* and *request-level* bridges. In-line bridges are part of the ORB core proper, whereas request-level bridging is performed by application code outside the ORB.
Request-level bridges can be further classified into *interface-specific* and *generic* bridges. Interface-specific bridges support a finite number of IDL interfaces that were included when the bridges were compiled. Code for conversion is usually generated by the IDL compiler.
Generic bridges can be used to convert all sorts of requests without being tied to individual IDL interfaces since they are built using CORBA's *Dynamic Skeleton Interface (DSI)*,

[4] A translation from IDL to an implementation language.
[5] 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)

Dynamic Invocation Interface (DII) and *Interface Repository (IR)*. The DSI receives requests and the DII – with the help of metadata stored in the IR – dispatches them to instances in other ORBs.
Rather than translating requests between different ORBs, bridges can also be used to translate requests between ORBs and other non-CORBA systems such as SNMP [CFSD90], CMIP [CMI], COM [Bro94] etc. [OMG95, 9.1.2].

CORBA bridges are located at the server side and used to *implement* functionality, whereas GOM is at the client side and used to *access* functionality. CORBA request-level bridges could for example be used profitably to implement XoJIDM's *interaction translation* approach (see 3.1.2). However, using XoJIDM's approach, the client would still 'see' all of the CORBA interfaces generated by their approach, regardless of whether CORBA bridges are used at the server implementation side.
The model proposed in this thesis could be used to *implement* generic request-level bridges since the main components needed for building these (DII and IR) are provided by GOM. As a matter of fact, using GOM in a generic request-level bridge would even go a step further: requests could be bridged between the models of CORBA *in addition to those of SNMP and CMIP*.

2.2 OSI Network Management (CMIP)

2.2.1 Architecture

OSI Network Management[6] has been defined by the joint ISO / ITU-T standard documents X.7xx[7] [ITU92a, ITU92b, ITU92c].
The approach proposed by X.700[8] is to place an *agent* close to the information that is relevant to be managed and / or monitored (cf. fig. 2.3).
An *agent* maintains information (in the form of *managed object instances*) about the state of a particular part of the network for which it is responsible. Queries about or changes of the state of the network can be sent only to the agent, not directly to the objects. The protocol used to interact with the agent is the Common Management Information Protocol (*CMIP*) [CMI].

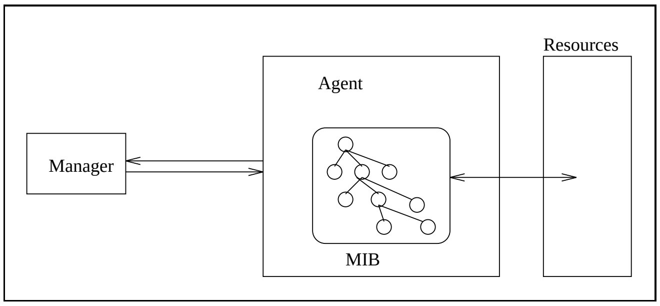

Figure 2.3: Manager-Agent paradigm

A *manager* 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 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

[6]The term *CMIP* will be used synonymously.
[7]An overview is given in the ISO / IEC documents 7498-4 (X.700) (functional areas) and ISO / IEC 10040 (X.701) (agent-manager architecture). The information model is defined in ISO / IEC 10165-1 (X.720).

the agent. M-SET changes information in an agent whereas M-GET retrieves information from an agent. As 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 is not powerful enough, it is possible to specify operations on managed objects (cf. below) which can be called using M-ACTION. If an error occurs in the network, an agent may at any time send an unsolicited message to the manager concerning the cause of the error by using the M-EVENT-REPORT request.

2.2.2 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 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. Notifications are messages that can be sent from the managed object to its agent which then forwards them to any manager that is interested in receiving them. CMIP requests that can be sent to an agent map directly to methods applied to managed objects that the agent contains. When, for example an M-GET request is sent to the agent, the agent will first find the managed object to which the request is directed, then retrieve the information from the managed object and finally return it to the sender.

2.2.3 Specification Language

Managed objects are defined using two specification languages: *Abstract Syntax Notation One* (ASN.1 [ASN90]) is used to define data types and *Guidelines for the Definition of Managed Objects* (GDMO [GDM92]) to define managed objects.

The type of attributes or operation parameters contained within managed objects is defined using ASN.1, which 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. 2.4.

The example defines 3 types: `BaseManagedObject` is a struct which contains 2 members, namely an `ObjectClass` (another ASN.1 type defined below) called `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 [GDM92] notation. An example of GDMO is given in fig. 2.5.

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

```
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}
```

Figure 2.4: ASN.1 example

notifications) must be present in an instance of the managed object class, whereas the presence of conditional packages is determined at runtime depending on the specified behavior.[9] This implies that although two instances may be of the same class, they may not have the same number of attributes, actions or notifications.

In the example shown in fig. 2.5, 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.

2.2.4 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)`. 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.[10] An instance within a containment tree can for example be accessed any time given its distinguished name. Contrary to a CORBA object reference (cf. 2.1.1), distinguished names do not reliably identify instances since they may be reused when an instance has been deleted to refer to a different instance.

[9] A conditional package may for example be included when an instance is created or before a SET- or ACTION-request is executed.

[10] The same naming scheme can also be used to ensure object identity across several agents.

```
customer MANAGED OBJECT CLASS
  DERIVED FROM top;
  CHARACTERIZED BY
    customerPkg,
    contactListPkg,
  CONDITIONAL PACKAGES
    customerTypesPkg PRESENT IF ! an instance supports it !,
    opNetworkListPkg PRESENT IF ! an instance supports it !,
    serviceListPkg PRESENT IF ! an instance supports it !,
    typeTextPkg PRESENT IF ! an instance supports it !,
    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};
```

Figure 2.5: GDMO example (edited)

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 because it can mandate that an instance cannot be deleted as long as it still contains other managed objects.

Using *scoping* and *filtering*, it is possible to access more than one instance at a time, e.g. to send a GET-request to multiple 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 CMIP 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.[11]

2.3 SNMP

2.3.1 Architecture

The Simple Network Management Protocol [Bla92, CFSD90, LF93] is a wide-spread Internet standard for network management, mainly due to its simplicity and dissemination. Its architecture is conceptually similar to that of OSI network management. A *manager* communicates with an *agent* using the UDP-based SNMP protocol, which allows the manager to issue GET-, GET-NEXT- and SET-requests and to receive responses. The agent performs the requests sent by the manager and in addition may send TRAP-requests to the manager, which are notifications of a problem that occurred on the agent side. Unlike CMIP, the SNMP protocol does not define *operations* that can be executed on *variables* (see below). Operation calls have to be simulated by the manager setting certain variables in the agent. Manager and agent have to agree that setting a certain variable invokes a certain operation.

2.3.2 Model

SNMP is not an object-oriented model because it does not have a notion of classes, it knows only *variables*. These represent information in the agent's Management Information Base (MIB) and can be accessed using a unique *object identifier*. An object identifier is a node in a global naming tree whose structure is jointly administered by the ISO/ITU-T standards bodies (cf. [LF93, ch. 7, p. 122]). New object identifiers (for new MIBs) can be added to the leaves of the tree in a well-defined manner. Object identifiers can conceptually be compared to OSI distinguished names and CORBA object references since they unambiguously identify a variable or object, respectively.
A variable has a certain type that is defined using *ASN.1 macros* (cf. below). SNMP has only a small number of types available for describing information, which are the simple types `INTEGER`, `OCTET STRING`, `OBJECT IDENTIFIER (OID)`, `NULL` and a few predefined types such as `Gauge` and `TimeTick` [RM88].[12]
An additional structuring mechanism are (conceptual) *tables* which allow to store any number of variables under a given variable. The GET- and GET-NEXT SNMP requests allow to traverse such tables.

2.3.3 Specification Language

The specification language of SNMP (ASN.1 macros) is used for two purposes, namely the creation of MIBs and the definition of variables of a certain type.

---

[11] This may not be possible when the name binding specifies that contained instances have to be deleted *before* deletion of their parent.

[12] All ASN.1 types allowed for use in SNMP are listed in table 4.9 on page 110.

A new MIB is essentially created by defining a number of variables at a certain position in the global naming tree. These variables then represent the information an SNMP agent will serve.
Each variable is of a certain type, which is defined using ASN.1 macros. An example is shown in fig. 2.6
This example defines a variable that is allocated under `system` (which is a node in the naming tree). Its type is an `OCTET STRING` and it is read-only (i.e. it cannot be set).
By knowing the MIB of an SNMP agent, it is possible to know the *structure* of the MIB (e.g. the part of the global naming tree the agent serves) and the *types* of its variables.

```
sysDescr OBJECT-TYPE
         SYNTAX OCTET STRING
         ACCESS read-only
         STATUS mandatory
         ::= { system 1 }
```

Figure 2.6: ASN.1 macro

The SNMP model has been included in this discussion since it is first relevant for network management (a de-facto standard) and second, it can show whether and to what extent non object-oriented models can be integrated in the Generic Object Model. A discussion on the integration of SNMP into GOM can be found in sections 4.6.1 and 4.4.5.

2.4 Summary

The three target models described here have some commonalities and also some differences.
protocol-based whereas CORBA is interface-oriented (cf. section 1.3.2). The architecture of the first two models is based on the concept of agents representing a set of resources through MIBs and managers accessing agents through the use of a well-defined protocol (CMIP or SNMP respectively). Access point to information is in any case the agent, while CORBA clients access instances directly regardless of the server in which they are located. In this respect, CORBA servers can be compared in their functionality to CMIP- or SNMP agents in that they contain objects, but – unlike the latter two – the presence of CORBA servers is transparent to client management applications.
The CORBA and OSI models are both object-oriented. They have a notion of classes including attributes and operations, whereas SNMP only has variables to represent information.
CORBA identifies instances through unique, non-reusable, opaque object references while managed objects are identified by their distinguished names, which can be reused after deletion of an object. Object identifiers are used to uniquely refer to SNMP variables.
The specification language for data types in the OSI- and SNMP models is ASN.1. In the case of OSI, GDMO is used to define classes for managed objects, while SNMP makes use of ASN.1 macros to define variables. CORBA uses OMG IDL to define its classes.
Since GOM has an object-oriented model, it should be relatively easy to map the OSI and CORBA object models to it, while mapping SNMP variables may be more awkward. Mapping a non object-oriented to an object-oriented model may require some counter-

intuitive translations, comparable to the ones found in database technology when mapping relational and object-oriented databases (*impedance mismatch* [Cat91, ch. 4.7.2]).[13]
A detailed comparison of the CORBA, CMIP and SNMP models can be found in [Rut93, QP93].

[13]These issues are discussed in sections 4.6.1 and 4.4.5.

Chapter 3

Related Research

The purpose of this chapter is threefold. First, the scope of *Inter-Domain Management (IDM)* and a classification of work being done in this area will be presented. Then we will see what part of this area the thesis will focus on. The major part of the chapter will be devoted to an overview of research done in this area, with emphasis on the work done in the XoJIDM task force.[1] That work will be used as a typical representative of the static approaches and used for comparison with the proposed generic model.
It will be seen that most approaches dealing with inter-domain management employ a static, compile-time based approach, which has some deficiencies. The most important ones will be discussed.

3.1 Inter-Domain Management

3.1.1 Overview

The term *Inter-Domain Management* is used here in the sense given by XoJIDM (cf. 3.1.2), which defines a *domain*[2] as a system by means of which management is accomplished.[3] Examples of domains are CORBA, CMIP (X.700), and SNMP.

[1] As of August 1997.

[2] Usually, the term *domain* is used in management terminology to denote an area of management to which the same policies apply, such as a security- or administration-domain (cf. [Slo94]).

[3] Throughout the thesis, *domain* will be used synonymously with *(object) model*; the latter being the more generic term.

	CORBA Server	**CMIP Agent**	**SNMP Agent**
CORBA client	———	• JIDM [Int95, Hie96b, Sou94c] • *GOM* • Liaison [Der97] • Bilingual Agent [SG94] • Genilloud [GG95, GP95, Gen96] • Telecom SIG [Tel96]	• JIDM • *GOM* • Liaison [Der97] • Mazumdar [Maz96] • Telecom SIG [Tel96]
CMIP manager	• JIDM [Hie96b, Sou94b, Sou94d] • Mascotte [Mas97] • [BGG94]	———	• CMIP-SNMP Proxy-Agent [RFC91a] • [ACH93] • [KS93] • [MPBS95] • [Cha93, New93]
SNMP manager	• JIDM • Mazumdar [Maz96]	• Protocol Independent Management Agent [MBL93] • IMC [LaB93b, LaB93a]	———

Table 3.1: Classification of Related Work in the Domains *CORBA*, *CMIP* and *SNMP*

Inter-Domain Management therefore deals with management of one domain through a different domain, e.g. management of an OSI agent through a CORBA manager. *Multi-Domain Management*, as used in this thesis, denotes management of *several* domains through a single domain. Whereas there exists a binary relation between the management and managed domain in Inter-Domain Management, Multi-Domain Management has an 1-M relation.
Currently, most work done in Inter-Domain Management deals with the management models of SNMP and CMIP[4], with CORBA becoming increasingly important. The overview of related research will therefore be restricted to the domains of SNMP, CMIP and CORBA.
An overview of work done in Inter-Domain Management between SNMP, CMIP and CORBA is given in table 3.1. The vertical row contains management domains whereas the horizontal row contains managed domains.
Management of a domain X using X itself does not fit the above definition of Inter-Domain Management, which dictates that the management and managed domains are to be different. Since, if both domains were the same, which is normally the case in management, then Inter-Domain Management would not be of interest to us. The cases of using an SNMP manager to manage an SNMP agent, a CMIP manager to manage a CMIP agent and a CORBA client[5] to manage a CORBA server are therefore not discussed here.

Management Domain = CORBA, Managed Domain = CMIP Work in this particular area tackles how CMIP agents can be managed by CORBA managers.
The XoJIDM task force has defined an algorithm for mapping a specification written in GDMO / ASN.1 to OMG IDL and is currently working on an algorithm for runtime conversion of requests between the domains of CORBA and OSI. As this work is important for the thesis, a more detailed description will be presented in section 3.1.2.
The work on the *Bilingual Agent* [SG94] describes how a CORBA interface can be added to an OSI agent, thus making it - in addition to OSI managers - also accessible to CORBA managers. Its approach is to translate GDMO templates and ASN.1 types into OMG IDL code. For each managed object in the agent, a corresponding CORBA (front-end) instance is created that forwards CORBA requests sent to it to the underlying CMIP managed object. Both static translation and runtime conversion code were generated manually, but the intention was to make use of an already existing tool (*Managed Object Agent Composer* [MOA94]) to automate both tasks. This work is similar to the solution described in [Gen96, ch. 4.3] which proposes to upgrade existing OSI agents with IDL interfaces to the managed objects.
Related work done on managing OSI objects from ANSA [ANS93] managers is described in [GG95, GP95, Gen96].[6] This approach can also be divided in a *translation* part and a

[4] Although *CMIP* actually only denotes the protocol used for communication between manager and agent in the OSI management model, in practice it is often used in a *pars pro toto* meaning, denoting the entire OSI model. It is therefore used interchangeably with the terms *OSI* or *X.700*.

[5] In the CORBA domain, the terms *client* and *server* are usually preferred to *manager* and *agent*, respectively.

[6] A previous paper [BGG94] described the opposite mapping, namely accessing ANSA objects from

runtime interaction part (cf. [Spe97]). The translation part maps an OSI GDMO/ASN.1 specification to its corresponding ANSA IDL specification plus conversion code. The runtime translation makes use of both the generated ANSA classes and the conversion code to translate ANSA operations into OSI requests and back.

An OSI agent is represented by a corresponding *ANSA proxy-agent* (*adapter*), whose task is the creation and deletion of *ANSA managed objects*, which are proxy instances for managed objects in an OSI agent, and the discovery and retrieval of existing objects from the real OSI agent. Each managed object in the OSI agent is represented by a corresponding ANSA managed object located in the proxy-agent, which itself represents the OSI agent to the ANSA world. Operations invoked on an ANSA managed object are translated to OSI CMIP requests and sent to the real agent. CMIP responses are converted to ANSA. Conversion code from GDMO/ASN.1 to ANSA IDL has previously been generated by a translator.

Another approach for managing CMIP agents from CORBA management applications is the *Liaison* project described in [Der97, Der96, BD97]. At its core is the *Proxy*, which works similar to an HTTP server. It accepts CMIP requests, encapsulated in HTTP, dispatches them to a CMIP agent and returns the response again in the form of HTTP to the management client.

The type system of Liaison is very simple: it consists only of type `string`. This forces the programmer to convert all types of the native host system (e.g. C++ or Java) to string form, e.g. to provide arguments to a function call or to convert a return value from a string to a host system type.

CMIP requests are mapped to HTTP, sent to the *Proxy* and there converted into a procedure call which implements the CMIP request. This has the advantage that the Proxy does not need to maintain any state information, but it also includes the disadvantages of not being object-oriented (internally).

The OMG Telecommunication Special Interest Group has issued a white paper on how to use CORBA in the field of telecommunications [Tel96]. The focus of this work is on provisioning CORBA *services* ([COS95]) tailored to the needs of the telecommunications domain. Physical and logical resources are supposed to be represented by CORBA interfaces. Access to OSI and SNMP managed systems will reuse the work done by XoJIDM.

Management Domain = CORBA, Managed Domain = SNMP This part of IDM deals with managing SNMP agents from a CORBA-based management domain, thus allowing transparent CORBA-based management of SNMP devices without having to deal with the SNMP protocol.

XoJIDM has defined a translation algorithm that maps CORBA IDL to SNMP [Spe97, Part 6]. Following this work, [Maz96] describes an approach in which SNMP MIBs are translated to CORBA IDL. The Dynamic Skeleton Interface (DSI) and Dynamic Implementation Routine (DIR) [OMG95] are used at runtime on the server side to convert incoming CORBA requests to SNMP PDUs and SNMP responses to CORBA values.

Liaison [Der97, Der96, BD97] allows CORBA management applications to manage SNMP agents. Aside from access to CMIP agents, it also offers access to SNMP agents.

OSI managers.

The OMG Telecommunication Special Interest Group has proposed a scheme describing how SNMP devices can be managed from a CORBA-based application [Tel96].

Management Domain = CMIP, Managed Domain = CORBA The subject of this research area is how a CMIP manager can manage objects in the CORBA domain.
The XoJIDM task force considers in its prospective Interaction Translation document [Int95, Hie94, Hie96b, Sou94b, Sou94d] the use of a proxy agent (*controller*) to receive CMIP requests and dispatch them to CORBA objects using the Dynamic Invocation Interface (DII). There is a *MIBserver* that records OSI names (Distinguished Names) and CORBA object references in a table, which can be queried to translate OSI names to CORBA object references and vice versa. The approach makes use of XoJIDM's Specification Translation document [Spe97, part 4], which defines a mapping from IDL to GDMO/ASN.1.
The Mascotte project [Mas97] deals with how the various elements of a CORBA system can be managed using CORBA itself on the manager side. To achieve this, CORBA IDL interfaces have been defined for a number of entities such as the *basic object adapter*, the *ORB core* etc.
To integrate these management interfaces with existing (OSI) management, their IDL specification has been (manually) translated to GDMO / ASN.1 and a scheme has been devised that translates at runtime requests between the CMIP and CORBA domains.[7] Thus, it is possible for a CMIP management application to manage (selected) CORBA objects.
Similar work is described in [BGG94], but the domain to be accessed from OSI managers is ANSA [ANS93] instead of CORBA. The goal is to access existing ANSA objects from OSI managers without modifying the ANSA objects. The approach is twofold: first, a static translation maps the ANSA object model specification to GDMO/ASN.1. Then, a runtime translator in an OSI *pseudo agent* maps CMIP requests to ANSA operation invocations.[8] In order to find ANSA interface references from OSI names, when an OSI CREATE request creates a corresponding ANSA object, the binding between OSI name and ANSA interface reference is recorded in the pseudo agent's naming table and subsequently used to retrieve the correct ANSA interface reference.

Management Domain = CMIP, Managed Domain = SNMP This work investigates how a CMIP manager can manage SNMP resources transparently.
The Internet Architecture Board (IAB) has defined the notion of an OSI *proxy agent* which has the role of an OSI agent towards an OSI manager and the role of an SNMP manager towards an SNMP agent, thus acting as a gateway between CMIP and SNMP requests. The goal of the proxy agent is to present SNMP resources to an OSI manager in the form of OSI managed objects. Employing this scheme, the Internet MIB II specification [RFC91b] has been (manually) translated[9] to allow OSI management applications to manage SNMP

[7]This scheme involves a *proxy OSI agent* translating OSI requests to CORBA operations and vice versa.
[8]The code for the runtime translator is supposedly generated by the static translator (the paper mentions that future work will include such a translator).
[9]The resulting OSI MIB is called OSI Internet Management (OIM II).

resources.
Similar work implementing a gateway between the domains of OSI and SNMP is described in [ACH93]. It consists of a special intermediate OSI agent that provides OSI managers access to SNMP resources in the form of managed objects. The proposal includes a translation that maps SNMP object types to OSI managed objects and, at runtime, OSI Distinguished Names to SNMP object identifiers (OIDs). Unlike in the above approach, translation of SNMP MIBs to OSI MIBs is automated. The gateway agent implements a *name mapping* from OSI names to SNMP variable names and a *service mapping* that maps CMIP requests to their SNMP counterparts.
The work described in [KS93] realizes an *application gateway* (cf. [Ros90b]) between OSI manager and SNMP agent, which is essentially an OSI agent acting as an SNMP manager and converting requests between the two domains. Similar to the approach described above, a *name mapping* and a *functional mapping* are defined, which allows to translate between OSI names and SNMP object identifiers on the one hand and CMIP- and SNMP-requests on the other hand. The mapping between SNMP and OSI MIBs is conceptually similar to the one in [ACH93], but results in fewer managed object templates being generated. The authors also describe problems encountered in creating an application gateway, such as reconciliation between a connection-oriented (CMIP) and connectionless (SNMP) protocol, and breaking up large CMIP PDUs into several smaller ones with subsequent reassembly.[10]
Another approach similar to the ones already described is [MPBS95]. This work is based on the Network Management Forum's ISO/ITU-T and Internet Management Coexistence (IIMC) activities, which define mappings between SNMP/CMIP and vice versa [LaB93b, Cha93]. A compiler has been written following the IIMC approach for mapping SNMP MIBs to OSI MIBs. The resulting GDMO code is already enriched with code that translates CMIP- to SNMP -requests and can be compiled into an OSI agent [Pav93].

Management Domain = SNMP, Managed Domain = CORBA Work in this area deals with enabling SNMP mangers to manage CORBA objects.
XoJIDM's specification translation document [Spe97, Part 5] defines a mapping from SNMP to OMG IDL.
[Maz96] describes an implementation that makes use of this mapping. It consists of a CORBA-based agent that acts as an SNMP agent (using CMU's[11] SNMP library). Incoming SNMP PDUs are converted to CORBA requests and CORBA events are mapped to SNMP traps.

Management Domain = SNMP, Managed Domain = CMIP How to use SNMP managers for management of CMIP agents is important here.
The goal of the approach followed in [MBL93, WMBL92] is to augment a CMIP agent to support the SNMP protocol as well. This is done by creating a special agent with a *protocol-independent MIB (PIM)* and CMIP and SNMP *protocol processors* that - depend-

[10] Unlike SNMP – which is based on UDP – CMIP has no packet size restriction.
[11] Carnegie Mellon University.

ing on the protocol used[12] - dispatch an incoming request to the corresponding object.
The SNMP part of the protocol-independent agent is implemented by translating the CMIP MIB to both an SNMP MIB and C++ objects for the PIM representing SNMP objects. Any CMIP object created also registers a corresponding SNMP entry in the *name registration table* in order to make it known to SNMP managers.
Use of this scheme allows vendors to provide additional SNMP access to their OSI agent managed devices, enabling existing SNMP-based applications to manage the device.

3.1.2 NMF - X/Open Joint Inter-Domain Management

The Network Management Forum – X/Open Joint Inter-Domain Management task force (XoJIDM)[13] has as its goal the integration of CORBA in the network management world. The task force tackles issues such as how CORBA can be used for building management applications (clients) and managed applications (agents, servers). Specifically, some of the major objectives are:

1. Management of OSI agents using CORBA clients
2. Management of SNMP agents using CORBA clients
3. Management of CORBA agents using OSI managers
4. Management of CORBA agents using SNMP agents

As the focus of this work is on the manager side, only the first item will be examined, namely the scheme proposed by XoJIDM to enable CORBA applications (clients) to transparently manage OSI Managed Objects.
The approach chosen involves translation of an OSI agent specification (GDMO / ASN.1) to CORBA IDL (*Specification Translation* [Spe97]) and subsequent runtime conversion of CORBA requests to CMIP and back (*Interaction Translation* [Int95]). CORBA clients include the language bindings of the IDL specification generated by a GDMO / ASN.1 to IDL translator.
Specification Translation defines a mapping from GDMO and ASN.1 to IDL as shown in table. 3.2.
GDMO templates may be mapped up to 3 IDL interfaces: a default CORBA interface, an interface for notifications and one for handling multiple replies.
GDMO packages as such do not exist in the translated code, but all elements (attributes, actions and notifications) of both mandatory and conditional packages are added to the resulting IDL interface. Information about conditional packages is present only in comment form. Possibly large IDL interfaces may result from this algorithm, containing all elements of conditional packages even if just a few conditional packages may be present in the resulting managed object.[14]

[12] The determination of the correct protocol is easy since SNMP requests are sent as connectionless UDP packets to a well-defined port, while CMIP requests use connection-oriented communication.

[13] More information can be obtained from the Web site at `http://www.rdg.opengroup.org/mem_only/tech/sysman/jidm/index.htm`.

[14] Which conditional packages are to be present is decided by the creator of the instance or by the OSI agent that creates an instance.

GDMO Templates	IDL Interfaces (up to 3: class, notifications and multiple replies)
GDMO Packages	Elements of IDL Interface
GDMO Attributes	IDL Types / Interfaces
GDMO Attribute Access	IDL Operations (get_X(), set_X())
GDMO Actions	IDL Operations
GDMO Notifications	IDL Operations
GDMO Parameters	IDL IDL Types
ASN.1 Types	IDL Types

Table 3.2: XoJIDM Specification Translation

Attributes in GDMO templates are mapped from ASN.1 to their corresponding IDL type and are made member variables in the resulting IDL interface. Access to them is allowed only through the use of *accessor operations*. Corresponding to their GDMO attribute access definitions, these will enforce constraints on attribute access such as absence of accessor operations that modify a value if the latter was declared read-only in the GDMO template. There may be a number of accessor operations per attribute for getting, setting, replacing with default, adding and removing attribute values.[15]
ASN.1 types are translated to their corresponding IDL types, e.g. an `OCTET STRING` is mapped to a `string`, a `SEQUENCE` to a `struct` etc.
Interaction Translation specifies the runtime conversion mechanism and the general embedding of the proposed scheme within the CORBA architecture as shown in fig. 3.1.[16]
The GDMO / ASN.1 documents that define the agent's MIB are translated to IDL according to the rules of Specification Translation. The resulting IDL code is subsequently translated to both client stubs and server implementation skeletons (server stubs) of the desired target language (e.g. C++).
The previous GDMO / ASN.1 to IDL translation provided not only IDL interfaces, but also code (behavior) for the server stub implementation, which converts CORBA requests into CMIP PDUs using for example XOM / XMP. The server stubs merged with the generated stub implementation code represent fully functional proxies for managed objects and reside in a CORBA *CMIP object adapter* (CMIP OA).
A management application may now include the generated client stubs in a desired language binding (e.g. C++ or Smalltalk). A request invoked on a client stub will be transparently forwarded to the server object in the CMIP OA, which will then use XOM / XMP to dispatch it to an agent using the CMIP protocol. The response from the agent

[15]Accessor operations for adding or removing values are present only for set-valued types.

[16]Note that at the current date (Aug. 97), no formal Interaction Translation document has been furnished yet, but a few proposals ([Hie94, Hie96b, Sou94b, Sou94d, Sou94a, Sou94c, Sou94e]) have been submitted to XoJIDM that are to be merged into a common document. Whenever XoJIDM's Interaction Translation document is mentioned, the author therefore refers to a prospective future document extrapolated from the proposals submitted so far. Note, however, that the final document may or may not be in line with the submitted proposals wrt. its contents. The proposals that deal with CORBA manager managing OSI agents are specifically [Hie94, Hie96b, Sou94c].

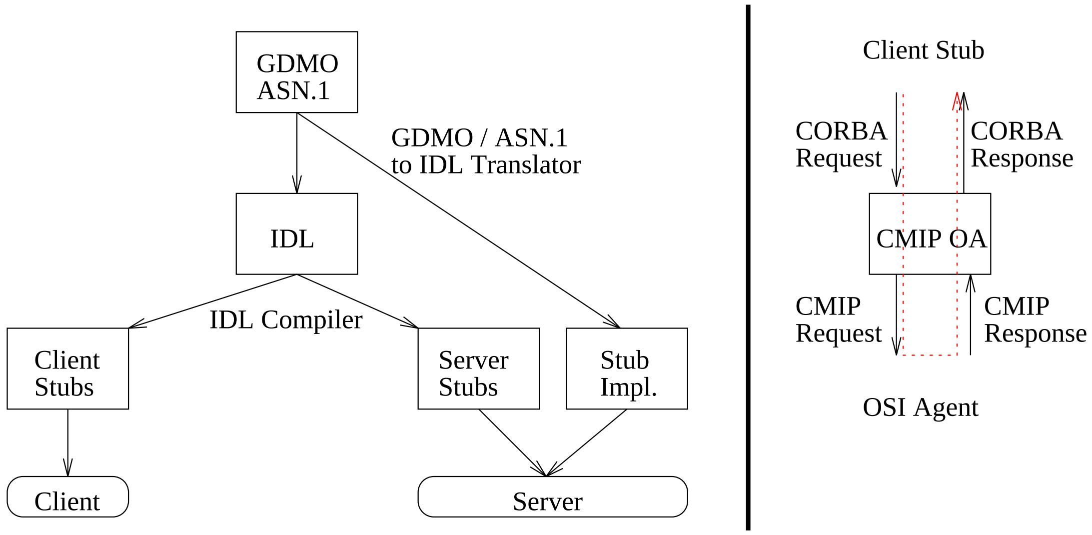

Figure 3.1: XoJIDM Interaction Translation Architecture

is subsequently converted to CORBA[17] and returned to the client.
While the document on Specification Translation is almost finished, Interaction Translation is still far from being ready: as of Aug. 1997, only a few proposals have been submitted for discussion.

3.1.2.1 Deficiencies

The static specification translation approach as exemplified by XoJIDM has a number of drawbacks/deficiencies resulting mostly from idiosyncrasies of OSI. Some of them are described below. A more detailed description of problems of specification translation is given in [Gen96, 152–167].

3.1.2.2 Multiple Replies

A GDMO ACTION may return a single or multiple responses. This is handled in XoJIDM's approach using exceptions [Spe97, 4.5.2]. When a single response is returned, no exception is thrown. When multiple responses are returned, an exception is thrown that has to be caught by the application to handle the responses.
Responses are collected via an *event channel* [COS95] using either the push- or pull model. For each GDMO template containing ACTIONs, the normal IDL interface plus two additional ones for the pull- and push models will be generated.
This scheme has the disadvantage that for each GDMO template containing ACTIONS, up to three IDL interfaces will be generated.
Moreover, using the event channel which was actually conceived for storing events for the purpose of iterating through a result set seems counter-intuitive to common design practice.

[17]I.e., it will be converted to a data type of the chosen language binding.

Last but not least, despite the fact that in practice few ACTIONs return multiple responses, clients always have to wrap code that invokes an action with exception handling code. This code may only be used a fraction of the time but nevertheless increases the size of the code and may impair readability.
[Gen96, 4.6.8] describes a scheme called *flexible translation* to solve this problem.
Section 4.6.2.4 describes how GOM handles multiple replies.

3.1.2.3 Attribute Operations

Another problem is the number of operations generated for each attribute in a GDMO template: since CMIP's M-GET/M-SET request may raise a number of exceptions, Xo-JIDM decided to map attribute access to IDL operations, which can raise exceptions as well. For each attribute, there are up to 5 operations to get, set, set to default, add and remove an attribute (see [Spe97, 4.3]), leading to increased code size. Genilloud [Gen96, 4.3.3] describes this problem in more detail.
A characteristic of CMIP is that multiple attributes can be set or retrieved with a single request. This has been taken into account by providing generic get- and set methods, but this actually defies the purpose of JIDM's strong typing concept.
Further problems concerning the manipulation of GDMO *attribute groups* are described in [Gen96, 4.6.9.1].

3.1.2.4 Conditional Packages

Actions, attributes and notifications of GDMO conditional packages may or may not be present in a managed object instantiated from a GDMO template. The JIDM translation algorithm by default includes *all* conditional packages that might possibly be instantiated in the resulting IDL interface [Spe97, 4.2]. When an implementation detects that an element to be accessed is in a conditional package that is not present, an exception is thrown. How to keep track of the conditional packages (and their elements) that have been included is implementation-dependent.
Including all elements of all conditional packages defies a central design choice of GDMO; namely to allow clients to decide which elements should be included in a managed object and which not, thus being able to 'pay' (in terms of memory size) for the elements that are actually used. If all conditional packages are included in the resulting IDL interface, but only a few are actually used, then an overhead is carried with each instance.
GOM's generic object model (cf. 4.2.2) allows to decide at instance creation time which elements should be present in an instance. It is for example possible to create an 'empty' instance (one with no attributes) and add an attribute only when it is requested the first time (lazy attribute creation).

3.1.2.5 Notifications

Notifications are mapped to four operations: two for pulling and two for pushing operations (confirmed, unconfirmed) [Spe97, 4.6]. As in the case of attribute operations this leads to increased code size.

GOM's event handling model (4.5) is generic and does not take into account typed events/notifications. Therefore, a significant code reduction can be achieved.
Problems of mapping notifications to IDL operations are described in further detail in [Gen96, 4.6.10].

3.1.2.6 Recursive ASN.1 Types

As IDL does not support recursive types (except with sequences), the JIDM translation algorithms supports only direct recursion using sequences of length 1. Indirect recursion is not supported. Genilloud states that recursion can best be supported through manual intervention (flexible translation) [Gen96, 4.6.3].
GOM allows to map recursive data types in a simple manner both in the instance- and meta model (see section 4.6.2.6).

3.2 Summary

A few characteristics common to most approaches are described here.
First of all, to perform Inter-Domain Management between a management domain (A) and a managed domain (B) requires that the specification of domain A be translated into a corresponding specification in domain B (syntactic translation). This translation is in most cases performed by a compiler, which accepts a specification of domain A and outputs a specification for domain B.
Second, the mapping is usually unidirectional and irreversible.[18] The reason for this is that transforming a specification from one domain to a different one results in most cases in loss of information since all domains are syntactically and semantically different. It is possible, however, to prevent information loss at translation time to a certain degree by storing it in comment-form in the resulting specification, enabling a B-A translator to transform specification B back to A.
Third, *semantic translation* determines the runtime behavior of a generated specification and involves bidirectional runtime translation between domains A and B. The code for a runtime mapping may have been generated by the specification translator or it may be an independent entity, possibly using helper information previously generated by the specification translator.

This thesis focuses on multi-domain management with the common management domain being CORBA and the managed domains being CMIP, CORBA and (to a lesser extent) SNMP. It will be contrasted mainly with the approach proposed by XoJIDM [Spe97, Int95][19], which has some limitations:

- Clients have to know the types of classes they will be managing; this is typically done by compile-time inclusion of generated client stubs, which creates a tight binding

[18] This implies that the exactly same specification of domain A cannot be re-engineered from the generated specification of domain B using a B-A translator.
[19] Specifically Inter-Domain Management between CORBA and CMIP and CORBA and SNMP.

between a client and a service (offered by a server in the form of a class). Modification of the specification leads to shutdown and regeneration of the client application, which is unacceptable for certain types of applications.

- Static inclusion of client stubs may lead to a large number of classes and methods being present in the client application since the mapping algorithms defined by XoJIDM's Specification Translation generate a large amount of code. Dependencies of classes on other classes have to be resolved at compile-time[20], which may make the client include all dependencies of a certain class into the executable. In the worst case, the transitive closure over all client stubs will have to be included.

- A static, compiled-time based approach is problematic with certain ASN.1 types such as `ANY DEFINED BY`[21], extensible attribute groups[22], and a mapping of conditional packages[23] is awkward.

- Finally, most approaches specialize in mapping exactly one domain into another one, requiring one translator per mapping. If n domains are given, then each additional mapping to/from all other domains requires $n^2 - n$ new translators.

It will be shown that a dynamic mapping approach – based on metadata – has advantages over static (compile-time) mapping between domains and will solve some of the problems mentioned above. The author's intention is to show that the approach proposed in this thesis does not necessarily compete with the one adopted by XoJIDM, but complements it, depending on the type of the application. For certain applications, compile-time inclusion of a fixed number of classes may be preferred, for others runtime inclusion via metadata mechanisms may be better suited (see chapter 5).
Moreover, it will also be shown that the model presented here does not necessarily need to be restricted to the domains of CMIP and SNMP, but can be extended to integrate other domains such as COM (OLE) [Bro94] or Java [Sun95].
Of course, the proposed model could also be employed to enable OSI managers to transparently interfere with SNMP or CORBA domains. However, this goal is beyond the scope of the thesis and will be treated only briefly in 4.6.6.

[20] At least in languages such as C++, which is probably the most frequently used CORBA language binding.

[21] This ASN.1 type can be determined only at runtime.

[22] Set of attributes to which attributes can be added or from which attributes can be removed. Attribute sets are manipulated as a whole, i.e. a GET-request on an attribute group returns all of its members.

[23] Members of conditional packages (e.g. attributes or operations) may or may not be present in an instance. XoJIDM by default includes all members of all conditional packages.

Chapter 4

The Generic Object Model

4.1 Architecture

This section will give an overview of the architecture of the Generic Object Model. It will briefly introduce the three main components of GOM, the *generic object model*, the *metadata repository* and the concept of *adapters*.
The architecture of GOM is shown in fig. 4.1.

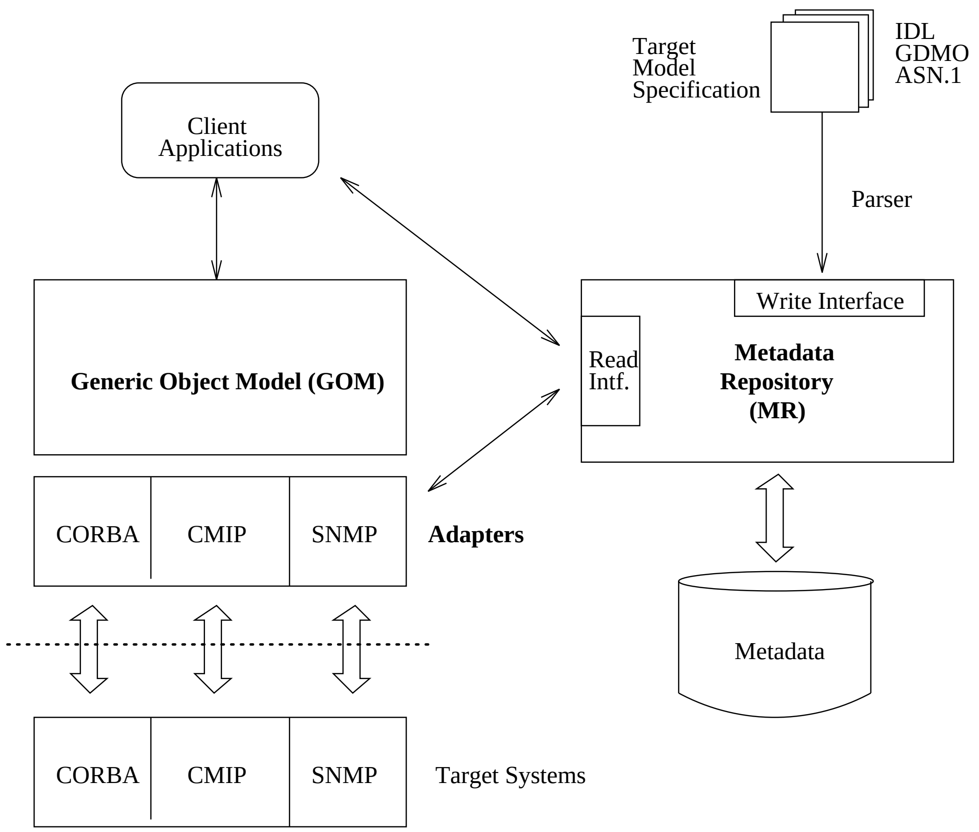

Figure 4.1: Architecture of GOM

At the core of GOM is the *metadata repository* (MR) which maintains information about

the structure of classes of several target models. It is needed to type-check request arguments at runtime and to convert values between GOM and target models. As GOM possesses no knowledge about the classes available in a target system because they are not included at compile time, it has to rely *entirely* on metadata for instance manipulation.
The next important component of GOM's architecture is the *generic object model*. It wraps instances of target systems in a generic representation and offers them as *proxy instances* to client applications. Proxy instances can be manipulated by applications as if they were normal instances, but rather than maintaining functionality of their own, operations invoked on them will be processed by GOM, which performs type checking, conversion, dispatching to the corresponding instance in the target system and conversion of return value [GHJV95].
All of these tasks are actually performed by *adapters*. An adapter is a *bridge* between the generic object model and exactly one specific target model. It type-checks, converts and dispatches requests between them. Every proxy instance will have a reference to an adapter to which it forwards all requests it receives.
The integration of these three components in GOM and their cooperation will be explained *in the context of the overall architecture* in sections 4.1.1, 4.1.2 and 4.1.3. A more detailed discussion of each component will be given in sections 4.2, 4.3 and 4.4.

4.1.1 Object Model

The object model of GOM is defined through a number of classes (IDL interfaces) with attributes and operations, and the interactions among them. These classes provide a homogeneous abstraction layer to clients, shielding them from and leveling the differences between various target systems such as CORBA, CMIP and SNMP.
The CORBA interfaces are `GenObj`, `Val` (and subclasses), `Adapter` and `Factory`. As these are explained in detail in section 4.2.2.2, it is the task of this section to focus on the *interaction* between their instances.
Each instance in a target system is represented by a corresponding *proxy instance* in GOM. Proxies receive requests such as for attribute retrieval or operation invocation and forward them to the target instance they are representing.
A proxy object (`GenObj`) may be created locally, i.e. in the client's address space (this is the default case) or in a different process on the same machine, or remotely, i.e. on a different machine (see fig. 4.2).
Local proxies with either local (2) or remote (3) target instances will probably be the most frequent situation. In these cases the proxy is always in the same address space as the client, and communication overhead between client and proxy is minimal.[1] Communication between the proxy- and target object takes place using whichever protocol is appropriate, depending on the target's object model. This communication may for example be a CORBA- or CMIP request.
Case (1) can be used if interaction between proxy and target object is much more intense than between client and proxy, e.g. in the case when a request to a proxy instance generates multiple requests to the target object, or if a certain type of adapter is not

[1] Essentially the cost of a local procedure call.

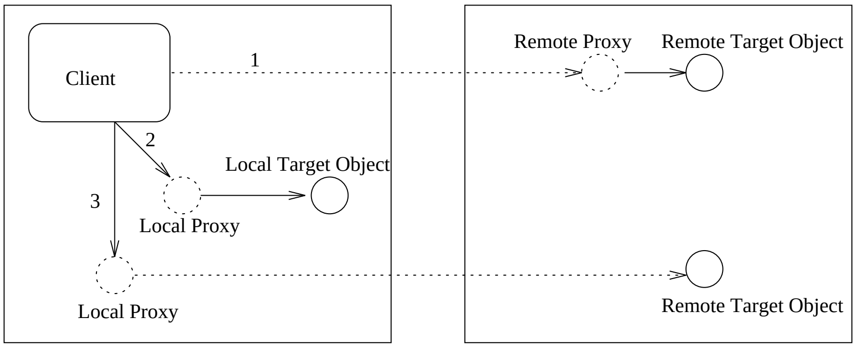

Figure 4.2: Local and remote proxies

available on the local machine. This may be the case when the local host wants to access OLE instances ([Box95]) without having an OLE adapter available. In this case, both the proxy and the target instance might be created in the remote host where a suitable OLE adapter is located (1).

A theoretical fourth case is that of a remote proxy and a local target object. As this increases communication overhead significantly without any visible advantage, this alternative is not considered (although not prohibited).

Proxy instances and their corresponding adapters always have to be co-located at the same location. This means that if a proxy instance is to be created remotely, then a suitable adapter will have to be present at that location as well.

IDL Interface `GenObj` defines the class for proxies. It contains operations for getting and setting attributes, invoking operations, deleting the instance, and accessing metadata about the target instance.

A proxy instance is *stateful*; it records the classname of the target object, its own instance name (may be empty), a reference to an adapter to which all requests are forwarded (see below) and a property list which allows any further state information to be attached (as for example needed by adapters).

Proxy instances are created using *factories*. A factory is responsible for locating the correct adapter for the desired target object model and dispatching the creation request to it. The adapter will then create the target instance (at the specified location), a (local or remote) proxy instance for it and return the proxy instance to the caller.

The adapter will set attribute `adapter` of the created proxy to refer to itself so that the proxy can forward any operation invoked on it to a suitable adapter.

There is exactly one adapter for each object model. Its IDL interface name consists of a concatenation of the name of the object model it represents and the suffix `"_Adapter"`, e.g. `"DSOM_Adapter"`. Whenever an adapter is created, it is registered with a *naming service* (cf. [COS95]) using this assigned name. For example, a CMIP adapter on host `"adl"` would have an interface name of `CMIP_Adapter` and would be registered with the naming service on host `"adl"` with key `"CMIP_Adapter"` and value `"<CORBA object reference of the CMIP adapter instance>"`.

When a factory creates a proxy instance, it first has to find a suitable adapter that serves the desired object model. Using the above name, the factory searches the naming service either on the local or remote machine (depending on the location of the proxy). If the

adapter is found, the creation request is forwarded to it, otherwise a new adapter will be created (if its interface is available) and its name added to the naming service.
A factory allows to specify the location of the proxy- and target instances in operation `Create`. If the location of the proxy instance is remote, then the naming service on the remote machine will be queried for a suitable adapter, otherwise the local naming service will be used.
The above discussion of the provided interfaces making up GOM comprises the *instance model* (cf. 4.2.2) which will be used most frequently by clients. However, clients may also access the other part of GOM's object model, the *meta model*, to retrieve metadata about elements of the target system. A discussion of the meta model is presented in section 4.2.3.
As the various *language bindings* for the CORBA interfaces that make up GOM's object model may not make use of the entire set of features available for that language (since it has to be 'portable' across many languages), it is sometimes useful to provide a layer on top of the generated languages bindings that conforms to the philosophy of the host language. These are called *convenience bindings* and are discussed in section 4.2.4.

4.1.2 Metadata Repository

The metadata repository (MR) is a central component of GOM that maintains metadata about the various target object models managed by GOM. Its main users are:

Adapters The generic object model of GOM needs access to metadata about the target instances for which it provides proxy instances in order to type-check and dispatch requests and convert between a target- and the generic model. This is the task of adapters.

Clients The metadata repository is not only used internally by adapters, but it is also open to clients, which can query it for metadata. Both adapters and clients use the same interface to the MR: the *read* interface.

Compilers Metadata can be added to the MR either by means of *metadata adapters* or compilers using the *write* interface. This allows compilers (or parsers) to generate metadata in the MR for any target model by reading the target specification and entering it (in an acceptable form) to the MR.

Metadata is kept persistently in *metadata caches*, one for each object model. Metadata caches are persistent, thus having the function of databases meta information.
Metadata adapters are responsible for providing target system metadata *just-in-time*. They are called by the metadata repository whenever metadata cannot be found in the metadata cache and have to provide the desired information by tapping into existing sources of metadata (such as other metadata repositories), converting it to a form that is accepted by the MR and copying the requested information to a metadata cache. The metadata repository is discussed in section 4.3.2.

4.1.3 Adapters

Adapters function as bridge between the generic- and exactly one specific target model. They level the differences between the various target models by assisting the generic object model in communication with target systems. All the system dependent code that needs to be written to communicate with the target system is located here.
An adapter is an abstract class. No instances of it should ever be created, but it should be subclassed by real adapters that override and implement its operations to provide bridging functionality.
As mentioned above, the CORBA interface name of an adapter is constructed as a concatenation of its object model name and the suffix `"_Adapter"`. The ASCII form of this name is also used to register an instance of an adapter with a naming service. This allows a factory to create proxy instances by (1) locating a suitable existing adapter instance from a naming service or (2) by creating a new instance of an adapter and registering it with the naming service.
An advantage of adapters – besides encapsulation of system-specific code – is that libraries for access to target systems will be linked with an adapter, but not with the proxy instances. Therefore target system specific code will not have to be linked with the client application, resulting in code size reduction. Also, since an adapter is represented as CORBA interface, multiple entities can retrieve an adapter instance using the naming service and access it without increasing code size since the code is loaded into memory only once.
Adapters are discussed in more detail in section 4.4.

4.2 Object Model

An *object model* is essentially the description of a system by means of objects and their interactions. Information (data) is represented through objects and information flow through interaction between objects (cf. 1.2).
The object model of GOM consists of a number of CORBA interfaces containing attributes and operations. Their purpose is to provide a uniform programming model to clients. This allows transparent manipulation of a set of *heterogeneous* target object models with minimal concern for their structure, thus allowing a user to get- and set attributes of a proxy GOM instance without necessarily needing to know that the target instance may actually be a managed object in an OSI agent.
GOM's object model is divided into an *instance*- and a *meta model*. The instance model is used to represent instances of target systems (e.g. managed objects in an OSI agent, SNMP variables in an SNMP agent or objects in a CORBA server), whereas the meta model is used to represent metadata about the target systems (e.g. information about attributes of a class, parameter types of an operation etc.).
The instance model is described in section 4.2.2. Its main characteristics will be discussed and a detailed description of the interfaces involved will be given. Also, use cases demonstrating how to manipulate instances of target models using GOM's instance model will be shown.

The meta model is described in section 4.2.3. Besides describing the interfaces structuring the meta model, the focus will be on *layout definitions* and their mapping to the target models of CORBA and CMIP. They define how the generic structure of the meta model is mapped to a specific target meta model, e.g. how metadata for a GDMO class template is represented in the meta model.
Section 4.2.4 describes the concept of *convenience bindings*, which are (optional) additional layers on top of the bindings generated from the IDL interfaces. Their objective is to offer a better and more user-friendly integration with the host language. Examples of convenience bindings for both C++ and Smalltalk will be given.

4.2.1 Overview

An overview of the interfaces available in GOM is shown in fig. 4.3.

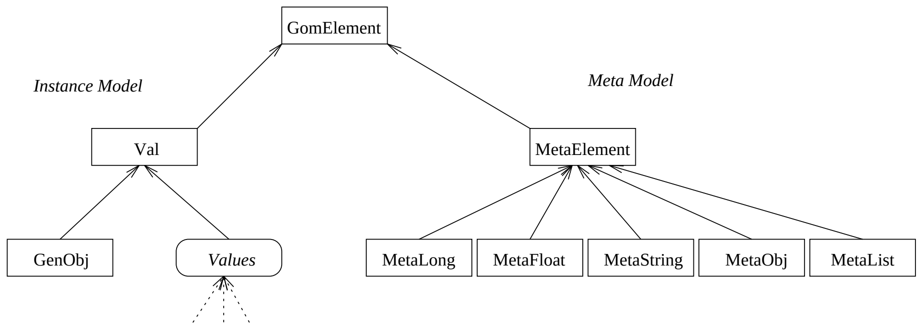

Figure 4.3: The object model of GOM

At the top of the hierarchy is IDL interface `GomElement` (definitions are shown in fig. 4.4) from which all classes of the instance- and meta model inherit. It is used to define a number of operations common to both models. `GomElement` is an abstract interface (i.e. no instances may be created) and therefore all operations must be overridden by subclasses.
The `GomKind` type enumerates all elements of the instance- and meta model and operation `GetKind` identifies for each element what type of element it is. This is important when one does not know which interface one has to deal with, as in the case of generic metadata browsers that displays metadata differently depending on their type.
Operation `AsString` returns a string representation of the instance and `Copy` returns a copy.
Operations `Dump` and `Read` write the contents of the object to a stream or read from the stream to re-create the object. Having the ability to stream out all of its instance- and meta model objects and to reconstruct them at a later time allows GOM to save its objects to a file, thus making them persistent, or to send them across a communication link (in order to re-create them at a remote location). This facility is currently used by the metadata cache (cf. section 4.3.2) to make its contents persistent, thus enabling faster re-population upon restart.
The interfaces of the instance model are shown in the left subtree and the ones of the meta model in the right subtree.

```
enum GomKind {
    // instance model types
    GenObjKind, ValKind, NILKind, ObjRefKind, BoolKind,
    CharKind, ShortKind, IntKind, LongKind, DoubleKind,
    StrKind, EnumKind, StructKind, SequenceKind,
    UnionKind, ArrayKind, AnyKind,

    // meta model types
    MetaObjKind, MetaLongKind, MetaFloatKind,
    MetaStringKind, MetaListKind
};

interface GomElement {
    GomKind             GetKind();
    string              AsString();
    GomElement          Copy();
    boolean             Read(in InputStream is);
    boolean             Dump(in OutputStream os);
};
```

Figure 4.4: Interface `GomElement`

The root of all instance model interfaces is `Val`, which models a (proxy) value of a target system. Derived from it are `GenObj` whose proxy instances represent objects of target systems (e.g. a managed object in an OSI agent) and a set of values such as `Int`, `Struct`, `Str` and `Bool` (cf. section 4.2.2 and appendix A.1 for a description).
The root of all metadata model interfaces is `MetaElement`. Its subclasses are `MetaLong`, `MetaFloat`, `MetaString`, `MetaList` and `MetaObj` (cf. fig. 4.16 on page 58).
A client of GOM will mostly use the *instance model* whereas an adapter implementor that needs access to metadata will also have to deal with the *meta model*.
The instance- and meta models are explained in the next two sections.

4.2.2 Instance Model

The instance model is that part of GOM that a client will use most often. Its task is to provide a uniform abstraction model for instances of other object models (i.e. *target instances*) by furnishing corresponding *proxy instances* for target instances, encapsulating knowledge of how to access the targets. Proxy instances are always instances of the interface `GenObj` and operations invoked on them are transparently dispatched to the target system they represent.
The instance model is *generic* in that it is a synthesis of features common to object-oriented models. Most object models provide classes; classes can be instantiated; instances have attributes and operations; operations have parameters and a return value and so on.

Target Model	**GOM Model**
Class `X`	Interface `GenObj`
Instance of `X`	Instance of `GenObj` (name == `"X"`)
Type	Subclass of interface `Val`
Value (Instance of Type)	Instance of subclass of `Val`

Table 4.1: Mapping of target system elements to GOM

GOM acknowledges this situation and tries to provide a common, *reified* instance model that uses the commonalities between a number of models and tries to reconcile their differences. Reification [Mae87, Cha94] is the ability to represent concepts of a system through elements of the system itself and is used by GOM in the sense that all major elements of GOM are modeled as objects (IDL interfaces).[2]

Therefore, classes of *any* target model are represented as instances of the IDL interface (`GenObj`), e.g. a managed object of GDMO template X in an OSI agent will not – as in the case of XoJIDM (cf. 3.1.2) – be mapped to an instance of IDL interface X, but always to an instance of IDL interface `GenObj` with class name "X".

By having no concept of classes in the instance model, but knowing only instances, the generic object model of GOM resembles those of *prototype-based* models such as SELF [UCCH91, US91].

As shown in table 4.1, instances of any target system class are always represented as instances of `GenObj`, with the target system's class name parameterized within the equivalent `GenObj` (proxy) instance.

Target system *types* (e.g. a CORBA `long`, `struct`, or an ASN.1 `SEQUENCE`) are mapped to a subclass of `Val`. There are a finite number of GOM values and all values of all possible target systems should map to one of those. An *instance* of a target system type (a *value*, e.g. `'5'`), is mapped to an *instance* of a subclass of `Val`.

The IDL interfaces of the instance model (`GenObj`, `Val`, `Adapter` and `Factory`) will be explained in section 4.2.2.2.

4.2.2.1 Characteristics of the Instance Model

Uniform Model Mapping elements such as classes, attributes and values of target systems to a finite set of IDL interfaces has the advantage that the type system of GOM is known and that clients do not need to be recompiled because new types are added or existing ones modified. This is in contrast to approaches that statically translate an object model's specification to IDL and make clients include the resulting IDL interfaces. Using static translation, clients have to be recompiled whenever the original specification is modified, or additional classes are added, since the IDL interfaces will have to be regenerated.

[2] Complete (object-oriented) reification would also involve modeling method dispatch, inheritance etc. as objects. However, for our purpose a partially reified object model is sufficient.

Dynamic Access Classes of target systems that were not known when a client application was compiled can nevertheless be manipulated without the need for client recompilation. It is possible to create instances of these classes, get- and set their attributes and invoke operations on them.

Of course, an application must be written accordingly to use this feature. Applications that benefit from a dynamic approach are for example *class browsers*, *interpreted languages* and *roaming agents*.

A class browser offers the capability to inspect classes of a system, their attributes, operations, superclasses and may also allow to ad hoc create instances of them, set values and invoke operations interactively. A browser needs access to metadata about these classes as provided by GOM.

The task of writing an interpreter for the target systems supported by GOM is facilitated by the use of the instance- and meta model. Using such an interpreter, it is possible to interactively – or by writing scripts – manipulate instances of all target systems supported by GOM, i.e. instances can be created, deleted, accessed and their operations invoked. This is especially important for interactive discovery of the classes of existing systems by 'playing' with them in an interactive way. It can also be used for testing purposes, when for example new classes in a target system have to be tested without wanting to write a client application. Finally, small management scripts can be written that perform management chores by manipulation of instances of target systems. An example of an interpreter (GOMscript) built using GOM is given in section 5.1.

A *roaming agent* is a piece of code that migrates from location to location in a network and performs certain (management) tasks at each location. Assuming that resources at a certain location such as printers, switches, routing tables and password files are represented as objects (e.g. OSI managed objects, objects in a CORBA server or SNMP variables), management of these could be achieved by interpreted code using GOM. This model is especially suited for roaming agent applications that need to handle instances of classes that were not known when the agent was written. A simple roaming agent toolkit that has been written on top of GOMscript demonstrates this point (cf. section 5.2).

Client Independence The dynamic aspect of GOM eliminates the strong dependence of clients on servers that is present in static translation approaches where clients statically include classes generated through translation.

The difference between the static- and dynamic approach is shown in fig. 4.5 (see [BD97] for a comparison of a static with two dynamic approaches).

Rather than statically translating the target model's specification into a corresponding CORBA specification (IDL interfaces) to be used by client management applications through compile-time inclusion as shown in fig. 4.5 (a), the approach taken by GOM (fig. 4.5 (b)) is to generate metadata from a model's specification and collect it in a metadata repository. This repository can subsequently be accessed by adapters at runtime to retrieve metadata about target entities to be manipulated.

This has the advantage that – since clients do not include classes generated from a server specification – they are not dependent on changes to a server's specification. Thus, clients do not have to be recompiled when servers change.

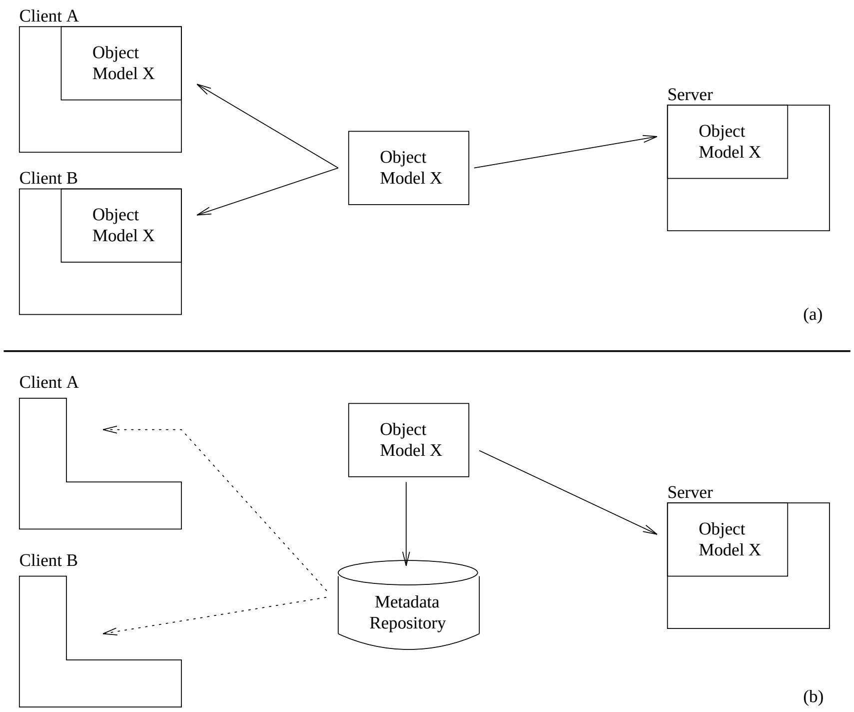

Figure 4.5: Static- and dynamic approaches

Small Client Size Client applications will have a tendency to become bloated since the generated CORBA bindings may – owing to interdependencies between interfaces and types[3] – include a potentially large number of interfaces. Even if only a small set of these is used by the application, it will nevertheless have to pay the price for all of them.
As shown in fig. 4.5, an object model X will be included both by servers and clients in the static approaches, whereas it will be included only by the server in the dynamic approach. Here, the model is included in the metadata repository only. Since clients do not have to include a potentially large amount of generated code, they are typically smaller (in terms of memory) and therefore use fewer resources.
Taking into account that applications may additionally have to include libraries for graphical user interfaces, networking, database access etc., to prevent them from becoming too large, it is an advantage if only a small, finite set of interfaces need to be included, thus reducing at least one component that bloats client code.

4.2.2.2 Interfaces

In this section the CORBA IDL interfaces that comprise the instance model of GOM will be explained. Note that only excerpts of the IDL code will be shown, for the complete definitions refer to appendix A.1.

Val The CORBA interfaces for generic values of GOM are shown in fig. 4.6.

[3] E.g. when interface `A` is derived from or contains interface `B`, it will have to include interface `B`'s specification. It will also have to recursively include `B`'s dependencies even if it does not need them.

```
interface Val : GomElement {};
interface NIL : Val    {};
interface Bool : Val   { attribute boolean  val;      };
interface Char : Val   { attribute char     val;      };
interface Short : Val  { attribute short    val;      };
interface Int : Val    { attribute long     val;      };
interface Long : Val   { attribute long     val;      };
interface Double : Val { attribute double   val;      };
interface Str : Val    { attribute string   val;      };
interface Enum : Str   { attribute long     long_val; };

interface Struct : Val {
    Val               Get(in string key);
    Val               GetIndex(in long index);
    boolean           Set(in string key, in Val val);
    boolean           Add(in string key, in Val val);
    boolean           Remove(in string key);
    long              Size();
};

interface Sequence : Val {
    boolean           Add(in Val new_val);
    long              Size();
    Val               At(in long index);
};

interface Union : Val {
    attribute string  name;
    attribute Val     val;
};

interface Array : Val {
    Val               Get(in long index);
    boolean           Set(in long index, in Val new_val);
    long              Size();
};
```

Figure 4.6: Interface Val

The values present in the generic object model currently represent an *intersection* of values present in the two object models CORBA and CMIP. It should be possible to represent most values of other target systems using only these generic values. If, however, a value of a target system cannot be represented using the available set of generic values, then the generic model would have to be extended.
Every value is derived from the CORBA interface `Val` which is itself derived from `GomElement` (described on p. 41).
Interface `NIL` represents the NULL value, i.e. no value. `Bool` represents boolean values (true or false). The interfaces `Char`, `Short`, `Int`, `Long`, `Double`, `Str` and `Enum` represent simple values, whereas `Struct`, `Sequence`, `Union`, `Array` and `Any` represent *constructed* values.
The `Struct` interface represents struct values that have a (finite) number of member elements of which each has a name and another value (subclass of `Val`). There are a few operations that manipulate the contents of a struct.
The `Sequence` interface models *lists* of values. Operation `Add` adds a value to the list, `At` returns the value at a certain index.
The interface `Union` models a union (as known in C). It has a name and a value as members which can be replaced by other values.
Interface `Array` is similar to `Sequence`, but its size cannot be modified after its instantiation.

GenObj An instance of `GenObj` is a *proxy* object for a real object in a target system (e.g. a managed object in an OSI agent, or an SNMP variable). Operations invoked on it will be transparently forwarded to the corresponding object in the target system, using metadata available about that system maintained by the metadata repository (cf. 4.3).
A `GenObj` is always a value too because it is derived from `Val` and can therefore be used whenever a `Val` is expected, e.g. as argument to an operation.
The definition of `GenObj` is shown in fig. 4.7.
Each instance of a `GenObj` has as attribute a reference to the adapter that created it. This is necessary to forward requests sent to a proxy instance directly to the corresponding adapter. Each instance also maintains the classname of the target instance it represents (e.g. "circuit" for a GDMO managed object), and an instance name that may be set by the adapter that created the object to give it a (symbolic) name. The latter can for example be used to store the *distinguished name* in the case of an OSI managed object.
The attribute `properties` is important for attaching additional state to a proxy representing a target instance that cannot be accommodated using the normal set of member attributes of `GenObj`. It is a dictionary containing strings as keys and subclasses of `Val` as values. The `Dictionary` interface has operations for associating a key with a certain value and for retrieving a value associated with a certain key. A possible use could be for storing the object identifier (OID) of a GDMO managed object template.
In a future version properties will probably be replaced by OMG's *property service* [Obj95] which offers a standardized way of attaching information to any CORBA object without modification of the object's type.
A number of operations are defined in interface `GenObj` to get- and set attribute values, to invoke operations and to delete target instances.

```
interface GenObj : Val {  // An object can also be a value
    attribute Adapter    adapter;
    attribute string     classname;
    attribute string     instance_name;
    attribute Dictionary properties;

    Val                 Get(in string attrname) raises(GenEx);
    void                GetN(in Dictionary values) raises(GenEx);

    void                Set(in string attrname, in Val new_val) raises(GenEx);
    void                SetN(in Dictionary values) raises(GenEx);

    Val                 Execute(in string opname, in Arglist args) raises(GenEx);
    void                Delete() raises(GenEx);

    MetaObj             GetClassDef() raises(GenEx);
    MetaObj             GetAttributeDef(in string attrname) raises(GenEx);
    MetaObj             GetOperationDef(in string opname) raises(GenEx);
    MetaObj             GetElementDef(in string element_name) raises(GenEx);

    Val                 GetProperty(in string name);
    boolean             SetProperty(in string name, in Val new_val);

    string              GetPolicy();
    void                SetPolicy(in string new_policy);
};
```

Figure 4.7: Interface `GenObj`

The `Get` operation requires as its argument the name of the attribute to be retrieved from the target instance and – if found – returns this value in the form of a generic GOM value. If not found (e.g. if consulting the metadata reveals that there is no attribute with the specified name in the target instance's class), or if the target instance cannot be reached, an exception of type `GenEx` is thrown (cf. section 4.2.2.3).
Instances of `GenObj` may cache attribute values for faster access, by storing them locally in the proxy instance and returning the local copy rather than the remote target value. Not having to access the target instance every time a request for an attribute value is received results in improved response time and reduced network traffic. Also, values can still be returned for cached attributes in case a target instance is not reachable, e.g. due to a network partition.
A *caching policy*, which is a description of the way attribute values should be cached, can be specified either on individual proxies (operation `GenObj::SetPolicy` or on all proxies of a certain object model (operation `Factory::SetPolicy`. The latter allows to set a default policy for all objects that will be created, but still enables modification of the caching policy for a single object. The argument to both operations is in string form, which allows to specify various policies and which has the advantage that the signature of the operation does not have to be changed when new policies are added. Currently, three policies are specified:

- "ttl=X": Specifies a time to live (in seconds). After this time has elapsed, the attribute will be retrieved from the target instance when a GET-request for an attribute value is received.
- "never_cache": Always access the target instance. This in effect turns off caching.
- "always_cache": Once the attribute value has been retrieved from the target instance, it will be cached and only the cached value will be returned.

Note that in strongly typed languages such as C++, in order to use the value it has to be narrowed to its actual value, e.g. from `Val` to an instance of `Long`. A type-safe narrowing mechanism for C++ is presented in section 4.2.4.1. Bindings for other strongly-typed languages may provide similar narrowing mechanisms.
`GetN`[4] allows to retrieve a number of attributes in a single operation. Its parameter is a dictionary which contains keys (in the form of strings) and values associated with the keys. To perform a get-operation on multiple attributes of an object, the dictionary has to be populated with key/value pairs where the value part is NULL. Upon successful completion of the operation, the corresponding values will be provided for those keys whose attributes were successfully retrieved.
The `Set` operation sets an attribute value in a target instance and requires the name of the attribute and the new value as arguments. If the attribute cannot be found, the target instance cannot be reached, or if the value cannot be set because it is read-only, an exception of type `GenEx` is thrown describing the error condition (cf. 4.2.2.3).
Similar to `GetN`, `SetN` allows to set multiple attributes in a single operation call. Its only parameter is a dictionary containing the attributes to be set together with the new values.

[4]Note that since overloading is not allowed by IDL, a second operation called `Get` is not valid.

Operation `Execute` invokes an operation of a target instance. It requires the name of the operation and a list of arguments as an instance of `Arglist`. This interface represents a variable number of arguments and offers operations to add, remove and traverse its members. A typical GOM operation call would first add all arguments to an instance of `Arglist` and then call `Execute` with the name of the operation to be invoked and the instance of `Arglist`. Note that *convenience bindings* (cf. 4.2.4) may offer a more convenient interface to operation execution on top of `Execute`, e.g. in the case of C++ a method using *variable argument lists* (cf. [Str91]).

The return value is an instance of a subclass of `Val` and can be downcast to the actual class using the narrowing mechanisms available for each language binding as described in 4.2.4.

If arguments are modified by the operation call, then these are available in the `Arglist` instance at the same position as before the call and can be retrieved by the client. For a discussion of how CORBA's IN, OUT and INOUT arguments are handled see section 4.4.3.

In case of an error condition, an exception of type `GenEx` will be thrown.

The operations `GetClassDef`, `GetAttributeDef`, `GetOperationDef` and `GetElementDef` allow a client to retrieve metadata about the class of the target instance which the `GenObj` proxy object encapsulates (cf. 4.3).

`GetClassDef` returns metadata about the target instance's class in the form of a `MetaObj` (cf. appendix A.2), `GetAttributeDef` returns information about an attribute and `GetOperationDef` about an operation of the target instance's class. These are *convenience* operations for the *CORBA layout* (cf. 4.2.3.2) and assume the strings `"interfaces"` for the class metadata, `"attributes"` for the attribute metadata and `"operations"` for information about a certain operation.

As an example, `GetAttributeDef` called on an instance of `GenObj` will be forwarded to the corresponding *adapter*, which will first retrieve metadata about the class of the target instance, using operation `Find` of the metadata repository (see 4.3.2) with the name of the object model (stored in the adapter, e.g. `"CORBA"`), the string `"interfaces"` as key and the name of the CORBA interface (e.g. `"Printer"`) as arguments. If found, the returned instance of `MetaObj` should contain all information about the desired CORBA interface. The next step is to search the dictionary of `MetaObj` for the key `"attributes"`. If found, the adapter will return it as result of the `GetAttributeDef` operation.

If the CORBA layout is not used, the convenience operations cannot be used and the more generic operation `GetElementDef` has to be used which allows to retrieve metadata about any element of the target instance's class, e.g. `"cond_packages"` in the case of the GDMO layout (cf. B.2).

Operations `SetProperty` and `GetProperty` allow to attach or retrieve additional state to an instance of `GenObj` (cf. discussion of attribute `properties` above).

The target systems supported by GOM may have different mechanisms and semantics for operation invocation such as *group communication* [Maf95] where an operation call may return a number of results (or will use the first result returned), or asynchronous operations where an operation invocation is handled by a separate thread. An asynchronous operation returns immediately and the result can be fetched at a later time. These issues will be dealt with in sections 4.6.2.3 (group communication) and 4.6.5 (asynchronous

```
interface Adapter {
    attribute string  object_model;
    GenObj            Create(in string classname, in string inst_name,
                             in string target_location, in Arglist args);

    Val               Get(in GenObj objref, in string attrname);
    void              GetN(in GenObj objref, in Dictionary values);

    void              Set(in GenObj objref, in string attrname, in Val new_val);
    void              SetN(in GenObj objref, in Dictionary values);

    Val               Execute(in GenObj objref, in string opname,
                              in Arglist args);
    void              Delete(in GenObj objref);

    MetaObj           GetClassDef(in GenObj objref);
    MetaObj           GetAttributeDef(in GenObj objref, in string attrname);
    MetaObj           GetOperationDef(in GenObj objref, in string opname);
    MetaObj           GetElementDef(in GenObj objref, in string element_name);

    ProxyFilter       CreateFilter(in string type, // class or struct
                                   in string target_location,
                                   in string target_name,
                                   in Arglist attrs,
                                   in ConsumerList consumers);

    void              SendEvent(in EventInfo event_info,
                                in string destination_address);
};
```

Figure 4.8: Interface `Adapter`

operations).

Another important issue concerning proxy-based systems is what should be done when the target instance represented by a GOM proxy instance is currently unavailable for reasons such as communication problems, target instance death or a crash of the target instance's server. These problems will be tackled in section 4.6.3.

Adapter The IDL code for interface `Adapter` is shown in fig. 4.8.

An adapter is the bridge between the generic object model and exactly one target object model (cf. section 4.4). It knows how to interact with the target system and mainly has to perform type-checking and conversion of requests between GOM and the target system.

Since all requests sent to a proxy instance will immediately be forwarded to the corresponding adapter (to which each proxy has a reference), the interface of `Adapter` is

```
interface Factory {
    GenObj   Create(in string object_model, in string classname,
                    in string inst_name,
                    in string proxy_location,
                    in string target_location,
                    in Arglist args) raises(GenEx);

    Val      GetConstant(in string object_model, in string const_name);
    string   GetPolicy(in string object_model);
    void     SetPolicy(in string object_model, in string new_policy);
};
```

Figure 4.9: Interface `Factory`

almost the same as that of `GenObj`. One difference is that operations `Get`, `Set`, `Execute` and `Delete` additionally have a reference to the proxy instance as first argument which can be used by an adapter to retrieve information from the GOM proxy when needed.
Operations `GetClassDef`, `GetAttributeDef`, `GetOperationDef` and `GetElementDef` offer access to metadata of the target object's class (cf. section 4.4).
Operations `CreateFilter` and `SendEvent` are part of GOM's generic event handling model and are described in section 4.5.2.
The `Adapter` interface is *abstract*, that is no instances of it can be created. Its main goal is to dictate a common set of operations to subclasses. Any adapter in GOM has to be derived from this interface and implement all operations of `Adapter`.

Factory The OMG IDL interface for the GOM *Factory* is shown in fig. 4.9.
The factory is used for the creation of (a) proxy instances (instances of `GenObj`) and (b) values that represent target model *constants*.
Operation `Factory::Create` is used to create local or remote proxies. Its arguments are the name of the target object model (e.g. "CMIP" or "CORBA") in which the target instance will be created, the name of the class[5], the instance's name (may be empty), the location of the proxy instance, the location[6] of the entity in which the target instance is to be created and an argument list (as described in the `Execute` operation above).
Parameter `proxy_location` allows to create the proxy instance at a specified location (e.g. a CORBA server). To create proxies locally – e.g. in the client's address space – this argument has to be null (e.g. a NULL pointer in the C++ language bindings). To create them at a remote location, `proxy_location` indicates the location where the proxy instance should be created. Remote proxies were discussed in section 4.1.1.
Parameter `inst_name` can be used to attach a description to an instance, e.g. in the case of CMIP it could represent the distinguished name ([ITU92a]).

[5]If modules or similar name space structuring mechanisms are used, then the fully specified name has to be used.
[6]Target system dependent, e.g. an AE-title in CMIP [ITU92a], or the name of a CORBA server.

```
enum completion_status {COMPLETED_YES, COMPLETED_NO, COMPLETED_MAYBE};

exception X {
    unsigned long      minor;
    completion_status  completed;
};
```

Figure 4.10: Example of OMG IDL exception

To create values that represent constants in target systems, operation `GetConstant` can be used. It accepts as arguments the name of the target object model and the fully specified name of the target constant, e.g. `"MyDocuments::Account::max_limit"`.
Operations `GetPolicy` and `SetPolicy` modify or retrieve the current caching policy for all proxy instances of a certain object model.
This is different from the corresponding operations in `GenObj` which modify the caching policy on the basis of single instances.

4.2.2.3 Error Model

The error model of GOM is based on *exceptions*. An exception is an interruption of the normal invoke/response control flow of an operation invocation in an error situation. They offer an additional exit from an operation call signaling an error condition and describing the cause of the exception.
A number of operations may raise a `GenEx` exception (cf. appendix A.1). It contains attributes `name` which is the name of the exception in the target model, `ex_type` which defines whether the exception occurred in GOM or in the target model (see below) and `members` which is a dictionary that contains strings as keys and subclasses of `Val` as values. Since exceptions are similar to structs, the generic interface `GenEx` should allow to represent any target model exceptions.
An exception defined in OMG IDL as shown in fig. 4.10 would contain `"X"` as its name and the keys of `members` would be `"minor"` with an instance of `Long` as value and `"completed"` with an instance of `Enum` as value.
A `GenEx` is thrown by an *adapter* (1) when it cannot dispatch the operation to the target instance or another error occurred in GOM, or (2) when the operation of the target instance signals an error condition *itself*.
The first case may for example involve not being able to locate the target instance, or a type mismatch between a GOM generic value and its formal parameter as defined in the metadata. In this case, the operation is not even dispatched to the target instance, but an exception is thrown immediately. Attribute `ex_type` will contain `GOM_EX` as value. To describe the error condition, two entries will be inserted into `members`: `"errcode"` indicates the error in a programmatic manner[7] and `"errstr"` describes the error condition in string form.

[7] GOM's error codes are described in the enumeration `GomErrcode` in appendix A.1.

In the second case, GOM was able to dispatch the request to the target instance, but there an error condition ensued and was returned, either as return value or in the form of an exception. In this case, `ex_type` will contain `TARGET_EX`.
If a value was returned by the operation invocation describing the error condition, then GOM will not throw an exception, but convert the value as usual and return it. If an exception was thrown, then GOM will convert the target model specific exception into a `GenEx` and throw that on to the caller. It is the caller's responsibility to know the names of the fields an exception contains (its members) and retrieve them accordingly to handle the error condition.

4.2.2.4 Use Cases

The intention of these use cases is to show the control flows through the various components of GOM for a number of operations.

Creation Operation `Factory::Create` creates both a target- and a corresponding proxy instance at the specified locations.

1. The client has to create a factory object. This will typically be done once (at the start of the client application) and the instance will be stored in a variable for further reference.
2. The factory object is used to create a new instance using operation `Create`.
3. The factory object locates a corresponding adapter to which the creation request can be forwarded by constructing the name of the adapter through concatenation of the object model given and the suffix `"_Adapter"`.
4. The naming service at location `proxy_location` is searched for an instance of the adapter with the given name. If `proxy_location` is NULL, then the local naming service will be used. If no adapter with the specified name has been found, a new instance will be created at location `proxy_location` using the constructed name as *interface name* and registered with the naming service, using the constructed name (string) as key.
5. The creation request is dispatched to the adapter.
6. The adapter tries to create the underlying object (e.g. an OSI managed object, CORBA object etc) at location `target_location`. If successful, it creates a `GenObj` proxy instance and stores a reference to the adapter in the newly created object (attribute `adapter`). Further requests sent to the proxy will be forwarded to the adapter that created it.
7. `The GenObj` proxy instance is returned. An exception of type `GenEx` is thrown if the instance cannot be created.

```
GenObj_ptr      inst;
Factory_ptr     f=new Factory;
try {
    inst=f->Create("CORBA", "ZRL::Printer", "psp31", 0,
                   "adlerhorn.zurich.ibm.com", 0);
}
catch(GenEx& ex) { /* Error */  }
```

Figure 4.11: Creation of a GOM instance representing a CORBA target instance

```
GenObj_ptr      inst;
Factory_ptr     f=new Factory;
try {
    inst=f->Create("CMIP", "customer",
                   "netId=TelcoNet;customerID=(name IBM)", 0,
                   "agentID=266352",0);
}
catch(GenEx& ex) { /* Error */  }
```

Figure 4.12: Creation of a GOM instance representing a CMIP target instance

In figure 4.11 an example is given that shows how a client may create a new CORBA instance (using the C++ language binding).
Fig. 4.12 shows how a CMIP instance is created.
Essentially the code is the same, but rather than specifying CORBA as object model, CMIP is used. Also, the distinguished name of the managed object to be created as target instance is given in `inst_name`.[8] The location of the target instance is in this case the AE title of the agent in which the managed object is to be created.

Deletion The `Delete` operation of `GenObj` deletes first the target instance and – if successful – also the proxy instance. If the target instance cannot be deleted, an exception of type `GenEx` is thrown, indicating the cause of the failure.
Note that in some cases (such as CMIP [ITU92a]), objects may contain *subordinate* objects (children objects) that may have to be deleted before the parent object can be deleted. Trying to delete an instance containing children will result in an exception.
Deletion of a proxy instance by other means such as the `delete` operator in the C++ bindings will only cause deletion of the proxy, but not of the target instance.

Getting the value of an attribute

[8] In *string syntax* as defined in [GMR94].

```
GenObj_ptr      person;
Long_ptr        age;
try {
    age=(Long_ptr)person->Get("age");
    cout << "Age is " << age->val() << endl;
}
catch(GenEx& ex) { /* Error */ }
```

Figure 4.13: Getting an attribute value of a GOM instance

1. Operation `Get` is invoked on a proxy instance.
2. The operation is forwarded to the corresponding adapter whose reference is stored in the instance. The CORBA object reference is added as argument so that the adapter can retrieve more information about the proxy.
3. Depending on the caching policy (see above), if the attribute value is cached, it will be returned immediately.
4. The attribute value is retrieved from the target instance (as identified by the proxy instance) using target system specific calls (e.g. DII or CMIP) and returned to the client.

Example code (C++ bindings) is shown in fig. 4.13.
Note that *convenience bindings* for strongly typed languages may provide a safe narrowing mechanism (cf. 4.2.4). It can be used to replace the unsafe C++ cast operation performed in fig. 4.13.

Setting the value of an attribute

1. The `Set` operation is invoked on a proxy instance.
2. The operation is forwarded to the corresponding adapter whose reference is stored in the instance, adding the object reference of the proxy to the operation.
3. The adapter retrieves meta-information about the type of the attribute to be set.
4. The GOM value that is to be used as the new value is type- checked against the meta information. This involves checking type compatibility between the GOM value and the meta information for the attribute of the target instance.
5. The GOM value is converted to an object model specific value.
6. This value is then set in the target instance.
7. If successful, it is also set in the proxy (if attribute caching is enabled).

Code that demonstrates how to set an attribute value in a proxy is shown in fig. 4.14.

```
GenObj_ptr      person;
Long_ptr        age=new Long;
try {
    age->val(32);
    person->Set("age", age);
}
catch(GenEx& ex) { /* Error */ }
```

Figure 4.14: Setting an attribute value of a GOM instance

Invoking an operation

1. Operation `Execute` is invoked on a proxy instance.

2. The operation is forwarded to the corresponding adapter whose reference is stored in the instance, adding the object reference of the proxy to the operation.

3. The adapter fetches meta information about the operation and performs type-checking on the arguments provided in the argument list.

4. All generic values of the argument list are converted to corresponding values of the underlying object model.

5. The operation is invoked in the target instance.

6. Parameters of the argument list that have *reference* semantics (i.e. they can be modified by the operation, e.g. CORBA's IN/OUT parameters) are set accordingly.

7. The result is converted to a generic value and returned.

Fig. 4.15 shows sample code invoking an operation on a proxy instance.
First the argument list is prepared. In the example, two values (a string and a short) are added to the argument list. Then the operation is invoked and the result stored in a variable. When an error occurs, e.g. if types are incompatible or a logical error occurs, then an exception of type `GenEx` is thrown.

It is essential for adapters to have access to meta information about the target classes to perform type-checking and conversion of arguments (cf. 4.4).
The code example shown in this section are for the C++ language bindings of GOM's IDL interfaces. Clients, however, may choose any language for which a binding exists for access and manipulation of GOM proxy instances. *Convenience bindings* as described in section 4.2.4 provide a more elegant layer on top of a language binding to be used by client applications.

```
GenObj_ptr      printer;
Arglist_ptr     args=new Arglist;
Bool_ptr        ret;
Str_ptr         doc=new Str;
Short_ptr       copies=new Short;
try {
    doc->val("/u/bba/.profile");
    copies->val(3);
    args->Add("document", doc);
    args->Add("number_of_copies", copies);
    ret=(Bool_ptr)printer->Execute(args);
}
catch(GenEx& ex) { /* Error */ }
```

Figure 4.15: Invoking an operation on a proxy instance

4.2.3 Meta Model

The generic meta model forms an important part of GOM. Its major task is to provide metadata description of the classes, their attributes, operations etc. in a system. Its major goals are:

Simplicity CORBA's interface repository is rather complex. Special care has to be taken when to release memory to avoid memory leaks.

The meta model proposed here supports a very simple meta model, i.e. it has fewer elements than for example CORBA's interface repository, a clear memory management policy and a simple inheritance structure.

Ease of Use A simple conceptual meta model should contribute to ease of use for clients.

Integration with GOM The meta model should be integrated seamlessly with the instance model (cf. 4.2.2) to preserve the homogeneity of GOM. The uniformity of GOM would be destroyed by offering one type of syntax and semantics for the instance model and a different one for the meta model.

Extensibility CORBA's interface repository has been designed for CORBA interfaces, without the possibility of extension for use with other object models. GOM's meta model tries to take into account from the beginning the possibility of extension to accommodate metadata of other object models.

The task of a meta model is to provide metadata about the *elements* (e.g. *module*-, *class*-, *attribute*- or *operation* descriptions) a model consists of. A meta model usually comprises a finite number of elements since the syntax and semantics of the model it describes usually never changes. If, however, a non-finite *set* of models has to be described using the *same* meta model, providing a finite number of metadata elements to represent all the potential object models is likely to be too inflexible since this scheme has the drawback that combining a number of object models usually results in meta models that either follow an approach where the features of *all* models are combined to produce a common model (*union approach*), or the common model will consist only of features that occur in *every* model (*intersection approach*).[9] Both approaches are likely to require modification of the common model when a new model has to be integrated.[10]

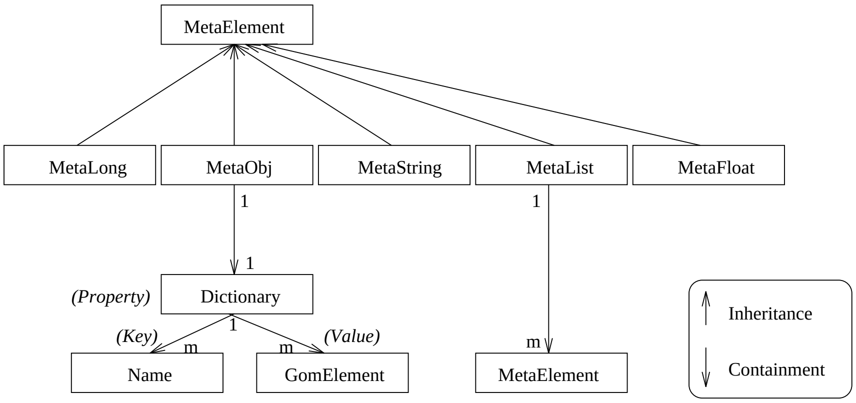

Figure 4.16: The meta model of GOM

4.2.3.1 Interfaces

The central idea of the generic meta model proposed here takes into account that the elements of a *generic* meta model have to be *generic* as well and cannot be represented as compile time classes or interfaces. Therefore a *generic* element (`MetaObj`) is proposed (see fig. 4.16) that contains only a set of *properties* (a dictionary) and a name.
A property denotes one aspect of the generic element and is an association between a *key* (string) and a *value* (the referred-to generic element). The value is a subclass of `GomElement` (cf. section 4.2.2) and is in most cases a subclass of `MetaElement`. However, since it is in certain cases necessary to model *instance model values*, e.g. to represent constants or default values for an attribute, the value of an association may also be a subclass of `Val`. To determine which GOM element the value of an association is, operation `GetKind()` can be used.
A `MetaObj` instance is used in GOM's generic meta model to represent a metadata element of a target object model such as a *package* in GDMO or an *IDL interface* in CORBA.
Instances of `MetaObj` can recursively contain another instance of `MetaObj`, or an instance of a subclass of `MetaElement`. All possible structures of metadata can be constructed using only the subclasses of `MetaElement` provided by the generic meta model. These are `MetaLong`, `MetaFloat`, `MetaString`, `MetaList` and `MetaObj`.
As this model contains only a small number of elements having a flat inheritance structure, it is conceptually simple and easy to use. Moreover, it should be possible to represent

[9] An intersection approach may e.g. omit *module* elements since they may not occur in every model.
[10] Of course, the more models are integrated, the smaller the probability of a modification of the common model since it may already contain the new features.

most metadata elements through a combination of these types.[11]
The IDL definitions for these generic elements can be found in appendix A.2.
An aspect described by a property may be the attributes contained in a class element. To retrieve the attributes of a class, the dictionary of the `MetaObj` representing the class element would be searched for an entry with `key == "attributes"` and return the corresponding value, in this case another instance of `MetaObj`. The returned instance (which represents a list of all attributes of a class) could then be used as starting point for another query, e.g. for the property named `"age"` of the attribute element which points to an instance of `MetaObj` that describes the attribute `age` of the class element. Fig. 4.17 shows how the metadata tree is structured for the attribute `age` of IDL interface `Person` shown in fig. 4.18.

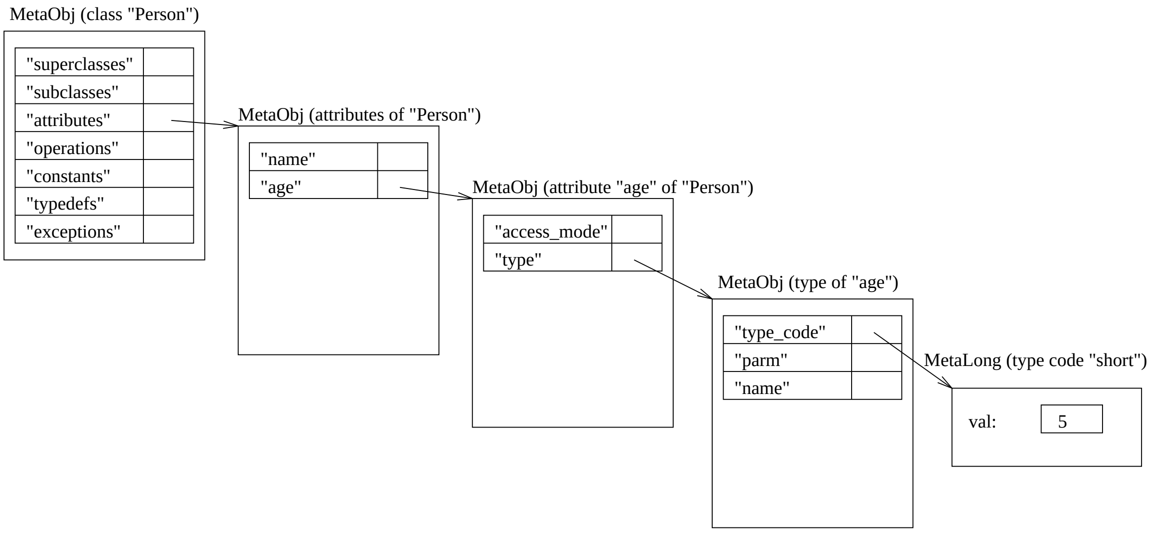

Figure 4.17: Metadata tree for attribute `"age"`

```
interface Person {
    attribute string name;
    attribute short  age;
};
```

Figure 4.18: IDL interface `Person`

To determine the type of attribute `"age"` the following steps have to be taken: first, the metadata about IDL interface `"Person"` has to be retrieved (see section 4.3.2 for an explanation of how to do this). Then property `"attributes"` has to be dereferenced to obtain a dictionary containing all attributes of the interface. Following property `"age"`,

[11] An exception is the representation of *values* in a meta model, e.g. in the case of *default values* as used in ASN.1. Since default values can adopt any ASN.1 type, additional types may need to be added to the generic meta model.

one reaches the corresponding attribute's definition. Then property `"type"` is chosen which describes the type of the attribute. Finally, property `"type_code"` refers to an instance of `MetaLong`, indicating that attribute `age` is of type `long`.
A definition of the relations between the instances of the meta model is contained in a *layout* (cf. below). The layout that would be used to perform metadata navigation for CORBA as described above can be found in section 4.2.3.2 on page 61 and is listed in detail in appendix B.1.

4.2.3.2 Layout Definitions

The previous discussion dealt with *elements* of a meta model that denote the constituents of an object model represented in metadata.
Elements that frequently occur in object models are for example *classes*, *operations*, *attributes* and *types*. These would typically be represented by a corresponding entity in the meta model describing the element.

> The elements that comprise a meta model and the relations between them shall for the sake of the discussion be called a *layout*. Layouts describe how metadata of a model X is represented in the generic meta model.

A layout defines (a) which elements are available in a meta model, (b) the semantics of the elements and (c) the relations between them, e.g. a *class* element is a template for the creation of instances (definition and semantics) and contains *attribute*- and *operation* elements (relation).
Owing to its genericity, the generic meta model does *not* prescribe a priori a special layout. Any layout can be imposed on it by a metadata adapter, which means that the implementor of a metadata adapter may define the layout of metadata. It is for example possible to define a layout in which classes do not *directly* have attributes and operations, but the latter are contained in *packages* as is the case in GDMO [GDM92] or in *interfaces* as in the case of TINA [TIN95].
It is important that implementors of metadata adapters provide a precise description of their layout in order for clients to be able to use that meta model. A description of a layout has to contain the following things:

1. The elements of the meta model (e.g. modules, classes, packages etc.)[12]. All elements have to be given unique names by which they are identified in the generic meta model (e.g. "constants", "attributes", "type_code" etc.)

2. The semantics of each element.

3. The relations between elements, e.g. "classes contain attributes and operations", or "an attribute contains a type and an access mode".

[12]Note that not all elements of the object model may need to have a corresponding representation in the meta model !

4. The *type code constants* used for each type. Each type code constant is a number that uniquely identifies a type (cf. [OMG95, ch. 6.7.2]), e.g. `5` for `short`, `13` for `struct` etc. A dictionary of type code constants and their string representation has to be returned by every metadata adapter so that type code constants can be mapped to names, e.g. for generic printing functions (cf. operations `GetTypeCodes` and `GetTypeCodeName` in appendix A.4).

5. The mapping description of metadata elements to the instance model. It consists of two parts:

   (a) Mapping of Elements: This is a description of how metadata elements of a layout are mapped to their equivalent representation in the instance model when creating an instance of the element, e.g. how class definitions are mapped to instances of `GenObj` in the instance model.

   (b) Mapping of Types: Describes how metadata type definitions in a layout are represented through values in the instance model. For example, a definition of the CORBA IDL type `long` would become an instance of `Long` in the instance model, or a definition of an ASN.1 `SEQUENCE` type would become an instance of `Struct` in the instance model.

   The latter point is important for two reasons: first, it eliminates implicit underlying assumptions of how a mapping is defined and second, it is important for users of the instance model to know which data types to use as arguments for GOM methods. For example, as in the case above, if the user has to set the value of a GOM instance representing a managed object which has the (ASN.1) data type `SEQUENCE`, then consulting the type mapping table will prescribe the use of the GOM value `Struct`.

Layout definitions are initially provided for the object models of CORBA and CMIP and will be described in the next two sections.
Each section first describes the layout. Then it is shown for all elements of the layout which their corresponding elements are in the instance model. Last, the mapping of layout elements describing types to the instance model is presented.

CORBA Layout The CORBA layout is modeled after the one used in CORBA's interface repository [OMG95, ch. 6.4.4]. It is shown in fig. 4.19.
The CORBA layout contains descriptions of modules, classes, attributes, types, operations, parameters, constants, type definitions and exceptions (cf. appendix B.1 for the full description of the layout). The most important elements are briefly described below.
A module is a container for a collection of metadata and may contain constants, type definitions, exceptions, other modules and classes. Its main use is for structuring the global name space to avoid name collisions.
A class contains a list of super- and subclasses and information about its constants, type definitions, exceptions, attributes and operations.
An attribute has a name and a type (modeled after CORBA's `TypeCode` interface [OMG95, pp. 6-35]) which can be *simple* or *constructed*.

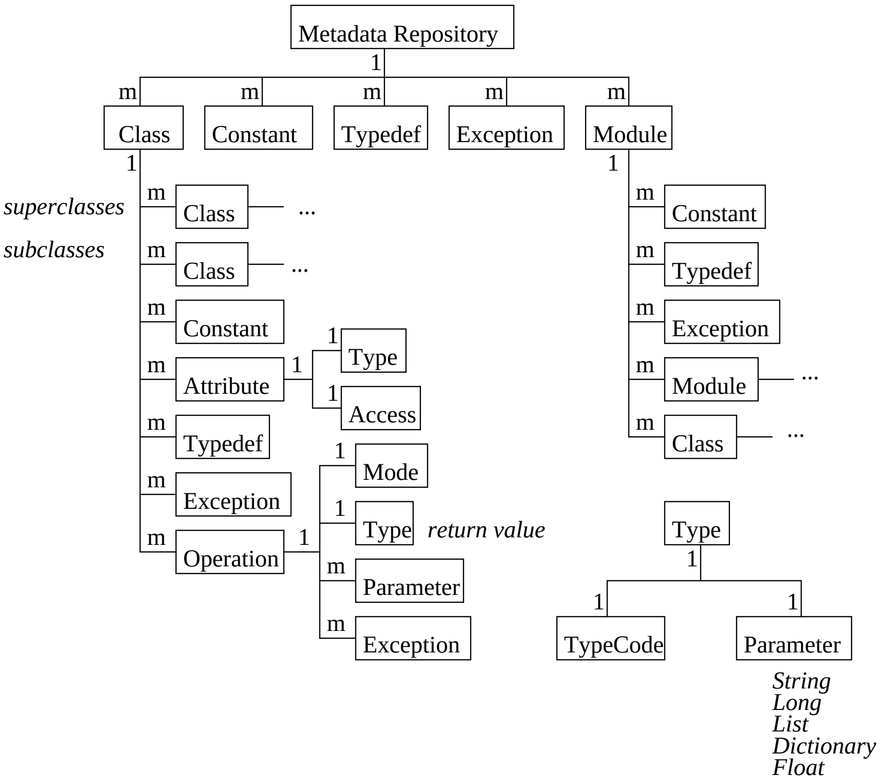

Figure 4.19: CORBA layout

A simple type has a *tag* (`"type_code"` property) which is a type code constant (number) describing the type. Constructed types may use the `"parm"` property to store additional information, e.g. a struct type may store information about its members in the dictionary of an instance of `MetaObj`.
An operation has a number of parameters, a mode (e.g. IN, OUT, INOUT) and a type describing the return value.

Mapping of CORBA Layout Elements to the Instance Model Table 4.2 describes the mapping between elements of the CORBA layout and elements of the instance model.
Not all elements of the meta model are represented in the instance model, e.g. a module is only present in the meta model as a means for structuring the namespace of identifiers and therefore not available in the instance model.
A class is modeled as an instance of `GenObj` in the instance model, i.e. for each creation of a CORBA interface a corresponding instance of `GenObj` will be created in GOM's instance model.
Attributes are not directly represented in the instance model, only their values are. Each attribute has a value with a corresponding type and access mode that are represented in the meta model. The metadata of an attribute can be retrieved in the instance model through method `GetAttributeDef` (cf. appendix A.1).
In agreement with table 4.1 (p. 42), a type in the meta model is instantiated to a value in the instance model, e.g. a type with type code `tk_struct` becomes a `Struct` value in the

Layout Element	Corresponding Element in the Instance Model
Module	No equivalent. Only present in meta model as name space mechanism
Class	Instance of `GenObj`
Attribute	No equivalent. Only present in meta model. Properties of attributes such as access mode can be retrieved through access to the meta model from the instance model (`GetAttributeDef`)
Type	Instance of `Val` or subclasses
Operation	Method `Execute` of `GenObj`
Parameter	Elements of `Arglist` of `Execute`
Constant	Instance of `Val` or subclasses. Constants can be created using the `Factory::GetConstant()` method, which returns a GOM value
Typedef	Instance of `Val` or subclasses. All type definitions are mapped to instances of `Type` in the meta model and therefore the same mapping as for `Type` applies
Exception	Instance of `GenEx` value (cf. section 4.2.2.3)

Table 4.2: Mapping of CORBA elements to the instance model

instance model (cf. table 4.3). The same applies to type definitions. A type definition creates a new type by referring to an existing type and giving it a different name. Type definitions can always be resolved to the original type and therefore are – as in the case of types – mapped to values in the instance model.
An operation element of the meta model is not represented through an object in the instance model, but through method `Execute` of interface `GenObj`. Metadata about an operation can be retrieved in the instance model using method `GetOperationDef` of `GenObj`. Each operation has zero or more parameter elements which have a type and a mode (e.g. IN, OUT) and are mapped to values in the instance model of a parameter list (cf. `Arglist` of `GenObj`).
Constants are mapped to values in the instance model. They can be created using method `Factory::GetConstant` which consults the metadata and creates a new value according to the definition in the meta model.
Exception elements are mapped to instances of `GenEx` (cf. section 4.2.2.3).
Instances in the instance model may have to maintain some additional state (e.g. their distinguished name in the case of GDMO or the repository identifier in CORBA). For that purpose, the `GenObj` interface has an attribute (`properties`) which allows to insert additional information that will be recorded in a dictionary under a chosen name. Using this name, the information can be retrieved later.[13]

Mapping of CORBA Layout Types to the Instance Model Table 4.3 describes the mapping between types as represented in the meta model and their corresponding values in the instance model.

[13] For a more detailed discussion refer to section 4.2.2.

type_code	**parm**	**GOM Value**
tk_null	NULL	NIL
tk_void	NULL	n/a
tk_short	NULL	Short
tk_long	NULL	Long
tk_ushort	NULL	Short
tk_ulong	NULL	Long
tk_float	NULL	Double
tk_double	NULL	Double
tk_boolean	NULL	Bool
tk_char	NULL	Char
tk_octet	NULL	Char
tk_any	NULL	n/a
tk_objref	`MetaString`. Name of IDL interface	GenObj
tk_struct	`MetaObj`. The dictionary contains as key the names of the members and as values their types (cf. `2.4 Type` in section B.1)	Struct
tk_union	`MetaObj`. The property `"discriminator"` of the dictionary points to a type (cf. `2.4 Type` in section B.1) which describes the discriminator. The other keys are the names of the union's members. Each name points to another instance of `MetaObj` which contains 2 properties: "type" and "label_value". The first describes the type, the second the value of the label (according to the type of the union's discriminator)	Union
tk_enum	`MetaList`. The list contains the strings (`MetaString`) that name the elements of the enumeration	Enum
tk_string	NULL or `MetaLong` (length of string)	Str
tk_sequence	`MetaObj`. The dictionary contains 2 properties: `"type"` and `"size"`. The first points to a type (cf. `2.4 Type` in section B.1) and the second points to a `MetaLong` which (if not NULL) contains the length of the sequence	Sequence
tk_array	`MetaObj`. Same as tk_sequence. Length is always present.	array
tk_except	`MetaObj`. Same as tk_struct	GenEx
Recursive	n/a	n/a

Table 4.3: Structure of types in the generic meta model using the CORBA layout

The table lists the types available in the CORBA layout, with the first column showing the type code each type is assigned. The middle column shows the parameters of each type. A parameter can be used to provide further information about a type and is used to model constructed (or aggregate) types. A struct type e.g. will typically provide further information about its members whereas a simple type such as `boolean` will not make use of the parameter. The rightmost column shows the instance model types to which the meta model types are mapped.
The mechanism for using parameters to represent constructed types is essentially the same as the one used in CORBA (cf. table 12 in [OMG95, p. 6-37]).
As the number of types in the instance model is smaller than in the meta model, multiple meta model types will map to the same instance model type, e.g. the types `tk_float` and `tk_double` of the meta model both map to the instance model type of `Double`.
The table can be used by clients to determine which value to instantiate, e.g. as argument to an operation. If for instance an operation of a GOM proxy instance representing a CORBA object requires a formal parameter of type `tk_float`, then an instance of `Double` has to be provided as argument to the `Execute` call.

GDMO Layout An overview of the GDMO layout is given in fig. 4.20.
Compared to OMG IDL, which defines its elements *inline*, i.e., elements are contained within other elements, e.g. an attribute *belongs* to its enclosing interface and can *not* be used by other interfaces[14], GDMO is more flexible in this respect.
In GDMO, the main element is the class template (cf. section 2.2.3). It can include packages which themselves contain attributes, actions (operations) and notifications. Since packages do not *belong* to the template that includes them, they can be used by *many* templates. The same principle applies to other elements, e.g. an attribute can be included in more than one package and therefore be included (indirectly) in more than one class template.[15]
The GDMO layout contains descriptions of name bindings, templates, packages, parameters, attributes, attribute groups, actions and notifications (cf. appendix B.2 for the full description of the layout). The most important elements are briefly described below; for more information refer to [GDM92].
A GDMO name binding element defines constraints for the creation of managed objects within other objects and for their subsequent deletion, e.g. an object may not be deleted if it contains other objects.
A class template element is the definition from which managed objects can be instantiated. It contains a number of mandatory and conditional packages. The contents of mandatory packages always have to be included in the resulting managed object instance, whereas the contents of conditional packages are only included when certain conditions apply (cf. [GDM92]).[16]
Packages contain attributes, actions (operations) and notifications.
An attribute has a name and a syntax (a type) (cf. table 4.5).

[14] Note that, contrary to attributes, *types* do allow out-of-line definition and subsequent inclusion in OMG IDL.
[15] Note, however, that the OID of the two attributes will be different.
[16] The decision which conditional packages are included is taken at instance creation time.

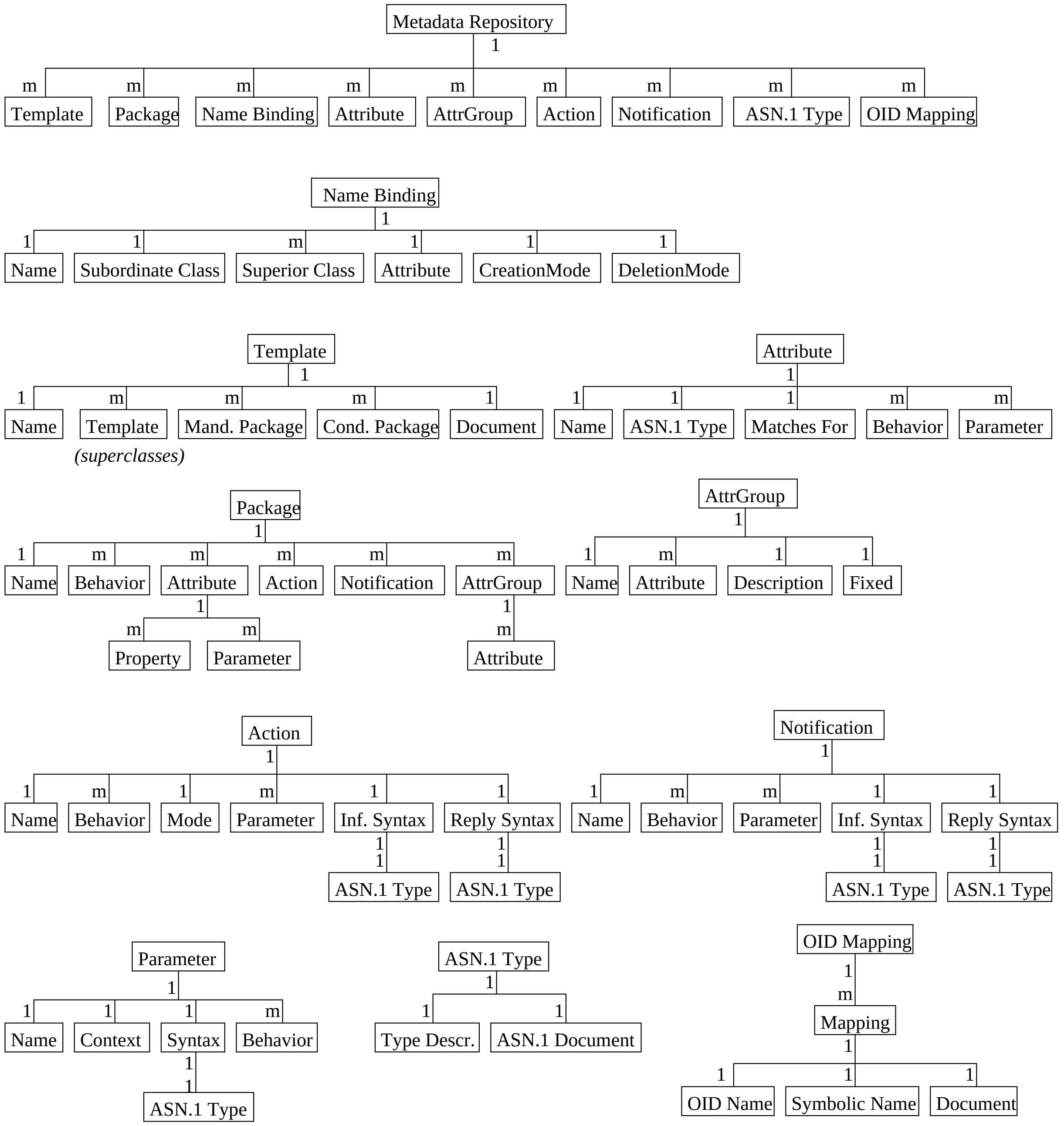

Figure 4.20: GDMO layout

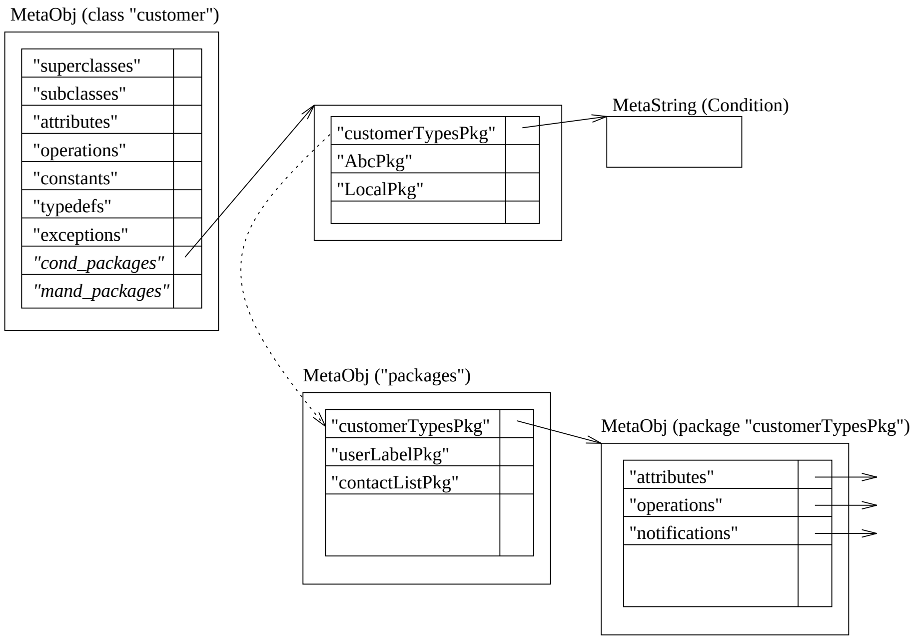

Figure 4.21: Example of GDMO layout for package

An action has a mode (confirmed or unconfirmed), a number of parameters (described by an information syntax) and a return value (described by a return syntax).
Notifications are similar to actions in that they have a number of parameters described by a syntax and a return value, described by a syntax as well.
An example of how the GDMO layout can be used is given in fig. 4.21. It shows how GDMO packages are represented in the meta model, using the GDMO layout.
GDMO templates consist of a number of *mandatory* and *conditional* packages which bundle a number of attributes, operations and notifications (see above).
Although packages are only a compile-time entity and have no runtime representation (their contents 'dwell' in the GDMO template that included them), it is sometimes necessary to record from which package an attribute or operation was taken.
This is done by adding the properties `"cond_packages"` and `"mand_packages"` as shown in fig. 4.21.
In the example both property keys point to an instance of `MetaObj` which has all the packages in its dictionary. By navigating along the values of this dictionary, it is possible to reach the contents of a package (e.g. its attributes, operations and notifications).
This scheme allows to flexibly attach any type of additional information needed to represent a target meta model with the generic one. There is no need to extend the structure of the meta model by subclassing its elements (class extension), but all elements can be extended by modifying their instance structure at runtime (instance extension).

Mapping of GDMO Layout Elements to the Instance Model Table 4.4 describes the mapping between elements of the GDMO layout and elements of the instance model. The name binding element has no equivalent in the instance model. It is consulted by the `Factory` and `GenObj` objects in the instance model to check whether creation or deletion,

Layout Element	**Corresponding Element in the Instance Model**
Name Binding	No equivalent. Name bindings are only present in the meta model and are used by adapters to check correctness of creation- and deletion requests
Class Template	Instance of `GenObj`
Package	No equivalent. The contents of a package are copied to the `GenObj` instance which is the result of the mapping of the template in which the package is included
Parameter	Elements of `Arglist` of `Execute`
Attribute	No equivalent. Only present in meta model. Properties of attributes such as access mode can be retrieved through access to the meta model from the instance model
Attribute Group	No direct equivalent (cf. section 4.6.2.5)
Behavior	No equivalent. Only present in the meta model
Action	Method `Execute` of `GenObj`
Notification	Event (cf. section 4.5)

Table 4.4: Mapping of GDMO elements to the instance model

respectively, are allowed.
A class template element is mapped to an instance of `GenObj`. The contents (i.e. attributes, actions and notifications) of all mandatory packages that it includes are inserted into the resulting instance. The contents of conditional packages are only inserted when certain conditions are met.[17] Note that operations and notifications are only conceptually inserted into the resulting instance since they belong to the class, not the instance. They are always forwarded by the instance to its corresponding adapter for execution.
A package element is not available in the instance model since only its *contents* are included in a resulting instance. The *names* of the packages included by an instance of `GenObj` are recorded as a list (`Sequence` in the instance model) in the `properties` attribute. It can be retrieved by calling method `GetProperty`, i.e.: `GetProperty("packages")`.
Parameter elements are present in the instance model as elements of an `Arglist` of method `GenObj::Execute`. Each element of the argument list is a subclass of `Val`.
Attribute elements are not directly present, but only the values they represent are available.
Attribute groups are handled differently from attributes, since manipulating them (i.e. getting and setting attribute groups) requires *group* mechanisms to be provided. For a more detailed discussion see sections 4.6.2.3 (group communication) and 4.6.2.5 (attribute groups).
Actions are mapped as operations in the instance model (`GenObj::Execute()` method).
The mapping for notification elements is described in more detail in section 4.5.

[17] Which conditional packages are included is determined either by the OSI agent or by the manager in the M-CREATE operation using the `packages` attribute.

Mapping of GDMO Layout Types to the Instance Model Table 4.5 describes the mapping between types as represented in the meta model and their corresponding values in the instance model.

It was the intention to make this mapping as similar as possible to the one for CORBA types described in table 4.3.

The table lists the (ASN.1) types available in the GDMO layout, with the first column showing the type code each type is assigned. The second column shows the parameters of each type which can be used to provide further information about a type and is used to model constructed (or aggregate) types. A struct type e.g. will typically provide further information about its members. The rightmost column shows the instance model types to which the meta model types are mapped.

An ASN.1 type is represented in the meta model through instances of `MetaObj` with the properties `"name"`, `"document"`, `"type_code"` and `"parm"`. The first two specify the type name and ASN.1 document in which it was defined. The third has as its value an instance of `MetaLong` which is the type code for the type. The `"parm"` property can be used to describe *subtype constraints* (cf. below) or to describe constructed types.

ASN.1 values can be simple or constructed. The parameter for simple types is NULL in most cases. It can also be used to represent an ASN.1 subtype construct which further constrains a type (i.e., the values accepted by that type). For example, an integer type may have a subtype constraint specifying that only a certain range of integer values must be accepted. Subtypes are discussed below.

Constructed values use the `"parm"` property to describe the type further, e.g. an ASN.1 `SEQUENCE` which maps to a `Struct` in the instance model will have as its value another instance of `MetaObj`, with the keys containing the names of the members of the struct, and the values containing their types.

A *subtype* further constrains a type (cf. [ASN90, section 4]) and is modeled as follows: for each ASN.1 type T, `"parm"` has as its value an instance of `MetaObj` which contains one of the properties shown in table 4.6.

The two tables can be used by clients to determine which value to instantiate, e.g. as argument to an operation. If for instance an operation of a GOM proxy instance representing a managed object requires a formal parameter of type `SEQUENCE`, then an instance of `Struct` has to be provided as argument to the `GenObj::Execute` call.

4.2.3.3 Modeling Values

In certain situations it may be necessary to model *instance model values* in the meta model. Examples are initial values for *constants* or default values for attributes. In most cases, the subclasses of `MetaElement` will suffice to represent these values. In certain cases, however, a value may be more complex. For this purpose, a value may be modeled using subclasses of `Val`. For this purpose, the dictionary present in `MetaObj` contains keys and values, where the key is a string and the value a subclass of `GomElement`. In most cases, `GomElement` can be narrowed to an instance of a subclass of `MetaElement`, but in some cases it will have to be downcast to an instance of a subclass of `Val`. What kind of GOM element it is can be determined using operation `GetKind()` of `GomElement`.

type_code	**parm**	**GOM Value**
BOOLEAN	NULL	Bool
INTEGER	NULL or *subtype* (cf. below)	Int
BITSTRING	NULL or *subtype* (cf. below)	Str
OCTETSTRING	NULL or *subtype* (cf. below)	Str
NULL	NULL	NIL
OBJECT IDENTIFIER	NULL	Str
ObjectDescription	NULL	Str
EXTERNAL	n/a	n/a
REAL	NULL or *subtype* (cf. below)	Double
ENUMERATED	`MetaObj`. The dictionary contains as values the strings (`MetaString`) that name the elements of the enumeration	Enum
SEQUENCE, SET	`MetaObj`. The dictionary contains as key the names of the members and as values instances of subclasses of `MetaElement` which describe the types or the members	Struct
SEQUENCE-OF, SET-OF	`MetaObj`. The dictionary contains 1 or 2 properties: `"type"` and `"size"` which point to instances of `MetaLong`. The first describes the type of the elements of the sequence, the second describes the length of the sequence (if present)	Sequence
NumericString	NULL or *subtype* (cf. below)	Str
PrintableString	NULL or *subtype* (cf. below)	Str
TeletexString	NULL or *subtype* (cf. below)	Str
VideotexString	NULL or *subtype* (cf. below)	Str
IA5String	NULL or *subtype* (cf. below)	Str
UTCTime	NULL	Str
GeneralizedTime	NULL	Str
GraphicString	NULL or *subtype* (cf. below)	Str
VisibleString	NULL or *subtype* (cf. below)	Str
GeneralString	NULL or *subtype* (cf. below)	Str
CharacterString	NULL or *subtype* (cf. below)	Str
CHOICE	`MetaObj`. The keys of the dictionary are the names of the union's members. Each name points to another instance of `MetaObj` which is an ASN.1 type	Union
ANY	NULL	n/a
ANY DEFINED BY	n/a	n/a
SELECTION	n/a	n/a
TAGGED	n/a	n/a
recursive	n/a	n/a

Table 4.5: Mapping of meta model types using the GDMO layout

Property	**Value**
"single_value"	ASN.1 value (composed of `MetaElements`)
"contained_subtype"	ASN.1 type
"value_range"	List (`MetaList`) with start- and end values (ASN.1 values)
"permitted_alphabet"	ASN.1 type
"size_constraint"	List (`MetaList`) with start- and end values (ASN.1 values)
"inner_type_constraints"	n/a

Table 4.6: Mapping of ASN.1 subtypes

4.2.3.4 Summary

A major point of the generic meta model is to accommodate metadata of several different object models in the same repository without modification of the meta model's structure. This is the main reason for its flexible structure.
Any type of model can be represented using the generic meta model, and there should be no loss of metadata information caused by a requirement to convert specific metadata to a (potentially insufficiently flexible) fixed metadata model.
As the one element of the meta model that can contain other elements, `MetaObj`, represents aggregation of or references to other elements through a dictionary (set of properties) in which the keys (strings) denote the property names and the values point to other meta model elements, *any* property can be represented by constructing recursive instances of `MetaObj`, thus forming a *tree*, with the leaves being instances of either `MetaLong`, `MetaFloat`, `MetaString` or `MetaList`.
This scheme allows to flexibly model any type of information needed to represent a specific meta model through the generic one. There is no need to extend the structure of the meta model by subclassing its elements (class extension) as proposed in section 4.3.1, but all elements of any meta model can be represented by creating a corresponding meta model structure using elements of the generic meta model (i.e. subclasses of `MetaElement`).
Since the structure of the generic meta model is neutral with respect to any type of specific meta model, it can be used to represent any specific metadata. How this is done is specified in layout definitions that define which elements are available (names and semantics), their relations to other elements (containment, aggregation), the type code constants for each type, and how elements and types are mapped to corresponding instances in the instance model.
Layout definitions are used by clients of the generic object model to determine which types of values have to be used in the instance model (e.g. for method calls).
Whereas the structure of the meta model is generic and can be specialized to suit metadata from any meta model, the structure of the instance model is fixed and cannot be changed, i.e., it should be used as is to represent other object models without having to be extended. This has the benefit that clients have to manipulate only one model and do not need to know about mappings from other models. The only extension possibility in the instance model is the `properties` attribute in the `GenObj` interface, which can be used to attach additional information to an object, e.g. the distinguished name of a GDMO managed

object (cf. 4.2.2).
The generic meta model lends itself to the creation of generic tools (e.g. metadata browsers, documentation tools) that traverse its structure and display it to users since that structure is always the same. The semantics of the meta model, however, changes from layout to layout imposed on the generic meta model and the only place where it is defined is in the layout definition itself. Therefore, although the structure of metadata can be displayed to a user, the semantics has to be grasped from the layout definition. One place where the semantics of a layout is needed is in an adapter where metadata is used for type-checking and conversion purposes. As it is anticipated that adapters are written by the same persons who also furnish the metadata adapters and write the layout, this should not pose any problems.

The goals of the metadata model listed at the beginning of this section were simplicity, ease of use, integration with GOM and extensibility.
Extensibility has been achieved by defining a very generic meta model on top of which layout definitions can be placed to suit any type of meta model. Layout definitions for CORBA and GDMO have been provided that demonstrate the usefulness of the generic meta model with respect to extensibility.
The generic meta model is very simple, it consists of only 5 classes which can be used to represent any specific type of meta model. By virtue of its simplicity, the model also achieves ease of use: clients do not have to learn a large number of classes as in the case of CORBA's Interface Repository, but only have to know the 5 classes mentioned above as well as the layout definition for a specific metadata model.
Memory management problems should not occur in the generic meta model since all data is owned by the metadata repository and will be deleted by it only when no longer used (cf. 4.3.2).
The latter point, namely integration with GOM's instance model, has also been achieved: the instance- and meta-models are similar. The result is that clients of GOM have to learn only one type of model, whereas in the alternative solution that will be proposed in section 4.3.1, the instance model (GOM) would be different from the meta model (CORBA's IR).

4.2.4 Convenience Bindings

All elements of GOM are defined in OMG IDL. As IDL interfaces cannot directly be used by client applications, they have to be translated to a desired *implementation* language (cf. 1.3.6) (e.g. C++, Smalltalk, Java etc.) using an IDL compiler which generates the language binding according to a standard mapping defined by the OMG (see 2.1.3).
Since IDL is required to support a number of implementation languages, it cannot make use of features available in only one language, but has to be based on features available in *all* of the supported implementation languages.
It is therefore possible that when mapping IDL to a specific language, the generated binding will not make use of language-specific features, resulting in a poor API compared to the richness of language features available.
A solution to this problem is to add *convenience bindings* for the generated language bindings as shown in fig. 4.22. They constitute an additional layer on top of the bindings for a particular implementation language, providing a more elegant interface to clients

which conforms to the overall look and feel of the host language and makes use of the latter's features.[18]

The following sections will present sample convenience bindings for C++ as an example of a strongly-typed and Smalltalk as an example of a weakly-typed language. Similar bindings can be provided for other languages.

IDL interface `GenObj` (fig. 4.7 on page 47) will be used as an example. The corresponding convenience class will be called `GomObj` in the examples. Convenience bindings for other IDL interfaces (e.g. the other interfaces of the instance model or the interfaces of the meta model) are not discussed here for space reasons. However, they will be similar to the ones proposed here.

The convenience bindings presented below are merely *examples* of how GOM can be adapted to different languages. Of course, any client application can provide its own convenience bindings if needed, e.g. in the case where the provided convenience bindings are insufficient, or if there are none available for a given language.

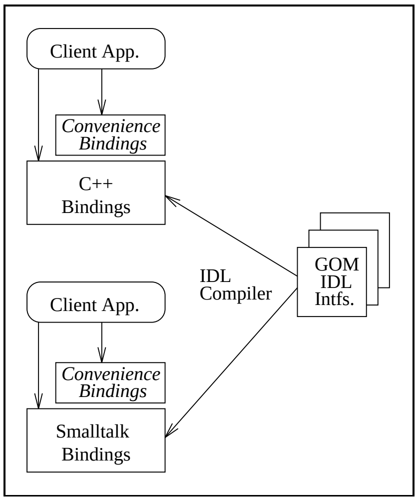

Figure 4.22: Convenience bindings

Since the set of IDL interfaces in GOM's instance- and meta model is finite, translation of the IDL interface code comprising GOM should be performed only once for each language binding. Therefore convenience bindings have a solid basis that is not expected to be modified.

Both examples below will show how to access the CORBA `Printer` interface as defined in fig. 4.23.

```
interface Printer {
    attribute string location;
    boolean Print(in Document doc, in long number_of_copies);
};
```

Figure 4.23: IDL interface `Printer`

4.2.4.1 C++

Generated Language Bindings When translating IDL interface `GenObj` to C++ using an IDL compiler, the generated C++ binding[19] is very similar to the IDL definition of `GenObj`. All IDL operations are mapped to methods in C++, and attribute access and retrieval is mapped to *accessor methods*. To get an attribute `T`, an accessor method `T()` is defined and to set it, the accessor method has the signature `T(const T_ptr)`.

[18] Offering different interfaces to existing classes employs the *adapter* design pattern [GHJV95].
[19] According to the OMG's C++ language binding as defined in [OMG95, ch. 16].

```
class GenObj : public virtual Object {
  public:
    Adapter_ptr    adapter();
    void           adapter(Adapter_ptr);
    char*          classname();
    void           classname(const char*);
    char*          instance_name();
    void           instance_name(const char*);
    Dictionary_ptr properties();
    void           properties(Dictionary_ptr);

    Val_ptr        Get(const char* attrname);
    void           Set(const char* attrname, Val_ptr new_val);
    Val_ptr        Execute(const char* opname, Arglist_ptr args);
    void           Delete();

    MetaObj_ptr    GetClassDef();
    MetaObj_ptr    GetAttributeDef(const char* attrname);
    MetaObj_ptr    GetOperationDef(const char* opname);
    MetaObj_ptr    GetElementDef(const char* element_name);

    Val_ptr        GetProperty(const char* name);
    void           SetProperty(const char* name, Val_ptr new_val);
};
```

Figure 4.24: Generated C++ binding for interface `GenObj` (shortened)

Use of Generated Bindings An example of using the generated class for the IDL interface specified in fig. 4.23 is given in fig. 4.25.

```
GenObj_ptr   printer=::name_server.Find("psp30d");
Factory_ptr  factory=new Factory;
Long_ptr     num=new Long;
Arglist_ptr  args=new Arglist;
Bool_ptr     retval;
Str_ptr      pw=new Str;
GenObj_ptr   doc=factory->Create("CORBA", "Defs::Document",
                                     0, 0, "adlerhorn.zurich.ibm.com", 0);
pw->val("/etc/passwd");
doc->Set("name", pw);
num->val(2);
args->Add("doc", doc);
args->Add("number_of_copies", num);
try {
    retval=(Bool_ptr)printer->Execute("Print", args);
    cout << retval->val() << endl;
}
catch(GenEx& ex) { /* error */ }
```

Figure 4.25: Use of generated C++ class

The example creates an instance of IDL interface `Defs::Document` and prints it on a printer, the reference to which is retrieved from a naming service. Note that the arguments to the `"Print"` method have to be created and added to an instance of `Arglist` which is one of the required arguments to `GenObj::Execute`.

Convenience Bindings A possible C++ convenience binding for `GenObj` is shown in fig. 4.26. It augments the generated C++ class `GenObj` (fig. 4.24).

```
class GomVal : public Val {};

class GomObj : public GenObj {
  public:
    GomVal*&      operator[](char* attrname);
    GomVal*       Execute(const char* opname,
                          const char* argname, GomVal* arg1 ...);
};
```

Figure 4.26: Convenience binding for generated C++ class `GenObj`

The C++ convenience binding inherits from `GenObj` which is different from the Smalltalk convenience binding where only a reference to an instance of `GenObj` is stored (see below).

Thus, all member attributes and method of the superclass are available in the convenience class `GomObj`.
The three main contributions of this binding are a safe narrowing mechanism for values, constructors and operators for C++ classes and variable-length argument lists.
Whenever a generic value `Val` is returned from a method call, it has to be narrowed to its actual class (e.g. a `Long`). For this purpose, a corresponding convenience value class with the prefix `Gom` is created for each `Val` class, e.g. `GomLong` for `Long`. Each new class has a static method `Downcast` that accepts as parameter a value instance (`Gom<X>`) and returns the downcast value if the downcast is permissible or NULL if it is not. An example will be shown below.[20]
The rich constructor and operator overloading mechanisms of C++ can be employed to facilitate the use of the generated C++ classes considerably. For instance, every subclass of `GomVal` has a constructor that allows to create an instance with a native C++ value or with another generic value. Also, conversion operators between native C++ values and generic values can be provided.
Rather than having to create and populate an instance of `Arglist` for operation `Execute`, it is possible to provide variable argument lists as shown in fig. 4.26. This allows to provide arguments to an operation invocation right away, without first having to construct an argument list object and adding the arguments to it.
An example of how overloaded operators can be used to facilitate the use of the generated C++ classes is to replace the `Get`- and `Set` methods by the subscripting operator [Str91, par. 13.4.5]. Thus the two operations shown below:

```
Long_ptr age=(Long_ptr)person->Get("age");
Long_ptr new_age=new Long;
new_age->val(32);
person->Set("age", new_val);
```

could be replaced by:

```
GomLong_ptr age=GomLong::Downcast((*person)["age"]);
(*person)["age"]=new GomLong(32);
```

Note that methods `Get` and `Set` may still be used since convenience class `GomObj` is derived from `GenObj`.

Use of Convenience Bindings An example of how the C++ convenience bindings can be used is given in fig. 4.27
The code in fig. 4.27 is essentially the same as in 4.25, but it uses the convenience bindings for C++ which makes the code shorter.

[20] Note that, if a C++ implementation supports runtime type identification (RTTI), then a safe downcast can be performed using the `dynamic_cast` operator [Str91].

```
GomObj_ptr      printer=::name_server.Find("psp30d");
Factory_ptr     factory=new Factory;
GomLong_ptr     num=new GomLong(2);    // use of constructor
GomBool_ptr     retval;
GomObj_ptr      doc=factory->Create("CORBA", "Defs::Document",
                                    0, 0, "adlerhorn.zurich.ibm.com",
                                    "name", new GomStr("/etc/passwd"),
                                    0);
try {
    retval=GomBool::Downcast(
             printer->Execute("Print",
                              "doc", doc,
                              "number_of_copies", new GomLong(2),
                              0));
    cout << *retval << endl;  // conversion operator
}
catch(GenEx& ex) { /* error */ }
```

Figure 4.27: Use of C++ convenience binding

Evaluation Use of the C++ convenience bindings has a number of advantages. First, constructors are defined that allow to create an instance and initialize it directly in one step rather than two. Second, overloaded C++ operators are used to make certain explicit operations implicit, e.g. automatic conversion of the return value to a boolean value when printed, as in example 4.27. Third, if RTTI is not available, a safe downcasting mechanism is provided. Finally, variable argument lists allow to reduce certain steps when invoking an operation.

4.2.4.2 Smalltalk

Generated Language Bindings According to OMG's Smalltalk mapping [OMG95, ch. 19–21], a corresponding Smalltalk class `GenObj` is generated for the IDL interface `GenObj` as shown in fig. 4.28.

Use of Generated Bindings An example of using the generated class for the IDL interface specified in fig. 4.23 is given in fig. 4.29.
First a previously registered instance (`"psp30d"`) of `Printer` is retrieved from a naming service. Then a document that will be printed is created. Target object model is CORBA, and the name of the document class is `"Defs::Document"`. The target instance will be created in a CORBA server on host `adlerhorn`, and its accompanying proxy instance will be created locally (in the client's address space). Attribute `"name"` of the target instance is set to `"/etc/passwd"` using the `set` operation of the proxy.
Then, operation `Print` of the target object is invoked using as argument list the previously created document and an instance of `Long` which was also created previously and set to

```
class:      GenObj
superclass: Object
instance variables: | adapter classname instanceName properties |
instance methods:
    get:              attrName
    set:              attrName   newVal: val
    execute:          opname     args: arglist
    delete

    getClassDef
    getAttributeDef: attrName
    getOperationDef: opname
    getElementDef:   elementName

    getProperty:     name
    setProperty:     name newVal: val
```

Figure 4.28: Generated Smalltalk binding for interface `GenObj` (shortened)

value 2.

Convenience Bindings One of the advantages of the reified object model of GOM is that message names (*selectors* in Smalltalk) do not have to be determined at compile time, but can be constructed at runtime. While the only possibility in strongly typed languages such as C++ to construct messages at runtime is to use strings for the message names and to provide a special interface that uses metadata (`Get`, `Set` and `Call` operations in `GenObj`), weakly typed languages such as Smalltalk have the ability to construct messages at runtime and send them to an object.
The Smalltalk convenience binding for `GenObj` (`GomObj`) makes use of this feature and works as follows. Every `GomObj` instance has a reference to an instance of `GenObj`. When a message is sent to an instance of `GomObj` and the message is not known by the instance's class, method `doesNotUnderstand` will be invoked by the Smalltalk runtime. The standard implementation of `doesNotUnderstand` would produce an error and jump into the debugger. If, however, this method is overridden in the convenience binding's class (`GomObj`), then we could construct a corresponding message and forward it to the `GenObj` instance.[21]
The implementation of message `doesNotUnderstand` is shown in fig. 4.30.
It receives as argument an instance of `Message` which represents a generic Smalltalk message with the name of the selector, its arguments etc.
First, the instance of `GenObj` is used to determine whether a variable should be set or retrieved or an operation performed. If the target class that the instance of `GenObj` represents does not have a variable with the same name as the message selector, it is

[21] This mechanism is known as *Proxy* pattern and is described in [GHJV95, P. 215].

```
| printer retVal doc num args factory |
printer := NameServer at: 'psp30d'.
factory := Factory new.

doc := factory create: 'CORBA' className: 'Defs::Document'
               instName: nil proxyLocation: nil
               targetLocation: 'adlerhorn.zurich.ibm.com'
               args: nil.
doc set: 'name' newVal: (Str new val: '/etc/passwd').
num := (LongVal new) val: 2.
args := ArgList new.
args add: 'doc' v: doc.
args add: 'number_of_copies' v: num.

retVal := printer execute: 'Print' args: args.
retVal printNl.
```

Figure 4.29: Use of generated Smalltalk class

assumed that an operation has to be invoked. In this case, an instance of `ArgList` is created and populated by iterating through all arguments of the message and, for each argument, converting it to an instance of a subclass of `Val` which is then added to the argument list. Finally the message is forwarded to the instance of `GenObj` that is the value of `realObject`.

In the case of a variable, it has to be determined whether the variable is to be set or retrieved. This is determined by the argument list: if it is empty, a `Get` message is sent to `realObject`, otherwise a `Set` message is sent.

Two sorts of conversions have to be performed by the convenience binding: Smalltalk values have to be converted to GOM values (subclasses of `Val`) and GOM values have to be converted to Smalltalk values.

In the first case, each Smalltalk value can be converted to a corresponding GOM value by looking at the class of the Smalltalk value.

In the second case, since each GOM value has a tag[22] that determines its type, it can be converted to the corresponding Smalltalk value as well.

Use of Convenience Bindings Fig. 4.31 shows how the code shown in fig. 4.29 would look using the Smalltalk convenience binding for `GenObj`.

The amount of code to be written using the convenience bindings is considerable less than if using the generated Smalltalk bindings. Additionally, it fits Smalltalk's philosophy more adequately in that 'helper operations' such as `get`, `set` and `execute` are converted into message selectors, e.g. `obj execute: 'operationname'` becomes `obj operationname`.

[22] Retrieved using operation `GetKind`, cf. fig. 4.4 on p. 41.

```
| msgName numberOfArgs args arg |
doesNotUnderstand: aMessage
    args         := ArgList new.
    msgName      := aMessage selector.
    numberOfArgs := aMessage arguments size.
    (realObject getAttributeDef: msgName notNil) "is msgName an attr. ?"
      ifTrue: [(numberOfArgs > 0)
               ifTrue: [arg := (arguments at: 0) convertToGOM.
                        ^ self realObject
                        set: msgName newVal: arg] "arg is converted val"
               ifFalse: [^ self realObject get: msgName]
              ]
      ifFalse:
        "process arguments and add to 'args'"
        [^ self realObject execute: msgName args: args]
```

Figure 4.30: Message `doesNotUnderstand` of class `GomObj`

Evaluation Using the Smalltalk convenience bindings, none of the operations `get`, `set` or `execute` have to be used. Instead, the attribute names can be used directly for attribute access/modification and the operation names for operation invocation, which results in tighter code and conforms to the Smalltalk philosophy.

4.2.4.3 Summary

Convenience bindings allow to adapt an implementation language binding generated from IDL to the look and feel of the host language at hand. They can be seen as an (optional) add-on to be employed for more ease of use of the instance model. Clients are free to write their own convenience bindings for a language if the provided ones are not sufficient or if there are none available.
Compared to the C++ convenience bindings, the Smalltalk bindings seem to be of greater use to clients since they eliminate the need to use operations `Get`, `Set` and `Execute` and instead offer an interface to clients that conforms to the Smalltalk philosophy.
The C++ convenience bindings provide only small additional ease of use on top of the generated bindings, with minor cosmetic changes, but no significant semantic modification.

4.2.5 Typing

Unlike the static approaches, which are *statically typed*, GOM is *dynamically typed* as shown in fig. 4.32.
Object-oriented languages can either be *untyped* or *strongly typed* [Weg90]. Untyped languages have no explicit type system and no type-checking is performed upon assignment between values. Examples of untyped languages are Lisp [Ste90a] and Smalltalk [GR89].

```
| printer retVal doc num args factory |
printer := NameServer at: 'psp30d'.
doc := factory create: 'CORBA' className: 'Defs::Document'
               instName: nil proxyLocation: nil
               targetLocation: 'adlerhorn.zurich.ibm.com'
               args: nil.

doc name: '/etc/passwd'.

retVal := printer Print: doc numberOfCopies: 2.
retVal printNl.
```

Figure 4.31: Use of convenience binding for Smalltalk

Strongly typed languages enforce type-checking, thus avoiding assignment of values of incompatible types. Type consistency is checked either at compile time or at runtime. Languages in which the type of expressions can be determined at compile time are called *statically typed*, while languages in which types are determined at runtime are called *dynamically typed* [Weg90].

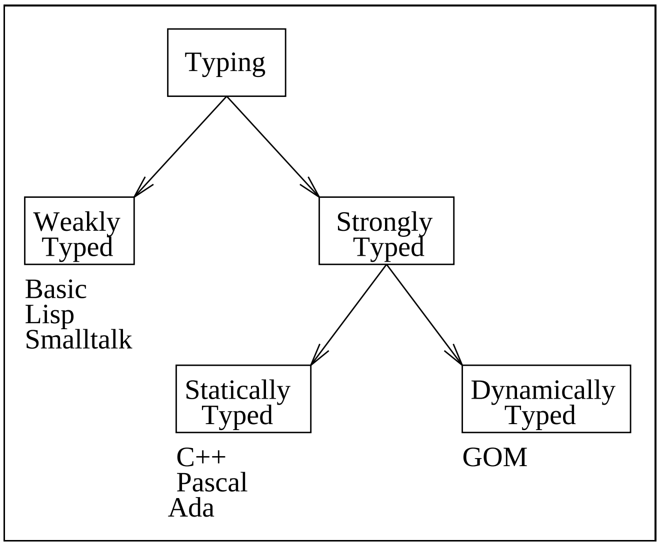

Figure 4.32: Typing classification of GOM

Both the XoJIDM (3.1.2) and GOM approaches are strongly-typed. But contrary to XoJIDM's approach, which maps GDMO / ASN.1 to C++ code that can be type-checked at *compile time*, GOM's model uses a type repository to enforce type-checking at *runtime*. This approach combines the advantages of untyped languages such as Smalltalk (flexibility) with the ones of typed languages such as C++ (safety). This is possible since, unlike Smalltalk which is an untyped implementation language, GOM handles target models most of which are specified using typed interface specification languages such as IDL or GDMO which are parsed and fed into a type repository which can subsequently be used for runtime type- checking. Thus, the GOM model is essentially strongly typed with type-checking enforced at runtime.[23]

4.2.6 Summary

In this section, GOM's generic object model and its main components, the instance- and meta models were discussed. It was shown that their genericity helps avoid having to

[23]Main use of runtime type- checking is in adapters which have to convert values between GOM and a target model, thereby enforcing type compatibility.

extend them to represent a number of different (object- and meta-) models. The instance model allows to manipulate a number of target instances whereas the meta model can be used to obtain metadata about the classes of a target system.
Convenience bindings contribute to the integration of the proposed model with a number of host languages. The closer a language adheres to the concept of object-oriented reification (everything is an object), the better GOM can be integrated using convenience bindings.

4.3 Metadata Repository

The concept of metadata is essential for GOM. As type information about the classes of the target systems is not compiled into GOM, information needed to handle these classes must be available *at runtime*. Access to metadata is needed by *adapters* to perform type-checking and conversion work (cf. 4.4) and can also be used by *client applications* that need it, for example, to implement class browsers, persistence services, documentation, interpreters and so on.
As the target systems to be managed by GOM include a number of different models, their metadata is necessarily heterogeneous, and this fact must be accounted for by the metadata repository. The latter's architecture must therefore accommodate a number of possibly widely differing object models while giving the impression of homogeneity to its clients. It is a purpose of the metadata repository to integrate all sorts of specific metadata into a *common model* (described in section 4.2.3).
Specific metadata should be converted into the common model as far as possible. Some idiosyncrasies, however, will simply *not* fit the common model and therefore the latter has to be extended using one of two mechanisms shown in the next two sections.
Two alternative designs for a metadata repository will be presented. Then an evaluation of the two alternatives will be made and the reasons for selecting the second alternative will be given. Finally, an overview of similar approaches to metadata repositories in the management domain will be given and compared with the proposed solution.

4.3.1 Extending CORBA's Interface Repository

The approach described in this section is based on CORBA's *Interface Repository (IR)* which is proposed as starting point for the common meta model. *Besides* accommodating metadata for CORBA, the *extended interface repository* (EIR) also provides metadata about other models such as CMIP (GDMO/ASN.1). The architecture of the EIR is shown in fig. 4.33.
For each target model a separate database is created in which all metadata about that model is kept. The databases are hidden from clients of the metadata repository and metadata can be retrieved or added only through the CORBA interface repository *read-* and *write*-access API [OMG95, ch. 6]. This ensures that the CORBA IR API need not be modified, but only its *implementation* needs to be changed.
A client of the metadata repository of course has to know from which model metadata is to be retrieved. Therefore, the name of the database (e.g. "X700") has to be given when

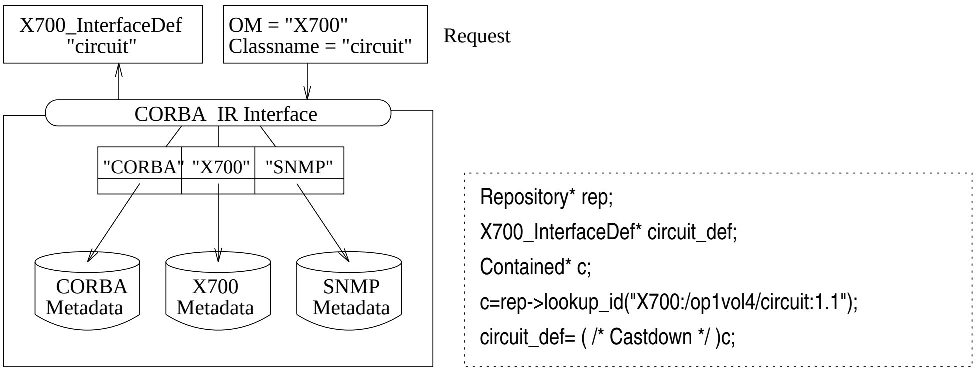

Figure 4.33: An extended interface repository

querying metadata about X700. As shown in fig. 4.33, this can be achieved by prefixing a CORBA *RepositoryId* string (see [OMG95, section 6.6.4]) with the name of the object model (and thus of the database) to be used for the metadata lookup. In the example the operation `lookup_id` is used with the RepositoryId `"X700:op1vol4/circuit:1.1"`. This causes the `X700` database to be searched for a GDMO template named `circuit` in a module named `op1vol4` (with version 1.1). Information about that template is then returned in the form of an `InterfaceDef` which has to be narrowed to an instance of `X700_InterfaceDef` to get to the subclass-specific information.

Metadata of target models should be described - whenever possible - using the existing IDL interfaces of CORBA's Interface Repository. If this is not possible, e.g. in the case of *GDMO packages* in a GDMO class template (cf. [GDM92]), the corresponding IR element (e.g. an `InterfaceDef`) must be subclassed by the metadata provider to accommodate the additional information, e.g. in an IDL interface `X700_InterfaceDef`.

Another example is the case of the IDL interface `AttributeDef` which is an element of the Interface Repository that provides metadata about an attribute of a CORBA interface. Since an attribute in a GDMO template needs to record more information than its CORBA counterpart (such as the package to which it belongs and initialization and access constraints such as initial, default, permitted and required values [GDM92, section 8.4.3.2]), `AttributeDef` cannot be used. Therefore, a subclass of `AttributeDef` needs to be created (`X700_AttributeDef`) that provides the additional information. This is demonstrated in fig. 4.34.

As shown, metadata about a GDMO attribute is kept in an instance of a subclass of IR's `AttributeDef` interface. The information in the `AttributeDef` part of the resulting instance contains the normal information such as the name of the attribute, its type and the access restrictions (such as for read-only attributes). The subclass `X700_AttributeDef` contains additional information that is not covered by the `AttributeDef` interface such as the OID for the attribute, initial and default values etc.

When retrieving metadata, a client of the EIR potentially has to narrow the returned instance to its actual class to be able to access the additional information. As an example, a query for an attribute of a GDMO template may return an instance of type `AttributeDef`, but its actual class may be `X700_AttributeDef` so the returned object needs to be narrowed to `X700_AttributeDef` to access the additional information about

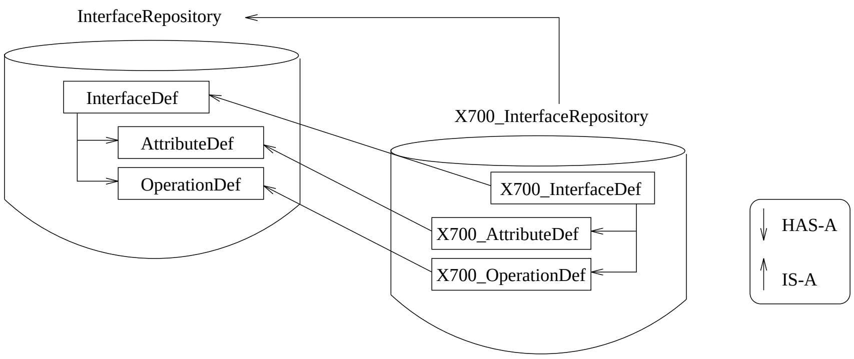

Figure 4.34: Integration of specific metadata

the GDMO attribute.
In order to integrate a new target model with the metadata repository, the following steps have to be taken:

1. A new database with the name of the target model must be created in the EIR.

2. A parser must be written that populates this database by parsing specifications of the new object model and generating input to the EIR as result. The decision whether to reuse existing elements of CORBA's IR to represent metadata of the new model or to create new metadata interfaces is the decision of the parser's author. Usually, one would reuse existing elements as much as possible, with a few exceptions where subclasses are needed.

3. An adapter (cf. 4.4) that converts between the generic object model and the new model has to be written. This will usually be performed by the same person who writes the parser and who will therefore know the metadata (e.g. which classes are subclassed and which ones are reused).

The scheme described above has the advantage that it reuses existing concepts from CORBA, actually enhancing CORBA's interface repository towards a multi-domain (object model) metadata repository *without modifying* its interface (API). Such a type repository could potentially play an important role in future heterogeneous networks as a common registry for metadata which can be used for several different object models.

4.3.2 Providing Own Metadata

Another approach to providing metadata of multiple object models is shown in fig. 4.35. With this approach, each generic interface `GenObj` has a corresponding generic metaclass `MetaObj` describing the *target* object[24] in the *metadata repository* (MR). The metadata repository is represented by an instance of `MetadataRepository` (cf. A.4).

[24]For example, if the `GenObj` proxy instance represents an OSI managed object (MO), then the corresponding `MetaObj` would describe the MO's GDMO template.

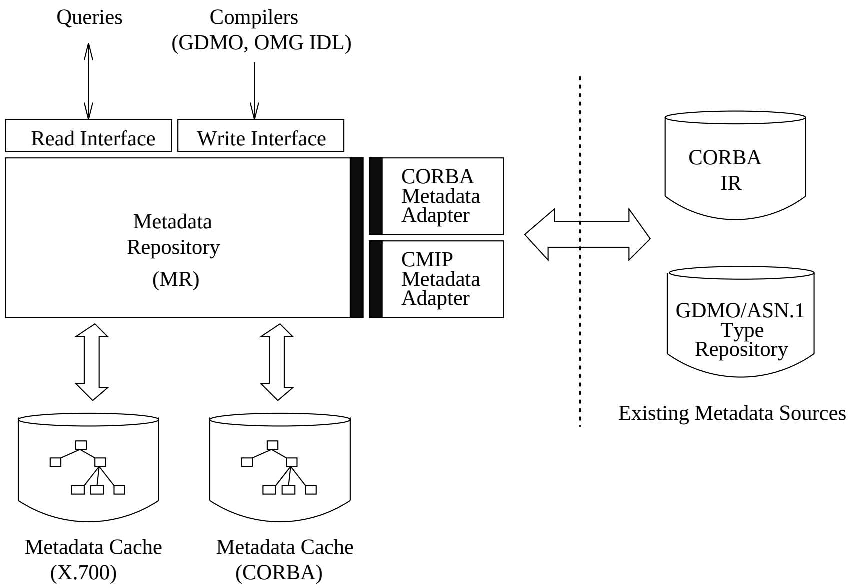

Figure 4.35: Providing own metadata

All metadata is located in a number of *metadata caches* in the client's address space. A metadata cache is a database for one specific meta model, e.g. for CORBA. There are separate caches, one for each object model.
There are two ways to populate a metadata cache: the first is to use a compiler which parses the specifications of a target object model, generates input to the metadata cache and uses the metadata repository's *write* interface to enter the data.
The second is to use *existing* metadata sources of a target model and to employ *metadata adapters* as bridges to these sources. A metadata adapter is responsible for furnishing metadata about a target object model to the metadata repository in the *generic metadata format* specified in section 4.2.3. It does so by accessing (in most cases) available metadata sources and converting and *copying* the information so that the metadata repository can add this data to one of its caches. A source may for example be CORBA's Interface Repository. A metadata adapter acts as a 'just-in-time' metadata provider; it is contacted whenever metadata cannot be found in the cache. When a metadata adapter returns metadata, the latter will be added to the cache so that the next request can be served directly from the cache.
These two alternatives allow both the integration of existing sources without first having to parse and enter them into the common metadata model *and* the (offline) parsing of specifications by a compiler to add them to the metadata cache, e.g. for performance reasons or if no existing metadata sources are available.
For a given object model X there can be either a metadata cache *and* -adapter or *only* a metadata cache available.
In the first case, the data is fetched from a specific source and added to the cache for efficiency purposes, populating it over time. Its contents will be deleted when the application is terminated, or can optionally be saved to a file so that the cache will be immediately usable when the application is started the next time. The latter scheme will reduce the need for metadata adapters over time as more and more data is available directly from

the cache. Actually, in the end this approach achieves the same result as employing a compiler to parse specifications and input them to the cache (without having to write a compiler back-end or parser). Of course, when the cache is flushed, metadata adapters will still have to be contacted to provide metadata.

In the second case, no existing metadata sources may be available for a certain object model. In that case, it may be feasible to write a compiler (or reuse an existing one) to parse the specifications and add them to the cache. Thus, the cache essentially acts as a database for metadata of a certain model.

When metadata is needed by a client, the metadata repository is accessed given the object model and the name of the element (e.g. a class name) to be looked up. An example is `FindClass("CORBA:CustomerApplication/customer:2.1")` which means that metadata about an IDL interface in the `CORBA` object model named `customer` in module `CustomerApplication` with version 2.1 is queried.

The metadata repository will alway first try to find a cache for a certain object model upon receiving a request. If found, it is used to search for the desired information. If the cache is not found or the information cannot be located, then a corresponding metadata adapter is sought. If found, the request will be forwarded to it and the result added to the cache before returning it to the client. If not found, or if the information cannot be located, then an error message will be returned to the client application.

To minimize network traffic, whenever a module or class definition is requested, the *entire* module or class definition will be shipped to the metadata repository by the metadata adapter and copied to the cache. This scheme avoids having to keep track of the pieces that have already been transferred. In addition, using TCP/IP it is more efficient to transfer one large packet rather than several small ones [Tan92]. Once the metadata is in the cache, it can be accessed by clients for queries.

The metadata cache can be flushed at any time to accommodate the fact that metadata from a specific source may have been modified in the meantime. Flushing the cache forces its subsequent repopulation which has the effect that modified metadata will immediately be usable. Note that if the cache is used as a database then the contents of it will be deleted as well.

The metadata repository provides a number of operations to populate the cache manually, i.e. operations that fetch all information that is available from meta adapters and copy it to the cache. This is useful for initial metadata discovery, e.g. to list all the classes available in a target object model. It can also be used to populate a cache in the background, e.g. by invoking these operations in a separate initialization thread which starts populating the cache at startup. Note that these operations do not make sense if there is no corresponding metadata adapter available. In this case, they simply return the requested information, or – if it cannot be found – an error.

Note that clients will always deal with metadata elements that are owned and maintained by the cache so they must not modify retrieved metadata. By giving clients direct memory pointers to data in the cache, memory ownership problems can be avoided: everything is owned by the cache and will be deleted by it when flushed or when the cache is terminated.[25]

[25] The C++ language is assumed here as language binding. Of course, this problem does not occur in languages with automatic garbage collection.

Whenever a new object model is to be integrated into the generic object model, a new metadata adapter has to be provided that is able to furnish information about the model to the metadata repository, or a compiler (back-end) has to be written that parses specifications for a given model and generates metadata to be input to one of the metadata caches. The advantage of this scheme is that existing metadata repositories can be 'tapped' and their information reused without the need to generate essentially duplicate metadata in a separate EIR (as proposed in the previous section).
As all elements of the meta model can be marshaled and unmarshaled (using methods `Dump` and `Read`, shown in fig. 4.4), the contents of the cache can be saved across invocations.[26] This is important in cases where a large amount of metadata is used and initial cache population from target sources through the meta adapters takes too much time. See also section 4.6.4.

Since the metadata repository is itself a CORBA interface, it can be located anywhere in the network. Retrieval of an object reference to it will typically be done by the CORBA *naming service* [COS95]. Whenever a new instance of the MR is created, it will be registered under a certain name with the naming service. Its object reference can subsequently be retrieved by any client that knows its registration name.
There may be a number of MRs in a network that clients can access. However, for performance reasons, clients will typically try to access the MR closest to them.
Process group mechanisms [Maf95, Bir96] could be used to increase availability and performance of the MR service. Instead of creating one central instance of an MR, a number of instances would be created in several locations, forming a group. An operation invocation would be sent to all members of the group and the first response received would be used. Thus temporary non-availability of an MR instance would not stop the service from working as long as there is at least one member left in the group. Also, since the first response is used, performance of the overall service would increase.

4.3.3 Related Work

4.3.3.1 SMK

Work being done on implementing OSI's Shared Management Knowledge (SMK) management function [X7593, TMN95] is described in [HPSK96, PHHS96]. The SMK function is implemented in CORBA and consists of translating the SMK definitions (GDMO/ASN.1) to CORBA IDL following XoJIDM's specification translation proposal [Spe97] and implementing the resulting IDL interfaces in an OSI agent. Management knowledge consists of the following elements:

- Instance and instance relationship (*Instance Model*). This function allows to retrieve information about an agent's containment tree, it enables e.g. OSI managers to retrieve the root instance(s) of an agent for traversal of the agent's MIB.
- Managed object class knowledge and name bindings (*Meta Model*). Knowledge of GDMO templates, packages, actions etc. and of ASN.1 types can be retrieved using this function. This is similar to the meta model of GOM.

[26] Actually, these two operations are the main mechanism for making metadata caches persistent.

- Protocol and association establishment knowledge and systems management functions (SMFs) available at the agent.

Unlike the proposed approach, it is not just metadata that is stored in the SMK, but also information about the instance model is available, i.e. the containment tree of an OSI agent. Moreover, information needed by a manager to establish an association to an agent is available which enables OSI managers to bind *dynamically* to agents using the information returned by the SMK function (such as the agent's presentation address, application entity name (AE-title) and protocol [ITU92a, ITU92b]).
The SMK function deals specifically with metadata about OSI agents' MIB's and is not designed to integrate other types of metadata, as this is not its purpose. It would be impossible or at best awkward to add metadata about CORBA interfaces to the SMK function. The meta model described in this thesis, however, is specifically designed to take into account all different types of metadata.
Using the SMK function, an OSI manager typically has to deal with two types of APIs: one for the exchange of shared management knowledge (CORBA) and another one for the manipulation of instance data in an agent (CMIP). The interfaces offered for the instance- and meta-model are therefore – unlike GOM's approach – not based on the same object model.

4.3.3.2 Management Interface Repository

The work described in [Hie96a] is part of a proposal for XoJIDM's Interaction Translation (cf. section 3.1.2). The *Management Interface Repository (MIR)* is a meta database for the purpose of recording information that was lost when translating GDMO/ASN.1 to IDL or IDL to GDMO/ASN.1, such as the object identifier (OID) of the GDMO template from which the corresponding OMG IDL interface was generated.
It is for example possible to determine the ACTION template definition from which a given IDL operation was derived or to retrieve the OMG IDL attribute definition to which a given GDMO attribute definition was mapped.
This is achieved by MIR's object model which essentially comprises two hierarchies of metadata; one consists of elements that describe the OMG IDL definitions and which are derived from CORBA IR interfaces (similar to 4.3.1), e.g. a `MIR::InterfaceDef` is derived from `CORBA::InterfaceDef`. These elements have as additional members links to elements of the second hierarchy, the management-specific metadata which essentially describes GDMO/ASN.1 information from which the OMG IDL definitions were generated. The two hierarchies are related through bidirectional links which allows to associate source- and target definitions, e.g. to retrieve the GDMO definition of an element for which OMG IDL code was generated.
Information in the MIR is used by the *gateway* that is responsible for converting CORBA to CMIP (CORBA manager of an OSI agent) and CMIP to CORBA (OSI manager of a CORBA agent) requests. Also, the MIR API can be used by applications that need access to metadata.
The main purpose of the MIR is to record the mapping between GDMO/ASN.1 and generated IDL (and vice versa) which is lost during translation. It proposes that CORBA's

interface repository be integrated by making the MIR a superset of metadata which includes the CORBA IR *and* additional management-specific information.
This concept is similar to what has been described as an alternative to metadata for this work in section 4.3.1 (EIR). A difference is that, while applications that require access to management-specific information must use the MIR API, they could continue using CORBA's interface repository API to retrieve management metadata in the proposed solution.
Also, the concept of subclassing existing interfaces that form CORBA's IR to record additional management-specific information was described in section 4.3.1. It was not used, however, in this work for the reasons that will be given in 4.3.4; the main reason being the inflexibility of this scheme when trying to integrate other object models.
Whereas the goal of MIR is to record information about the CMIP and CORBA models, the current proposal seeks to create a flexible meta model that allows to extend it easily to integrate metadata about other object models.

4.3.4 Summary

Whereas an extension of CORBA's interface repository to a multi-domain type repository (as described in 4.3.1) forces clients of GOM to enter metadata about the elements of their model into the EIR to enable the manipulation of their instances, the second approach proposed (4.3.2) allows to reuse *existing* metadata and merely requires an adapter for converting existing sources to the generic meta model. The latter approach does not require a client's knowledge of the IR, but only of the generic meta model which – in the author's opinion – is simpler to handle.
In the first approach, there is a gap between the syntax and semantics of GOM's *instance model* and its *meta model*. In contrast, having GOM provide its own meta model plus adapters for bridging to specific metadata sources has the advantage that the overall model of GOM becomes more homogeneous and uniform.
In the second approach, data from specific sources furnished by the metadata adapters is always copied to the metadata cache. This may initially slow down access to the metadata repository, but will pay off in the long run owing to locality of access reasons compared to the first approach where the EIR is always accessed via CORBA, especially in the scenario when the EIR is not in the address space of the client. In this situation, access to the EIR would always result in network traffic, while using a cache will merely result in traffic when the desired information cannot be found in the cache.
A disadvantage of the second approach is that the potential reuse of an existing specification for a metadata repository (CORBA's IR) is not realized, but a basically similar repository is created. Compared to the advantages described above, this disadvantage seems acceptable.
The second approach was chosen as the architecture for GOM's metadata repository because its meta model (e.g. its interfaces) fits more nicely into the instance model of GOM. Rather than providing one type of syntax and semantics (GOM) for the instance model and a different one (the CORBA IR interfaces) for the meta model, in the approach chosen, the instance- and meta model have similar syntax and semantics.
Moreover, GOM's meta model allows from the beginning the integration of metadata from

other models while, with the first approach of extending CORBA's interface repository, many IR classes would potentially have to be subclassed when adding metadata for a model other than CORBA.
Whereas in the first case, metadata providers *have* to write compilers to add their data to the EIR, in the second case, the metadata repository offers a choice between writing compilers and tapping existing metadata sources.

4.4 Adapters

The task of adapters is to enable GOM to communicate with target systems. They are the only place in GOM where target system- dependent code is located. One adapter is required per target system and its task is to *bridge*[27] the generic object model and exactly one target model. It contains knowledge about GOM plus one target model.

4.4.1 Characteristics

To perform the task of mediation between these two worlds, an adapter has to know

- syntax and semantics of GOM,
- syntax and semantics of the 'foreign' target model,
- metadata of the target model and
- metadata of GOM.

The syntax needed to access a target model may be defined by an interface definition language (similar to CORBA), as an API in the form of header files and a C library, or even as exchange of PDUs over a protocol. A description of how to apply the syntax to manipulate the model and what its effects are comprises the semantics.
Metadata about the target model is needed by the adapter to perform type-checking of arguments and conversion of values between the generic object model and a specific target model (and vice versa). Metadata about GOM is needed as well, e.g. when converting a generic GOM value to a target model value. This form of metadata is available in all elements of GOM as `GomKind` (cf. appendix A.1).
Adapters perform mainly three tasks:

1. Type-Checking. When a GOM value is an argument to an operation (e.g. `Execute` or `Set`) a check has to be done whether its type is compatible with the one expected in the target model as defined in the target model's specification of the operation. As the specification is available in the metadata repository (it has previously been parsed and added to the MR), the adapter has to retrieve metadata about the operation to perform type-checking.

[27] The concept of CORBA's *inter-ORB bridges*, which convert requests between different ORBs, is explained in 2.1.4.

2. Conversion. GOM values have to be converted to target model values and vice versa. The task of conversion of a GOM value to a target system specific value can actually be done while type- checking the generic value.

3. Communication with Target System. Eventually, an adapter will have to communicate with the target model to perform the operation requested by the client application. To do this it may need to access the proxy instance (on which the operation was invoked) to retrieve specific information needed for this communication, such as for example a CORBA object reference [OMG95], or an OSI AE-title and distinguished name [ITU92a].

The flow of a request through the system is depicted in fig. 4.36.

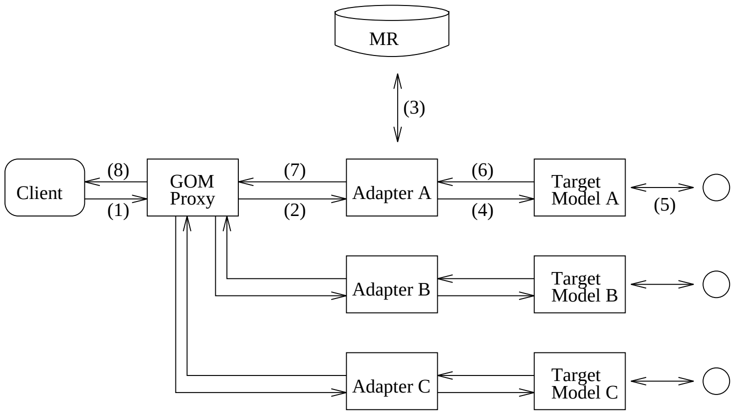

Figure 4.36: Adapters

Consider the case when a client invokes an operation on a proxy instance using operation `Execute` (1). The proxy instance has a reference to its adapter (attribute `adapter`, which was set when the proxy was created) to which the operation is forwarded (2). The adapter then retrieves metadata about the operation using `GenObj::GetOperationDef` (for the CORBA layout) or the more generic `GenObj::GetElementDef` (3). If found, all arguments of the operation are type-checked and converted to target model values. Then the operation is dispatched to the target system (4) in whichever form is appropriate.[28] The target system communicates with its local object to perform the operation (5). When done, the result is received by the adapter (6). This value is still in the target model form and has to be converted to a corresponding GOM value using the previously retrieved metadata. The converted generic value is then returned to the caller proxy instance (7) which finally returns it to the client application (8).
Note that the example uses a synchronous style of execution. For a discussion of an asynchronous model see section 4.6.5.

4.4.2 IDL Interface

All adapters have to inherit from IDL interface `Adapter` (cf. appendix A.1) and implement the operations dictated by the superclass.

[28] E.g. a Dynamic Invocation Interface (DII) request [OMG95], or a CMIP or SNMP request.

The `Create` operation has to create a target instance at a location given in `target_location` and to initialize the resulting instance with an (optional) argument list. Also, an accompanying proxy instance has to be created and the *handle*[29] returned as result of the creation operation in the target system which uniquely identifies the target instance must be stored in that proxy. This enables the proxy instance to later refer to the target instance whenever a request for the latter is received. Attribute `properties` of the proxy instance can be used to store the handle. In the case of CORBA, a stringified version of the object reference might for example be stored under the key `"objref"` in the `properties` attribute. It can be later retrieved by the associated adapter to dispatch requests to the target system. Additionally, the adapter has to store a reference to itself in the created proxy instance (in attribute `adapter`), thus allowing later requests to the proxy to be forwarded to itself.
Operations `Get`, `GetN`, `Set`, `SetN`, `Execute` and `Delete` of an adapter have as first parameter a reference to the proxy instance to which they were originally sent. This allows the adapter to retrieve necessary information such as the handle to the target instance or the classname from the proxy.
The operations to retrieve metadata make use of the adapter's attribute `object_model` which stores the name of the object model the adapter serves. It is used to specify the object model to choose when accessing the metadata repository (MR) (cf. appendix A.4). All metadata repository operations that provide metadata need as first parameter the name of the object model to access the correct metadata cache and/or metadata adapter (4.3.2).
Operations `CreateFilter` and `SendEvent` are used by GOM's *event model*. See section 4.5.2 for an in-depth discussion.
The name given to the IDL interface of an adapter has to be constructed according to a well-defined scheme; it is a concatenation of the name of the object model and the suffix `"_Adapter"`. This is necessary since factories (cf. 4.2.2.2) will always try to instantiate a class `X_Adapter` for an object model named `X` when creating an adapter. The obtained name is also necessary to retrieve an existing adapter from a naming service. In this case, the registration key for an adapter would be the name of its IDL interface as ASCII string, e.g. `"X_Adapter"`.

In the following sections a brief overview of the implementation of the three adapters for CORBA, CMIP and SNMP will be given.

4.4.3 CORBA Adapter

A CORBA adapter has to communicate with CORBA instances in a dynamic manner, with no precompiled knowledge of their classes since there are no included IDL interface definitions available.
Therefore a mechanism must be used that allows adapters (clients in general) to construct requests at runtime and dispatch them to CORBA instances, given an object reference to the instance.
There are essentially two schemes provided by CORBA to achieve this:

[29] E.g. a CORBA object reference, or an OSI AE-title and distinguished name

1. The *Dynamic Invocation Interface (DII)* [OMG95, ch. 4] and
2. The *General Inter-ORB Protocol (GIOP)* and its TCP/IP- based implementation, the *Internet Inter-ORB Protocol (IIOP)* [OMG95, ch. 12] .

DII allows dynamic construction and dispatching of requests to CORBA objects. It makes use of CORBA's *Interface Repository* (IR) to type-check at runtime whether the supplied arguments comply with the formal parameters specified in the IDL definition of the operation.
IIOP is a connection-oriented protocol that allows to send requests to any ORB that implements it. The main purpose of IIOP is inter-ORB 'on-the-wire' compatibility which enables ORBs of different vendors to interoperate.
The difference between DII and IIOP is that the former is an API that transparently allows clients to invoke operations on (remote) CORBA objects without concern for the underlying transport whereas IIOP is a protocol implementing the transport to be used.[30] DII may use IIOP as its underlying transport protocol.
Both DII and IIOP can be used to implement a CORBA adapter. The advantage of DII is that it is more abstract, therefore easier to implement, whereas IIOP is more complex. In a prototype implementation of a DSOM-based [SOM94] CORBA adapter, the DII was used [Ban96b, Ban96f, Ban96a] as described below.
The approach is straightforward. Whenever a GOM request is to be dispatched, after successful type-checking using metadata, a `Request` object will be created. All generic GOM values in the argument list will be converted to CORBA values and added to the `Request` object. Then the request is dispatched to the corresponding target instance[31] using operation `Request::invoke` (synchronous execution). After successful completion, all arguments having mode `INOUT` or `OUT` are converted to GOM values and replace their corresponding (old) values in the argument list of the operation (`Arglist`). This allows clients to retrieve modified arguments after the operation invocation. Finally, the result value is converted to GOM and returned.
For a discussion of how *event handling* is performed by the CORBA adapter see section 4.5.2.5.

4.4.4 CMIP Adapter

The task of a CMIP adapter is to convert GOM requests into CMIP requests, send these to an agent via an OSI stack, collect the CMIP response, convert it to a GOM value and return it to the caller.
There are a number of APIs, most of them in the form of C-libraries, that communicate with an OSI stack to produce requests for the CMIP protocol. Some examples are X/Open's XOM/XMP [XOM94], its successor TMN/C++ [TMN96] or IBM's cmipWorks [GMR94].
The prototype implementation uses the cmipWorks product, which offers a string-based API for ASN.1 values and implements all requests required by the CMIP protocol.

[30] In terms of ODP viewpoints [ODP95, Ban96d], DII would be in the *computational viewpoint* and IIOP in the *engineering viewpoint* [ODP95, DHR92].

[31] The target's object reference is available in the proxy instance.

Processing in the adapter is as follows: when a GOM request is received, all generic values in its argument list are converted to values of the mentioned string-syntax and added to a list. Then the corresponding (synchronous) cmipWorks function is called.[32] It will generate a CMIP PDU and send it to the agent. The response is then received by the adapter and converted to a GOM value which is subsequently returned to the caller.
The address of the agent to which to send the CMIP request is stored as AE-title in the proxy instance. Since the latter is part of the parameter list of most operations of the adapters, it can be accessed to retrieve the AE-title (and other necessary information). (An AE-title was stored in the resulting proxy instance upon creation of it. It was indicated in argument `target_location` of operation`Factory::Create`).
For a discussion of how event handling is performed in the CMIP adapter refer to section 4.5.2.5.

4.4.5 SNMP Adapter

The processing taking place in an SNMP adapter will be described in detail in section 4.6.1. The purpose of this section is therefore to describe some issues related to implementation. Issues related to event handling are discussed in section 4.5.2.5.
As SNMP is based on UDP[33], which is a connectionless protocol, there are some peculiarities to be considered when communicating with an SNMP agent. First, the packet size of UDP is limited. This may require SNMP adapters to break up larger packets into a sequence of smaller ones. One example is the `GET` request: when getting multiple variables at once, it may become necessary to issue a number of consecutive `GET` requests for each variable separately rather than send a single request.[34] An adapter may decide to implement rather sophisticated logic, or it may refuse to do so and return an error.
Another problem caused by UDP is that there is no such thing as a synchronous request (as in the case of CMIP): requests return immediately after sending them to the agent and their responses may arrive later in any order.[35] This requires a scheme that matches requests with responses and possibly an implementation of the *sliding window protocol.*

4.4.6 Summary

Adapters are the links between the generic object model and specific target models. They are the only place in GOM where system-dependent code is written. Their task is to make a target system available for manipulation using GOM and to type-check requests between the generic- and a specific target system.
Adapters are the main clients of the metadata repository, since they need metadata to perform their conversion and type-checking.

[32] There are both synchronous and asynchronous versions of all CMIP requests available. How asynchronous requests could be used is discussed in section 4.6.5.

[33] This is the most common form as described in RFC 1157; however, other datagram services or even connection-oriented services may be used to implement SNMP.

[34] This has been solved in SNMPV2 [CMRW96] with *bulk retrieval.* However, this version is not yet widely used.

[35] Actually UDP does not guarantee delivery; packets may get lost.

4.5 Event Handling

There are mainly two flows of communication between a manager and managed entity: (1) requests from the manager to the managed entity (with responses from the managed entity) and (2) unsolicited *events* from managed entities to managers. Events are situations that occur in a managed entity which may be of interest to a manager and which are sent from the entity in the *managed* role to the one in the *manager* role.
Examples of events are the creation of a new object, a network partition, the shutdown of a printer and so on. Events always carry information describing the situation, such as the name of the newly created instance, or the location of the printer that was shut down. The type of the information is typically defined in the specification language of the model at hand and should be available in the metadata repository.
Event handling is of interest to GOM because, unlike requests initiated from a *manager* and converted to a specific model by adapters, events are initiated by the *managed* entity (specific part) and have to be caught by GOM and converted to the generic model. This is just the opposite flow of control, and a generic event handling model has to be provided by GOM to cover this important area of the architecture of a management system.
In this section, a generic event handling model for GOM is proposed. Its purpose is to shield clients from the heterogeneous event models of CORBA, CMIP and SNMP by offering a higher, generic, abstraction layer conforming to the overall philosophy of GOM that lets clients perform event handling for all target systems. The provision of a generic event service also contributes to the uniformity of GOM.
The structure of this section is as follows: first, an overview of the event models of the three target systems of interest (CORBA, CMIP, SNMP) is given. Additionally a proposal by X/Open for enhancing CORBA's event service will be described. Then the generic event model of GOM will be presented which is on the one hand a *synthesis* of the models described above and on the other hand represents a *generalization* of the (quite specific) event mechanisms of each model, fitting with the generic nature of GOM.

4.5.1 Overview of Existing Event Models

4.5.1.1 SNMP Traps

The event model of SNMP as defined in RFC 1157 [CFSD90] is very simple and consists of *traps* being sent from agents to managers (or from managers to other managers). Traps carry knowledge about an event that occurred in an agent in the form of a trap PDU which has as fields (1) the object identifier of the MIB variable that triggered the trap, (2) the IP address of the SNMP agent that emitted the trap, (3) an enumeration describing a generic set of traps, if this is not sufficient, (4) an integer value describing a specific reason (application-dependent), (5) a time stamp and (6) a list of variable bindings (associations between a name and a value[36]) to further describe the cause of the trap.
The address of the target manager to which traps generated by an SNMP agent are to be sent is implementation dependent and there is no standardized way for managers to

[36] A value is a choice of the permitted ASN.1 types for SNMP (e.g. `INTEGER`, `OCTET STRING`, `Gauge` etc.).

register their interest in certain types of events as in the case of OSI event handling (cf. below).

4.5.1.2 OSI Event Handling

The event model of OSI is defined in X.734 (Event Report Management Function). Whenever managed objects (MOs) need to inform the agent about the occurrence of a new situation they send a *notification* to the agent's *event forwarding discriminators*. These are special MOs that evaluate the notification against filters set by managers and forward those that pass the filter to a suitable manager in the form of *event reports*.
An event report is a structure similar to SNMP traps containing (1) the class object identifier (OID) and (2) the distinguished name of the managed object that triggered the event, (3) the event time, (4) the type of the event information (OID) and (5) the event information itself. The syntax of an event report is defined using ASN.1.
Filters are represented by instances of *event forwarding discriminators* (EFDs) (derived from GDMO template `discriminator`) in an agent. EFDs contain a number of attributes that can be modified to change their behavior, e.g. suspension, resumption, scheduling time etc.[37]
Two important attributes are the *discriminator construct*, which is a filter containing constraints on the notification's parameters, and the *event destination*, which is a list of addresses (AE-titles) to which events that pass the filter are to be sent. These will typically contain the addresses of the managers that registered for the events.
As these attributes may be modified at runtime it is possible to change the destination list or discriminator construct. This may for example be used to un-subscribe a manager from event reports or to further restrict the number of event reports being sent to a manager by putting an additional constraint on the discriminator's list.
When a notification is generated by a managed object in an OSI agent, it will be sent to all EFDs present in the agent. These evaluate the EFD's discriminator construct against the notification's parameters and either reject the notification or pass it on in the form of an event report to each destination specified in the destination list.

4.5.1.3 CORBA Event Service

The OMG event service [COS95, SV97] allows *suppliers* to send events to *consumers* either directly or via *event channels* which decouple direct communication. A consumer may be notified (*push model*) whenever a new event has been added to an event channel, or it may retrieve (*pull model*) events actively from the channel. Likewise, suppliers can be pulled for events or actively push events on the channel.
Event information may be typed (using normal operations and data types of a consumer object) or untyped (using type `any`).
A consumer subscribing to an event channel will receive all the events added to the channel without a possibility to specify filters that discard unwanted events. OMG has issued an

[37] Most of the attributes of an EFD may actually be absent since they are contained in conditional packages.

RFP for a notification service that should augment the basic event service with more specialized capabilities such as event filtering.

4.5.1.4 X/Open Event Management Service

X/Open's Event Management Service (EMS) [X/O96] is a proposal for OMG's notification service RFP [Obj97] that exceeds the scope of CORBA's event service. It consists of *consumers* and *suppliers* of events, communicating using an *event channel*. Consumers register for certain events by providing *filters*. When an event is received by an event channel, it will be matched against all filters and sent to registered consumers if the filter is passed.[38] A filter consists of a list of expressions of a name, a boolean operator and a (self-describing) value. Each item of the list is matched against the attributes of an event (which are self-describing as well).
Besides the central event channel, there are 4 repositories: the *filter repository* stores all filter constructs defined by consumers and provides an interface for creation, deletion and modification. The *schema repository* maintains metadata about event types, optionally supporting the filter evaluation process.[39] The *event repository* is a volatile queue for events. Events are dequeued whenever a consumer pulls an event from the queue. Finally, the *consumer/supplier repository* maintains a list of all currently connected consumers and suppliers.
It is noteworthy that all data in events and event filters are self-describing, i.e. they consist of names and values. The latter is a struct containing a *type tag* and the actual value. This facilitates filter evaluation since the algorithm is generic for all possible values. In this respect, they resemble GOM's generic values (cf. section 4.2.2.2).

4.5.2 The Generic Event Model of GOM

A synthesis of the systems described above yields a number of commonalities generally found in event handling systems:

Consumers These are the entities in a system that are interested in the occurrence of certain events and are therefore notified (push model) by the event service when an event occurs, or they actively poll the event service periodically for events (pull model).

Producers Producers generate events and supply them to the event service.

Event Service An event service (or event channel) decouples consumers from producers and performs some computation on the events supplied by producers. It stores events in a queue.

[38] It is also possible for consumers not to register with the event channel (push model), but to actively retrieve events from the event channel (pull model).

[39] In case either the elements of an event or an event filter are not self-describing, metadata will be used for filter evaluation.

Event An event is the occurrence of a situation in a managed system that may be of interest to a manager. Each event carries information describing its cause in more detail.

(Target) Filter Filters are usually types or classes whose instances are created in a target system (1) to constrain which events are sent to management systems and (2) to specify the destination(s) of consumers to which events that pass the filter construct are to be sent. The filter construct consists of boolean expressions over the event information.

In addition to these concepts the generic event model proposed here uses the following concepts:

Adapters The generic event model of GOM requires adapters (cf. 4.4) to

1. act in the manager role as recipients for events from the target system that the adapters represent,
2. convert the received event information into a generic form as required by GOM and
3. dispatch the event (plus information) to all consumers that have registered their interest in the filter (and add it to the event queue).

Proxy Filter Proxy filters are local handles for remote filters in target systems. Any operation invoked on proxy filters will be forwarded to the corresponding target filter.

Local Filter Local filters are present only locally, e.g. in the client's address space. They allow to specify (additional) constraints on incoming events, thus deciding whether to forward or discard them. Their purpose is (1) to *emulate* filtering capabilities in GOM for event models that do not possess them (such as SNMP or CORBA's basic event service) and (2) to allow for additional filtering stages in the local system e.g. for temporary testing purposes.

4.5.2.1 Architecture

The architecture of the generic event handling model of GOM is shown in fig. 4.37.
The central piece of GOM's event handling model is the *event service*. It is a storage for events that consumers can access to retrieve events and to which producers can supply events. Managed systems (in the role of producers) generate events and – with the help of adapters – add them to the event service, whereas managers (as consumers) use it to retrieve events.
An instance of the event service interface (cf. `EventService`, app. A.3.) will typically be present locally on every machine running a GOM-based management application although – the event service being merely another CORBA interface – it may be located anywhere on the network. Initial retrieval of an object reference to the event service will be by means of the CORBA naming service [COS95].

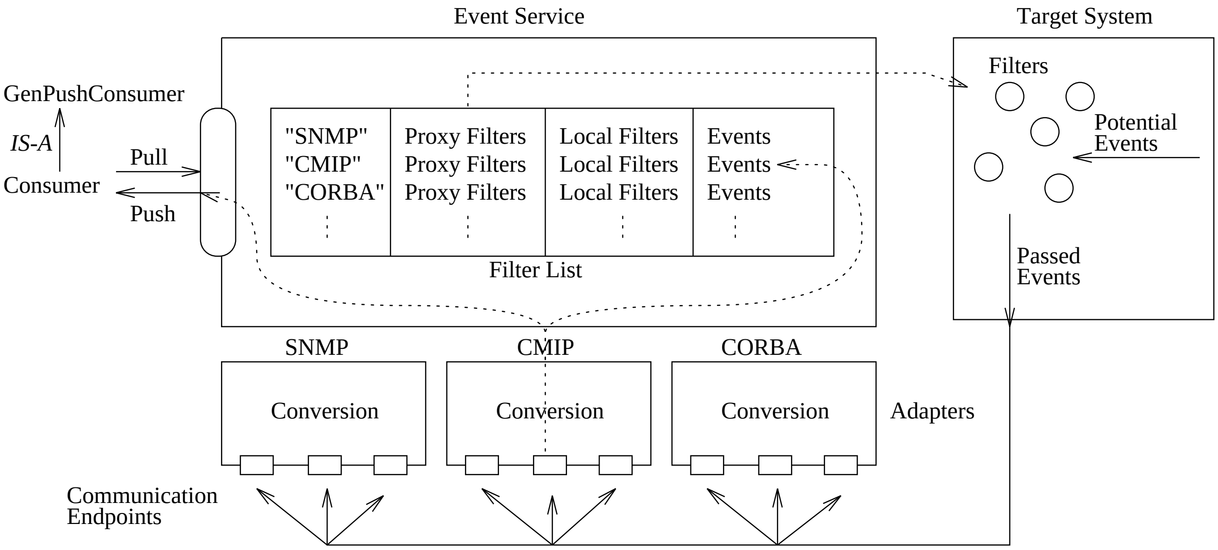

Figure 4.37: Event handling architecture of GOM

Consumers may register their interest for certain types of events by creating *proxy filter* objects which represent a target filter object (or type) in a target system. Proxy filters have an attribute for the destinations to which any incoming event should be sent. All destination objects have to be subclasses of interface `GenPushConsumer`. As soon as an event is received it will be sent to all registered destinations (consumers) by invoking operation `HandleEvent` on all object references.[40]
Consumers may choose active event polling by calling operation `PullEvent` on the event service, which will remove the first element of the queue and return it to the client. This operation will block if no event is available unless argument `wait` is false.
When a new filter object is created in a target system, the corresponding adapter will create a *communication endpoint* in itself to which all events are to be sent. The destination (list) of the target filter will then refer to that endpoint. Any event that passes the filter object will therefore be sent to the adapter (see below).
Local filters allow to specify constraints (in a simple boolean language) that are applied to incoming events before adding them to the event service.
In the next sections the IDL interfaces of the main components of GOM's generic event model will be explained. For the complete definitions see appendix A.3.

4.5.2.2 Events

The event service contains events that are provided by producers and retrieved by consumers. An event carries information which is modeled by interface `EventInfo`. This is a `Struct`, essentially a list of attribute names and values (cf. appendix A.1). All events received by adapters from target systems have to be converted to an instance of `EventInfo` before being submitted to the event service.

[40]As this operation may become blocked in consumer code, it will preferably need to be handled in a separate thread.

4.5.2.3 Proxy Filters

These are proxy instances for filters located in target systems. Both target and proxy filter are created using operation `EventService::CreateFilter` (see fig. 4.38). Proxy filters have two attributes:

- A reference to the target filter and
- A list of consumers to which incoming events will be forwarded.

The target filter may be an instance (e.g. a managed object of template `eventForwardingDiscriminator` in [ITU93]) or a value (e.g. a struct). It will typically contain a filter construct against which potential events are evaluated (cf. fig. 4.37) and forwarded or discarded, and accordingly a list of destinations to which passed events will be forwarded. These destinations will be tweaked by adapters that create the filters to forward all events to the *adapters' communication endpoints* rather than to individual objects of that target model.
The attributes of a filter may be manipulated using the proxy filter, similar to proxy instances in GOM's instance model. A client may for example wish to modify the `administrativeState` attribute of an `eventForwardingDiscriminator` in CMIP to disable/enable a filter for a period of time or to set scheduling times in it.
Note that `ProxyFilter` objects do *not* perform any filtering at all, but are only generic placeholders for filters in target systems which do the real filtering work. However, they can be used to change the behavior of their corresponding target filter objects by modification of attributes in the proxy filter (which will transparently be propagated to the target filter).

4.5.2.4 Event Service

The `EventService` IDL interface (shown in fig. 4.38) is the main entity of GOM's generic event model.
It maintains a *filter list* (see fig. 4.37). In it, all local- and proxy filters are kept for each object model, together with the events waiting to be pulled from the event queue by clients.
Operation `CreateFilter` instantiates a new filter object in the target system and – if successful – a corresponding proxy filter which is returned to the caller. The parameters of this operation are (1) the name of the object model so that the correct adapter can be selected, (2) the name of the class of the filter[41], (3) the target location which is the address of the target system in which the filter is to be created[42], (4) the name of the newly created instance[43], (5) an attribute-value list initializing certain attributes of the newly created instance with values and (6) a list of consumers which will be notified whenever an

[41] E.g. `"eventForwardingDiscriminator"`. In cases where filters are not modeled through classes, this may also be the name of a type, e.g. a struct or a simple string.
[42] E.g. the AE-title of an OSI agent, or the name under which a CORBA event service is registered with a naming service.
[43] E.g. the distinguished name of a managed object.

```
interface EventService {
    ProxyFilter   CreateFilter(in string object_model,
                               in string type,
                               in string target_location,
                               in string target_name,
                               in Arglist args,
                               in ConsumerList consumers);

    LocalFilter   CreateLocalFilter(in string object_model,
                                    in string oql_expr,
                                    in ConsumerList consumers);

    void          AddFilter(in string object_model,
                            in Val new_filter);

    FilterList    GetFilters(in string object_model);

    EventInfo     PullEvent(in string object_model, in boolean wait);

    void          PushEvent(in string object_model,  // used by adapters
                            in FilterList filters,
                            in EventInfo event_info);

    void          SendEvent(in string object_model,
                            in EventInfo event_info,
                            in string destination_address);
};
```

Figure 4.38: Interface `EventService`

event passes this filter. Each consumer has to be a subclass of `GenPushConsumer` whose operation `HandleEvent` will be called if an event occurs (push model).
A consumer may choose not to subscribe to a certain filter, but rather to retrieve events from the event service actively. This is done through operation `PullEvent` which returns the next available event or waits until one is available (if argument `wait` is set to true.).
Local filters are objects of their own, i.e. they are not proxies for a target object. They are used to specify additional constraints on incoming events and are suitable to discard unwanted events of target systems for which there is currently no filtering scheme available (such as SNMP).[44] The difference between filters in target systems and local filters is that the former perform filtering in the target system and send only those events that pass the filter to the event service (which saves bandwidth) whereas in the latter case, all events are sent from the target system to the event service and filtering is performed in the event service itself.

[44]Note that in SNMP the `Gauge` type actually provides some primitive form of filtering.

Local filters contain a list of consumers and a filter construct which is currently a string containing a (proprietary) string-based boolean expression[45] which has to be evaluated (i.e., parsed) by adapters against the information carried in the event (`EventInfo`). Note that the conversion between the string-based constraints and typed values for evaluation of filters is easy by means of GOM's instance model which represents values as objects of their own and offers access to attributes given their names (in string form). Evaluation of filters against strongly-typed values (as in the case of XoJIDM's approach, cf. 3.1.2) would be awkward, if not impossible.
If operation `CreateFilter` is not sufficient, e.g. by not offering all parameters to create filters in target systems, the latter may be created by the clients themselves and subsequently added to the filter list for a certain object model.
Operation `GetFilters` returns all filters for a certain object model.
`PushEvent` allows adapters to submit an event to the event service for further processing. When an event is received by the event service, it will perform the following steps: first, if there are any local filters available for the object model specified in the `PushEvent` operation, these will be evaluated against the event information carried in the event. If the event does not pass the local filter, it will be discarded. Otherwise, it will be pushed to all registered consumers and afterwards added to the event queue. See use cases (4.5.2.6) for more details.
Operation `SendEvent` is used by clients to send events to other management entities, e.g. in the case when an event cannot be handled by a local manager and has to be forwarded to a superior management application. Parameter `destination_address` may for example be the AE-title of another OSI manager.

4.5.2.5 Adapters

In the event handling model of GOM, adapters have the following major tasks:

1. Creation of target filters and their corresponding proxy objects.
2. Reception of events sent from target filters.
3. Conversion of target system specific event information `EventInfo`.
4. Submission of event to event service.
5. Sending out of events; conversion of `EventInfo` to target system specific event information, dispatching of events to target systems.

The latter task is initiated by a client to emit an event to some other management entity, whereas the first four deal with receiving and converting events sent by managed or management entities.
Adapters have to intercept events sent out by target filters by 'tweaking' the latter's destination to point to an event destination *in the adapter* so that all events are routed to the adapter, and not directly to consumers. This is done by the adapter when creating a new target filter as shown in fig. 4.39.

[45] The syntax is not yet determined, but syntaxes proposed e.g. for OQL [ADF$^+$94] or the OMG query service [COS95] could be used.

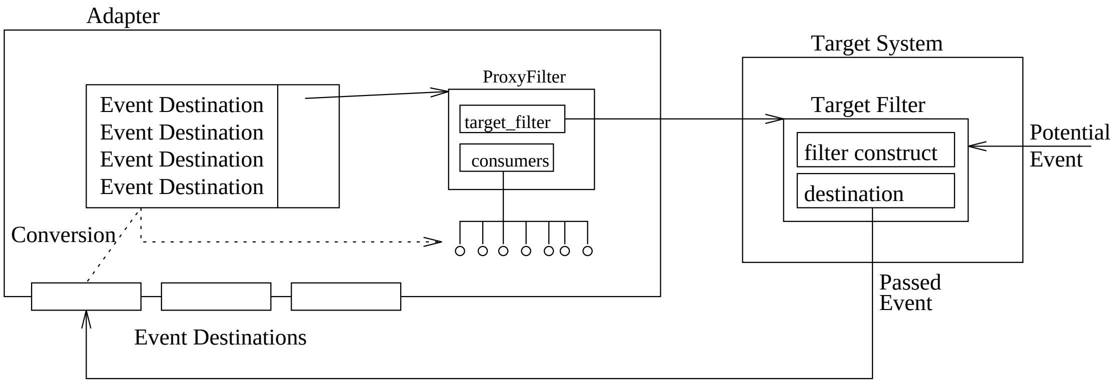

Figure 4.39: Event handling in adapters

For each target filter, the adapter creates an *event destination* to which the target filter's destination will point. The association between the resulting proxy filter and the event destination is recorded in a table within the adapter. When an event is received at a certain event destination, the latter will be looked up in the table to retrieve the corresponding proxy filter. Having retrieved the filter, operation `PushEvent` of the event service will be called which in turn will push the event to all consumers of the proxy filter and then store it on the event queue.[46]
An event destination may simply be a UNIX socket as in the case of SNMP, an AE-title in the case of OSI, or a reference to a *proxy push consumer* (CORBA [COS95]). What to choose as key in the table and how to match keys is largely implementation dependent. It might also be the case that keys have multiple associated proxy filters, effectively mapping multiple filters to the same destination.
An overview of a possible implementation of SNMP, CMIP and CORBA adapters with respect to event handling is given in the following paragraphs.

CORBA Adapter Whenever a new *target filter* is created, an adapter has to create an *event sink* (event destination) which will receive and process all events sent out by the filter. An overview of how this can be done using CORBA's event service [COS95] is given below.
When a filter is created using as name argument `target_name` of operation `CreateFilter`, the name is looked up in a naming service. If not found, a new *event channel* is created and registered under that name with the naming service. All subsequent times this reference will be returned by the naming service.
Next the adapter retrieves a *proxy push consumer* from the event channel. All events pushed on the event channel will be received by this instance.
The proxy push consumer is then registered in a table (maintained by the adapter) as *key*. Its corresponding value is the reference to the generated proxy filter. This has the effect that, whenever an event is pushed on the proxy push consumer instance, the following steps are taken: (1) the corresponding proxy filter is looked up in the table using the object reference of the proxy push consumer instance on which the event was received.

[46] A similar solution is described in [Gen96, ch. 4.5.3].

(2) The event information carried in the event is converted to a generic GOM event value and (3) pushed to all consumers registered with the proxy filter instance.

A request for proposal (RFP) for a *notification service* based on – and more powerful than – the event service supporting among other things *filters* has been issued by the OMG [Obj97]. It will be beneficial for CORBA adapters to support this new service in order to provide more sophisticated event service capabilities.

CMIP Adapter When a target filter is created, the adapter will create a managed object of template `type` as given in the argument list of operation `CreateFilter`. Its distinguished name will be taken from argument `target_name`, and `target_location` should be the AE-title of the agent in which the filter instance is to be created. Filter attributes (such as the filter construct) should be defined in the attribute list of the operation.
The following processing will take place: (1) the target instance will be created (e.g. an instance of `eventForwardingDiscriminator`) in the agent with the given AE-title under the given distinguished name and (2) an event destination to receive and process incoming events (and to which the destination of the created filter points) will be created (if not yet existent). The event destination might for example be the address of a manager's event reception part in which a thread waits for incoming events. (3) The AE-title will be added as key to a table and will be associated with the proxy instance that was previously created for the target filter. Thus, incoming events can later be pushed to all registered consumers of the proxy filter instance. This is done by looking up the AE-title of the event destination in the table. Its value will point to a proxy filter in which all consumers are available in attribute `consumers`.

SNMP Adapter The address of the target manager to which traps generated by an SNMP agent are to be sent is implementation dependent. There is no standardized way for managers to register their interest in certain types of events as in the case of OSI event handling. Therefore any implementation of an SNMP adapter will have to use a non-portable mechanism to receive traps from an agent.
The scheme proposed here is to use UNIX sockets on which a thread in the adapter is listening and to make the agent send traps to that port. How the agent knows to which ports to send traps is again implementation-dependent (e.g. using configuration files).
Since SNMP does not support filters, operation `CreateFilter` is a NIL operation and will not be used. Instead, operation `CreateLocalFilter` can be used. It adds a number of consumers to a local filter and adds the latter to the event service's list for the SNMP model. A NULL `oql_expr` argument will pass all events received on to the registered consumers. This is essentially the SNMP model unchanged. Defining a simple string-based expression allows to implement filtering capabilities for SNMP as well. The `filter_construct` of a local filter will be parsed and evaluated by an SNMP adapter against incoming events, discarding events that do not pass the filter construct.
Since SNMP does not support the concept of target filters there is no need to register event destinations with corresponding proxy filters as in the case of CMIP or CORBA.

Instead, the adapter will simply listen on a UNIX socket at a certain port that has to be made known to the agent in order to receive traps.
When a trap is received it will be converted to a generic event and normal processing will continue as discussed above.

4.5.2.6 Use Cases

To illustrate the control flows within the generic event system, two use cases are presented below.

Creation of a Target Filter

1. Operation `CreateFilter` is called by a client on the event service object (created previously or retrieved from a naming service).

2. The correct adapter for the indicated object model is retrieved to forward the operation to.

3. An instance of class `classname` is created by the adapter in the target system at location `target_location`. The location is specified as a string and may be the name of a CORBA server or an OSI AE-title. An argument list as parameter is used to initialize the newly created instance. The adapter makes use of a factory (cf. 4.2.2.2) to create the remote target filter instance. In case a target system does not have filters, the target filter to be created can be thought of more as a *conceptual* filter in that *all* events will be passed unfiltered to the adapter. In the case of SNMP, the class name and argument list would be empty, whereas the instance name could, for example, contain the host name (and optionally the port) of the SNMP agent whose traps are to be intercepted. In the case of CORBA's event service, the instance name could denote the location of an *event channel* and a `ProxyPushComsumer` reference could subsequently be retrieved from the channel [COS95].[47] In the case of OSI, the target filter could be an `eventForwardingDiscriminator` instance.

4. A proxy filter instance (`ProxyFilter`) is created that points to the target filter (attribute `target_filter`). Attribute `consumers` is set to the value of the previous argument of the creation operation (`consumers`). This instance will subsequently be used by clients to manipulate the 'real' filter instance.

5. A local event destination address is created in the adapter on which, for example, a thread listens for incoming events.[48] The destination address is added as key to a table[49] and associated with the proxy filter instance. This is necessary to later retrieve all consumers registered for a certain event when dispatching an event (see below).

[47] 'Real' filters may be provided in the prospective OMG notification service ([Obj97]) of which the generic event service could make use in the future.
[48] Other schemes are possible, such as listening on multiple sockets simultaneously (e.g. using SELECT).
[49] If the key already exists, the value (proxy filter) will be added.

6. The destination of the newly created target filter representing consumer(s) to which to send passed events must be tweaked to point to the newly created event destination in the adapter. Thus, all events will be intercepted by the adapter.

7. The proxy filter instance is added to the event service's filter list using operation `EventService::AddFilter`.

8. The proxy filter instance is returned to the client.

Occurrence of an Event

1. An entity within a target system emits a potential event (e.g. a managed object in an OSI agent emits a notification).

2. The filter in the target system evaluates the attributes of the event against its filter construct (see fig. 4.39). The event will be discarded if it does not pass the filter construct.

3. If the event passes the filter construct, it will be sent to the destination(s) indicated in the filter. This attribute was tweaked by the adapter that created the filter to point to a destination address within the adapter. Thus all events that pass the filter are sent to an address within the adapter. This is the point where an event enters the generic event model of GOM.

4. The adapter receives the event (still in the form of the target system).

5. The information carried by the event is converted into the generic form (an instance of `EventInfo`) using the metadata repository.

6. The address of the communication endpoint (destination) at which the event was received is used as lookup key into the table in which destinations and their associated proxy filters are recorded (see fig. 4.39) to find the relevant proxy filter(s).

7. Operation `PushEvent` of the event service is called with the name of the object model, the previously retrieved proxy filters and the converted event information.

8. `PushEvent` performs the following steps:

   (a) If there are any local filters available in the filter list of the object model at hand, each local filter's filter construct will be evaluated against the event information. If the event passes the filter construct, operation `HandleEvent` will be called on all registered consumers of that filter.[50]

   (b) Then all registered consumers for the proxy filters will be called with the same operation. The consumers that are to be called are stored in attribute `consumers` of each proxy filter of the list that is the value of argument `filters`. Note if the proxy filter list is NULL, this step will simply be skipped. This is for example the case in SNMP. Here, only the local filters (if available) will be evaluated.

[50] Consumers have to be derived from interface `GenPushConsumer`.

(c) The event is finally added to the event queue. Further `PullEvent` operations may retrieve an event by removing it from the event queue.

4.5.3 Summary

The proposed generic event service provides simple event handling capabilities for a number of target systems (specifically SNMP, CMIP and CORBA) and – in conformance with the philosophy of GOM – shields clients as much as possible from the differences found in those systems. Transparency is achieved by providing a generic layer consisting of a synthesis of the event handling concepts found in SNMP, CMIP and CORBA, which levels the idiosyncrasies employed by the event handling schemes of different systems.

The event service proposed here is modeled after CORBA's event service [COS95], and the concept of filter objects has been taken from the OSI event report management function [ITU93].

In line with the philosophy of GOM, new event handling schemes can be integrated by provisioning additional adapters which constitute a bridge to the new system, knowing how to create filters in these systems and how to receive and convert events to the generic form.

The goal of the generic event service was to make event handling of heterogeneous target systems as transparent as possible, in conformance with GOM's overall model and bridging the differences between heterogeneous event models.

4.6 Other Issues

In this section issues that were not (or only briefly) touched upon in the previous sections are discussed. They do not contribute to the overall conceptual picture of GOM, but are important for both clients using the framework and developers implementing extensions (e.g. integration of new target models).

The purpose is to describe ideas that will enhance GOM's usability, e.g. by adding functionality, for example for taking into account idiosyncrasies of other object models.

The discussion of the issues presented here is not at a very detailed level. Instead. the purpose is to make the reader aware of problems/open issues in connection with GOM and to outline possible schemes for coping with them.

Some of the issues mentioned are targets of further work, some of them have already been implemented.

4.6.1 Integration of SNMP

The Simple Network Management Protocol [CFSD90, Bla92] has found widespread acceptance for management. Unlike the other two models (CORBA, CMIP) that GOM tries to integrate by providing a uniform API, SNMP is not object-oriented, running counter to GOM's basic assumption of the target models being object-oriented.

The purpose of integration of SNMP into GOM is therefore twofold:

1. SNMP is the de-facto standard for LAN device management. GOM simply *has* to integrate it in order to be useful. There are a lot of network resources managed by an SNMP agent. Access to these from GOM may be crucial.

2. It will be shown that the generic nature of GOM's framework makes it possible to integrate even non object-oriented models.

In this section a proposal of how SNMP can be integrated into the overall framework will be presented. Other integration schemes can be considered which are implemented by providing metadata and an SNMP adapter.
As described in 2.3, information in an agent's MIB is represented through *variables*. A variable has a type that defines the values it can accept. Variables are arranged in tree form, with *tables* representing conceptual *collections of variables*. The variables in a table can be retrieved one after another using the SNMP `GET-NEXT` request.
The term MIB (Management Information Base) is used to denote both the specification (ASN.1 macros) and the runtime representation (structure in SNMP agent's memory) of such a tree.[51]
The overall procedure to integrate SNMP into GOM is the same as for other models: first the specification of an SNMP agent's model – a MIB – is parsed and added to the metadata repository.[52] Then the agent can be manipulated using GOM's instance model.

First a description of how the instance model is used to access SNMP variables follows. Then an overview of the mapping of SNMP MIB's to GOM's meta model will be given.

4.6.1.1 Accessing SNMP Agents Using the Instance Model

A possible mapping between SNMP and GOM's instance model is shown in table 4.7.

SNMP	**GOM**
MIB	Instance of `GenObj`
Variable	Attribute of `GenObj` (`Val`)
Group / Table	List of `Vals`

Table 4.7: Mapping of SNMP to GOM.

An SNMP MIB is considered a container with a (potentially large) number of elements (obeying a certain structure). Translated to object-oriented terms a MIB is seen as a class and its variables as attributes of the class. Each MIB is represented at runtime by an instance of that class and access to the MIB's variables is provided by retrieving attributes of the instance. A group or table of SNMP variables is seen as a list of attributes.
Adopting this scheme to GOM, MIBs are represented by `GenObj`, where each instance of `GenObj` refers to a different MIB. Operations invoked on these instances retrieve or set SNMP variables, which are modeled as GOM values.
The mapping of SNMP requests to GOM operations is shown in table 4.8.

[51] In CMIP, the term Management Information Tree (MIT) is used to denote the runtime structure.
[52] Actually, if metadata about a certain MIB is already available, this step may be skipped by using just-in-time metadata providers in the form of *metadata adapters* (see section 4.3.2).

SNMP request	GOM operation
Identification of SNMP agent	Creation of MIB (`GenObj`)
GET (1 variable)	GenObj::Get
GET (multiple variables)	GenObj::GetN
GET (all variables)	GenObj::GetN
GET (table)	GenObj::Execute
GET-NEXT	GenObj::Execute
SET (1 variable)	GenObj::Set
SET (multiple variables)	GenObj::SetN
TRAP (send trap)	GenObj::Execute

Table 4.8: Mapping of SNMP requests to GOM operations

In order to manipulate SNMP variables, a reference to a MIB (in the form of a `GenObj` instance) has to be retrieved. This is done either by creating a new instance or retrieving a previously created one, e.g. by using CORBA's naming service. Using `GenObj`'s `Get`-, `Set`- and `Execute` operations, generic values representing SNMP variables can be retrieved or set.
The following sections describe how operations offered by GOM's instance model can be used to manipulate SNMP agents.

Identification of an SNMP Agent A reference to a MIB is created using the `Factory::Create` operation (see 4.2.2.2), adopting the following conventions: the `object_model` parameter must contain the string `"SNMP"` and the classname must denote the MIB to be manipulated. This may for example be `"MIB-II"` to denote the Internet MIB as described in RFC 1213 [RFC91b]. Parameter `instance_name` can be used to give the resulting instance a description. The location of the proxy instance is determined by `proxy_location` (usually this will be NULL to create a local proxy) and `target_location` should contain a string identifying the address of the SNMP agent (e.g. `"adlerhorn.zurich.ibm.com:161"`). Additional arguments such as the community can be specified using parameter `args`. All of this information will be stored by the SNMP adapter in the resulting proxy instance (`GenObj`) for further reference. Otherwise, the creation of a MIB proxy does not generate any communication with an SNMP agent. Communication will only take place when an operation is invoked on the proxy MIB.

Getting an SNMP Variable The value of a variable can be retrieved using operation `Get` on the newly created MIB instance. It takes as parameter the name of the variable either in symbolic form (e.g. `system.sysDescr.0`) or in OID form (e.g. `1.3.6.1.2.1.1.1.0`)[53] and returns a GOM value that is the result of applying the mapping described in table 4.9.
In case of an error, an exception (`GenEx`, cf. 4.2.2.3) will be thrown that contains the

[53] The test for discrimination between OID and symbolic form is simply by checking whether the first character is a digit and, if it is, whether it is a `0`, `1` or `2`.

SNMP Type	**GOM Interface**
NULL	NIL
INTEGER	Int
OCTET STRING	Str
OBJECT IDENTIFIER	Str
SEQUENCE	Struct
SEQUENCE-OF	Sequence
NetworkAddress	Union
IpAddress	Str
Counter	Long
Gauge	Long
TimeTicks	Long
Opaque	Str

Table 4.9: Mapping of SNMP types to GOM

following attributes: `name` will be NULL, `ex_type` will contain `TARGET_EX` and `members` will contain the keys `"errcode"` with an integer value and `"errstr"` with a string value describing the error as defined in RFC 1157.[54] Optionally there may be an additional key named `"ErrorIndex"`, which has an integer as value. If variables are returned as part of the exception (corresponding to field `"variable-bindings"` in SNMP's GetResponse PDU), the `"members"` dictionary will contain the names of the variable bindings as additional keys and their values as corresponding values.

Operation `GetN` can be used to retrieve a set of variables at once. It has a dictionary as parameter which contains name/value pairs (cf. appendix A.1). To retrieve variables a, b and c, a dictionary with keys a, b and c has to be the argument of `GetN`. Upon completion of the operation, for each key a corresponding value will be available in the dictionary (provided the variable was found and readable).

`GetN` can also be used to retrieve all variables of an SNMP agent by providing an *empty* dictionary. The adapter will retrieve all variables by walking through the MIB and adding them to the dictionary.

A *table* is a collection of variables that 'belong' to the same parent. Tables are merely a conceptual entity and there is no SNMP request available to get all elements of a table in a single request. Rather, `GET-NEXT` has to be used to traverse a table sequentially.[55]

Unlike SNMP, GOM offers a higher-level API and can thus shield the user from several successive `GET-NEXT` requests (including determination of end of table) by providing operation `Execute` to retrieve all elements of a table.

By convention, argument `opname` of operation `Execute` will be `"GetTable"`. The only element of the argument list will be the name of the variable that marks the conceptual beginning of the table (e.g. `"interfaces.ifTable.0"`). This element will be removed from the argument list upon successful completion, and the elements constituting the

[54] Key `"errcode"` corresponds to field `"ErrorStatus"` of an SNMP GetResponse PDU.

[55] There are algorithms available to determine when a table ends, but they are not described here.

result set will be added to the argument list.

Request `GET-NEXT` can be invoked by using operation `Execute` as well. The convention adopted in this proposal is to name the operation `"GetNext"` and to add all variable names to be retrieved[56] to the argument list (`Arglist`) of the operation. Upon return of the operation, the variable names in the modified argument list will contain corresponding string values (instances of `Str`) with associated values (instances of `Val`) that represent the next element for each variable in the list.

Setting an SNMP Variable A variable in an SNMP agent's MIB can be modified (if it is not read-only) using operation `Set` of the retrieved MIB instance (a `GenObj`). Parameter `attrname` specifies the name of the variable either in symbolic or OID form (see above), `new_val` contains a GOM value which is the result of applying the (reverse) mapping of an SNMP value to a GOM value as shown in table 4.9.

As SNMP's SET request allows to modify multiple variables at once, GOM has to provide for this possibility as well using operation `SetN`. Similar to `GetN`, it requires a dictionary as parameter which contains the names and corresponding values of the variables to be set.

Sending an SNMP Trap SNMP traps are usually sent from agent to manager. However, a manager may also wish to send traps, e.g. to another manager to inform it of some problems.

Unlike CMIP's event report management function [ITU93], SNMP does not define a mechanism that records to which manager(s) a trap is to be sent, and how managers register for traps with an SNMP agent; RFC 1157 leaves this behavior implementation-specific.

Therefore a parameter is added to the TRAP request, thus allowing the sender of a trap to define the trap's destination address (see below).

As described in 4.5.1.1 on page 95, an SNMP trap carries fields about the object that triggered the trap, the address of the management entity that sent the trap, a generic and a specific integer number describing the problem and a list of bindings between variable names and values to provide additional information.

The TRAP request can be mapped to operation `Execute`, with argument `opname` set to `"SendTrap"`, thus following a similar solution as the one adopted for `GET-NEXT` (see above).[57]

The operation's argument list will contain the elements shown in table 4.10.

The names of all parameter are the same as the ones listed in RFC 1157, with the exception of the destination address to which the trap is to be sent, which is required by GOM. Fields `agent-addr` and `time-stamp` are missing because they will be provided by the SNMP adapter directly.

[56] SNMP's `GET-NEXT` request allows multiple variables to be specified in the same call.

[57] Thus the SNMP adapter has to differentiate between `"GetNext"`, `"SendTrap"` and possibly other operation names to invoke the desired behavior.

Name	Value	Description
destination	Str	Address of recipient, e.g. in the form "9.4.21.218:161"
enterprise	Str	OID of the object generating the trap (may be empty)
generic-trap	Int	Generic trap type
specific-trap	Int	Specific trap type
variable-bindings	Struct	Bindings between variable names and (GOM) values

Table 4.10: Arguments to the TRAP operation

4.6.1.2 Mapping SNMP MIBs to the Meta Model

The Management Information Base (MIB) structure of an SNMP agent is defined in a document (MIB specification) that lists all variables of the MIB, together with their OIDs, symbolic names, syntax and access permissions.
In order to manipulate an agent's variables dynamically at runtime, the MIB's specification has to be available in metadata form. Therefore, the textual description of a MIB has to be translated to an electronic version.
To this end, a parser is used which reads MIB specifications and adds them to the metadata repository. If existing electronic MIB definitions are available online, metadata adapters can be used to provide metadata in GOM's generic form by tapping these existing sources. In the latter case, no MIB parser is needed. See section 4.3.2 for a detailed discussion of the metadata repository.
As GOM's meta model (cf. 4.2.3) is generic and allows to represent any type of model, we have to describe how a specific model is represented in the meta model using a *layout* (see 4.2.3.2).
The layout for SNMP is shown in graphical form in fig. 4.40 and in textual form in appendix B.3.

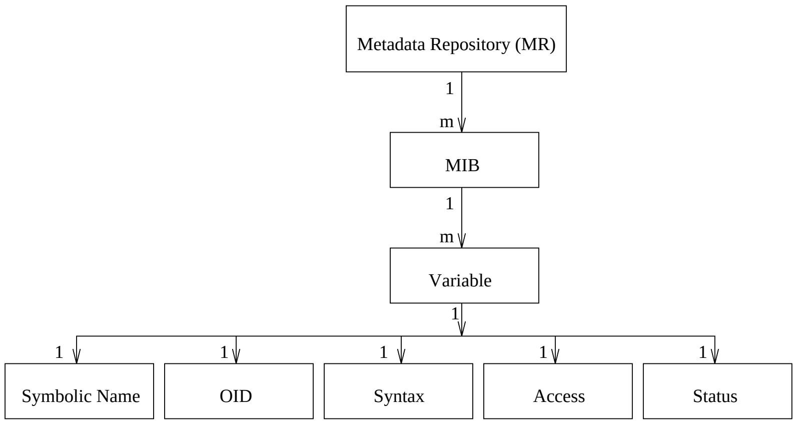

Figure 4.40: The layout of SNMP

Conceptually, the metadata repository maintains a separate database for each target (object) model in which the metadata for that model is available in electronic form. The

database for SNMP contains a number of MIBs, each identified by a name, e.g. `"MIB-II"` defined in RFC 1213 [RFC91b]. A MIB contains a number of variables. Each variable has an OID (e.g. `"1.3.6.1.2.1.4.2"`), a symbolic name (e.g. `"system.sysUpTime.0"`), a syntax denoting which type the variable has, an access permission (e.g. read-only, not accessible etc.) and a status indicating the requirement on the presence of the variable (see appendix B.3).
The SNMP metadata in the metadata repository will be needed first of all by adapters which need it to perform type-checking and conversion between GOM's generic values and SNMP variables, but it can also be used by client applications using the *read* interface of the MR to navigate the electronic MIB specifications. Moreover, parsers make use of MR's *write* interface to add new MIB specifications or to update existing ones.

4.6.1.3 Summary

It was shown that it is possible to integrate a non object-oriented model such as SNMP into the generic framework of GOM.
GOM's *instance model* allows to manipulate variables in an SNMP agent, thus shielding the clients from the more complex details of communication using the SNMP protocol by providing an abstract higher-level layer on top of SNMP. This abstract layer allows to comprise multiple primitive SNMP requests into single logical operations. This is for example the case with SNMP's `GET-NEXT` request to traverse a MIB. GOM's instance model provides a single operation to retrieve all variables of a MIB, or only all variables of a conceptual SNMP table. Also, adapters take care of the communication with an SNMP agent which – being based on UDP – is connectionless and requires some resynchronization and reassembly mechanisms for multiple response which may arrive out of sequence. (See section 4.4.5 for a discussion of these issues.)
GOM's *meta model* allows to integrate all kinds of metadata due to its flexibility. The latter makes it relatively straightforward to represent SNMP MIB specifications as shown in the SNMP layout. The SNMP layout describes the structure of the specific model of SNMP metadata within the flexible meta model. It is described in appendix B.3.
The proposed approach for integration of SNMP is merely an *example*; other more elegant schemes are conceivable. The point, however, is not whether the proposed integration model is adequate, but that GOM's generic and flexible model gives developers considerable freedom to choose how to integrate a new model.

4.6.2 Reconciling Idiosyncrasies of Different Object Models

The purpose of this section is to show how characteristics (idiosyncrasies) of various target systems (in this case; SNMP, CMIP and CORBA) can be taken into account, *without extending the model proposed by GOM.*
In this paper, idiosyncrasies are defined as being features of a model outside the intersection of the features of all target models. The intersection of the features of those models results in a set of features common to all target models. Any feature not included in this set is considered idiosyncratic.

Of course, basing the set of common features in GOM on only three target models may overly constrain the number of elements considered to be common in the resulting set. The author acknowledges that such a *feature set* may grow with the inclusion of further target models. The goal was to create a feature set that allows to integrate all three target models without, however, the addition of features found only in one target model.
The general philosophy followed in coping with integration of idiosyncrasies is adopted from CORBA, which adds new functionality to its architecture in the form of *CORBA services* [COS95]. These do not require modifications to the core architecture of CORBA, but are defined *using* elements of CORBA's architecture (e.g. through definition of additional CORBA interfaces). Therefore we should try to integrate idiosyncrasies using mechanisms similar to CORBA's services and/or, as GOM's model is based on CORBA, make use of CORBA's services whenever possible. This should alleviate the need to modify GOM's model in most cases. Thus a feature of a target model will be integrated according to the following scheme:

1. Is there a CORBA service that covers the functionality needed by the OSI feature ? If so, it will be used.

2. If not, can we use a related CORBA service and extend it to integrate the feature ? If so, subclass the existing service (i.e. some of its CORBA interfaces) and use the new service.

3. If there is no related CORBA service, create a new one that covers the functionality offered by the feature.

All of these three approaches require no modification of GOM because they are implemented as CORBA services consisting of CORBA interfaces that can already be handled by GOM (through the CORBA adapter).

In the following sections some selected idiosyncrasies of the OSI model (CMIP) will be examined and a possible integration scheme will be described for each in turn.

4.6.2.1 OSI Naming Tree

OSI managed objects are arranged within an agent in a naming tree form as discussed in 2.2.4.
Each object has a relative distinguished name (RDN) which has to be different from the names given to its sibling objects. The concatenation of all RDNs from the root down to the object yields the distinguished name (DN) with which the managed object can be uniquely identified within the naming tree. Thus, every managed object that is ever created will be part of a naming tree.
Management applications may need to navigate the naming tree, e.g. to display all objects of an agent's MIT (management information tree) in a topology application. Therefore a mechanism has to be provided that allows clients to navigate the OSI naming tree.
Keeping in mind that the proposed model is based on CORBA, we should make as much use as possible of already existing services of CORBA. In this case, the *naming service* [COS95] could be used. However, managed objects in an agent are not registered with

a naming service automatically, but a scheme has to be found that will register newly created managed objects, deregister ones that have been deleted and – generally speaking – keep the naming service's contents synchronized with the OSI naming tree.
Obviously, the events of interest to us here are creation and deletion of managed objects.

Creation and Deletion of Managed Objects Managed objects can be created or deleted in two ways: either by the management client using operation `Factory::Create` or `GenObj::Delete` respectively, or by another management entity that has access (and proper authorization) to the OSI naming tree.
The first case is easy: any time a managed object is successfully created/deleted, it will be registered/deregistered by the CMIP adapter with the CORBA naming service using its distinguished name.
In the second case, OSI notifications can be used. By creating an *event forwarding discriminator* (EFD) managed object in an OSI agent that sends out event reports about creation and deletion of managed objects[58], we are able to update the contents of the naming service accordingly. This synchronization task could be performed by a UNIX *synchronization daemon* [Ste90b] (shown in fig. 4.41) that creates EFDs in all agents whose AE-titles are listed in a configuration file.

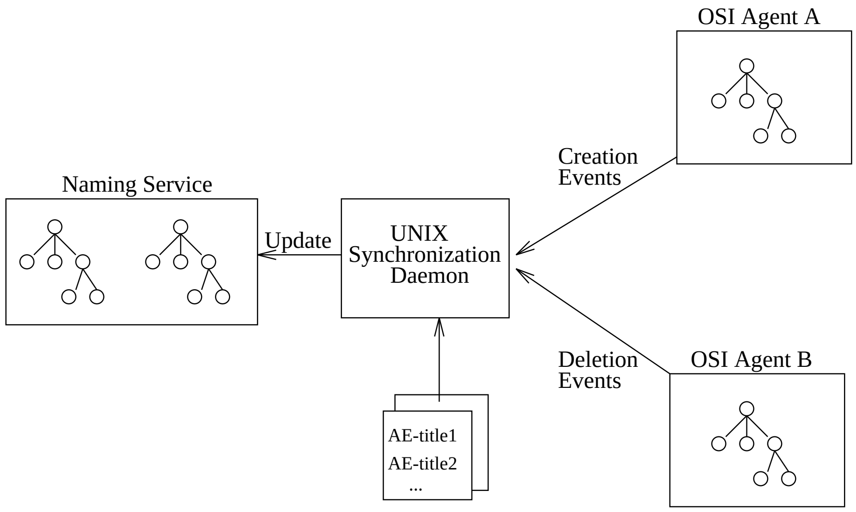

Figure 4.41: UNIX synchronization daemon

The list of agents to be monitored for creation- and deletion events could be modified at runtime by changing the configuration file and sending a signal (e.g. SIGHUP) to the daemon to re-read its configuration file (and accordingly create or delete EFDs). A creation event received by the daemon would result in the creation of a new entry in the CORBA naming service and a deletion would unregister this entry again. Using this scheme, the naming service would always be in sync with the OSI naming tree.
Note that if both schemes described above are used simultaneously, when an adapter creates or deletes an object, the naming service will be accessed twice, first by the adapter

[58]Note that emission of notifications upon creation and deletion of managed objects in an agent is optional.

when it creates the object and second by the adapter when it receives the event from the agent. This can be avoided by using one of three alternatives:

1. The adapter registers and deregisters all objects with the naming service, no notifications are used. This has the advantage that no daemon is required, but raises the problem of inconsistencies between the naming service and the OSI naming tree because creations and deletions by other management entities are not 'seen'. This problem can be solved in part by using the *synchronization* mechanism of the *proxy agent* proposed in 4.6.2.2.

2. Only notifications are used. The adapter does not register/deregister an object when it creates/deletes it. This has the advantage that naming service and naming tree are always synchronized, but adds the administrative task of managing the daemon. Also, the daemon may crash and during its downtime not be able to receive events, or it may not monitor all relevant agents for notifications due to configuration file inconsistencies. In this case, synchronization can also be achieved using the *proxy agent's synchronization mechanism* (see 4.6.2.2).

3. Both schemes are used. When an adapter creates/deletes an object, it registers/deregisters the object with the naming service. The daemon always checks with the naming service whether an object with the given name already exists. If this is not the case (object was not created/deleted by adapter), it will register/deregister the object. Otherwise, the daemon will not do anything. This scheme duplicates certain tasks, but results in a higher probability that naming service and naming will be synchronized.

Mapping Distinguished Names to the Naming Service A distinguished name would be mapped to a `Name` type in the naming service [COS95] which is a list of structs, each containing an `id` and `kind` field which are both string types. In this proposed mapping the `kind` field contains the attribute name of the RDN, and the `id` field the attribute value (as a string).[59]
Clients may now access the naming service to retrieve a proxy object for a managed object, given its distinguished name. Also, as the naming service preserves the hierarchy of OSI naming trees, it can be used to navigate through the naming tree and access its managed objects via the corresponding proxies registered in the naming service.

4.6.2.2 Discovery of Managed Objects in an Agent

In some cases the above proposal for synchronizing the naming service will not work. This is for example the case when an agent does not support notifications.[60]

[59] The form of the string is taken from the string-syntax proposed in [GMR94], but other schemes are also conceivable.

[60] OSI agents are not required to support the entire functionality. The feature set supported by an agent is given in a *functional unit* that can be negotiated between manager and agent when establishing an association [Bla92, p. 158].

A second reason might be that an agent already exists. Notifications for creation of new objects may be received, but there is no way to discover the managed objects already created in the agent.
A third reason is that in some situations a management application does not need to constantly keep the naming service in sync with the naming tree, but only occasionally needs to retrieve information from the naming tree. In such cases, it would probably be better simply to *poll* an agent for its objects or to *perform selective synchronization* for a part of the naming tree, thus avoiding the overhead of constantly keeping naming service and OSI naming tree synchronized.
CORBA does not currently know about the idiosyncrasies of OSI network management. Therefore no service exists that can be taken or extended to cover OSI functionality such as (1) retrieving managed objects from a naming tree, (2) discovering the roots of an OSI agent, (3) discovering the children of a specific object, (4) performing scoping and filtering and (5) handling multiple replies associated with (for example) scoping and filtering.
Therefore we have to define this additional functionality in a new service and make it available to clients as part of the entire system.
The approach taken was to define this new service using CORBA's object model (i.e., using CORBA interfaces) and provide the corresponding functionality in form of a CORBA implementation. The advantage of this approach is that clients can manipulate the service's components through GOM *like any other CORBA objects*. A clients sees no difference between managing its own CORBA interfaces or any prepackaged ones supplied by GOM. Instances of the new service are created as usual using operation `Factory::Create`.
OSI functionality mentioned in points 1–3 above are covered in this section, points 4 and 5 in the next two sections.

To this end, a new IDL service interface called `ProxyAgent` is introduced. It is a local proxy for a remote CMIP agent and offers operations to synchronize the CORBA naming service with a specific agent's naming tree and to retrieve a number of managed objects given a distinguished name. It is shown in fig. 4.42 and appendix A.5.
A proxy agent is essentially a placeholder for a real OSI agent and allows to perform operations on the basis of the entire agent rather than single managed objects.
It allows to manually start synchronization of a part of the naming tree with the naming service and to discover managed objects of an OSI agent.
Attribute `agent_address` must contain a valid address of an OSI agent, e.g. in the form of an AE-title. This address is not checked for validity when set, but only when the real agent is accessed. It can therefore be modified by clients to manipulate different agents.
Operation `SyncNamingService` accepts a *naming context* of a naming service [COS95] and a distinguished name (DN). It will use the address of an agent as given in `agent_address` to register with the naming service all children below (and including) the managed object with the indicated name. Parameter `level` determines how deep into the naming tree the search will be performed. A value of `-1` means that all managed objects will be registered recursively. If the distinguished name argument is NULL, all managed objects within the agent will be registered with the naming service.
Operation `DiscoverManagedObjects` allows a client to retrieve certain managed objects in an OSI agent. Parameter `distinguished_name` specifies the instance under which all

```
typedef sequence<GenObj> GenObjList;
typedef sequence<Val>    ValList;

interface ProxyAgent {
    attribute string agent_address; // AE-title

    boolean          SyncNamingService(in CosNaming::NamingContext nc,
                                       in string distinguished_name,
                                       in short level) raises(GenEx);
    GenObjList       DiscoverManagedObjects(in string distinguished_name,
                                            in short level) raises(GenEx);
};
```

Figure 4.42: Interface `ProxyAgent`

managed objects are to be retrieved, and `level` is used to restrict the number of objects returned (see above). This operation allows, for example, to retrieve all children of a certain managed object (by setting `level` to `1`).

If `distinguished_name` is NULL, this means that the *root objects* of an agent are to be retrieved. These are the topmost objects in an agent and constitute the 'entry points' to the agent's objects. Being able to retrieve the roots is important when the names of the roots are not known, allowing to retrieve all managed objects of an agent without knowing the names of the starting points. To achieve this goal, the operation could, for example, make use of the OSI *shared management knowledge* function as defined in X.750 [X7593] (see also 4.3.3.1).

4.6.2.3 Scoping and Filtering in OSI

Scoping and filtering allows managers to select a subset of the managed objects of an agent as target of a request. It is for example possible to perform an `M-GET` request for attribute `administrativeState` of all managed objects whose class is `eventForwardingDiscriminator`. Using `M-SET` with the same target set, it is possible to modify the `administrativeState` attribute of all managed objects in one request. Scoping and filtering is possible with the CMIP requests `M-GET`, `M-SET`, `M-ACTION` and `M-DELETE` (cf. [Bla92, ch. 6]).

There are various ways to add OSI scoping and filtering functionality to GOM. The approach chosen here is *not* to add parameters for scoping and filtering to normal operations such as `GenObj::Get` or `GenObj::Delete`, which would result in a mapping that is very close to the CMIP requests.

Instead it was chosen to regard scoping and filtering as a way to select a number of managed objects on which an operation can then be invoked.

Therefore the concept of a *group object* [Maf95, Bir96] is introduced. A group object is a container for a number of *group members*. A request sent to it will be sent transparently to all members of the group. Replies received can be handled either in a transparent fashion (the first reply is returned to the caller), or the request can collect all replies in a list and return the list to the client. The IDL interface of the group object is shown in fig. 4.43 (appendix A.5).

```
interface GroupIterator;

interface GenGroupObj : GenObj {
    void          SetAgentAddress(in string agent_address);
    void          SetScope(in short scope);
    void          SetFilter(in Val filter); // e.g. GenObj or Str

    void          PerformSelection() raises(GenEx);
    void          ClearResultSet();

    ValList       GroupGet(in string attrname) raises(GenEx);
    void          GroupSet(in string attrname, in Val new_val)
                    raises(GenEx);
    ValList       GroupExecute(in string opname, in Arglist args)
                    raises(GenEx);
    GroupIterator GetResultIterator();

    boolean       Add(in GenObj    new_obj);
    boolean       Remove(in GenObj old_obj);
    long          Size();

    // Operations Get and Execute are overridden to use the first result
    // that is returned (failure safety)
};
```

Figure 4.43: Interface `GenGroupObj`

Using this approach, a client has to perform the following steps for scoping and filtering:

1. Create an instance of `GenGroupObj`.

2. Set the address of the target agent (`SetAgentAddress`).

3. Set scope and filter (`SetScope`, `SetFilter`). A filter can be a `Str` instance obeying a certain syntax, which will be evaluated by the group object, or it could be a GOM value representing an ASN.1 `CMISFilter`.

4. Select the target set of managed objects (`PerformSelection`). The group object will maintain a list of all proxy objects that make up the result set until `ClearResultSet` is called.

5. Invoke an operation against the target set, e.g. `GroupGet`.
6. Perform some work with the result (in this case, a list of values).

A client may also access the proxy objects of the result set using an *iterator object*. An iterator allows to browse through the result set and perform some work with each of the proxies in turn. Operations `GetN` and `SetN` can be invoked, for example, on each object to obtain multiple values at once.[61]
The group object may not only be used for scoping and filtering, but it can contain any sort of proxy objects (SNMP, CMIP and CORBA) upon which to call the same operation. The restriction is that all members in a particular group object have to be able to 'answer' the same message sent to them (in Smalltalk parlance). This means that members do not even have to be of the same class (or subclass), or of the same object model, to be included in a group, but they just have to contain the same operation, that is, the same operation name and number and types of parameters.
Operations `Add` and `Remove` allow to manipulate the group. To make a group appear like a single object, all operations of the superclass that return a value (such as `Get` and `Execute`) are overridden[62] in the subclass and return only the first response received by *any* member of the group. This allows for failure resilience and high availability of the group object.[63] Of course, a requirement for such a service is that all members of the group be actually of the same class.

4.6.2.4 Multiple Replies in OSI

`M-ACTION` requests in OSI network management may return a number of responses rather than a single one as result. In static approaches such as XoJIDM (section 3.1.2), a client must include the signature of an operation (in the generated code) at compile time, therefore an operation *has* to return the value exactly as defined in the operation's signature. The approach taken by XoJIDM is to include only one return value in an operation's signature, and – if multiple replies[64] are received (which will not happen in most cases) – to raise an exception which has to be handled by the client, e.g. by iterating over the result set. The disadvantage of this method is that, since any OSI ACTION can possibly return multiple results, regardless of its signature, which may return only a single value, clients always have to wrap code that invokes an operation with exception handling code, which is quite awkward and increases code size.
The approach followed by GOM is very simple; as operations always return an instance of `Val`, multiple responses are returned as an instance of `Sequence` (cf. 4.2.2). The client may now check whether the result is a `Sequence` or a different value using operation `GetKind` of the returned value.

[61] These operations were not included in `GenGroupObject` for complexity reasons.
[62] Note that overriding is not yet permitted as of CORBA 2. However, a number of CORBA vendors have devised (yet) proprietary schemes to provide overriding and it is believed that this will be available in the CORBA standard eventually.
[63] Many details of group communication – such as the questions dealing with state transfer to a new member – have been omitted here. See [Maf95] and [Bir96] for a detailed discussion of fault-tolerant group communication mechanisms.
[64] Called *linked replies* in OSI parlance.

4.6.2.5 OSI Attribute Groups

Attribute groups are defined in [GDM92, 8.8]. An attribute group consists of a number of attributes under a single name (OID). The attributes that make up the group may be defined to be *fixed* in the GDMO specification, which means that no member may be removed or added at runtime, or they may be *extensible*. An extensible attribute group may receive additional members at runtime, or members may be removed from it.

When an attribute group is retrieved, all its member attributes will be returned in the form of a `Struct`. The keys of the struct will be the names of the attribute members and the corresponding values of the actual values of the attributes (subclasses of `Val`).

When a `GenObj::GetN` operation is used to retrieve a number of attributes, and one of the arguments in the dictionary refers to an attribute group, then its corresponding value will contain an instance of `Struct` upon return of the operation.

4.6.2.6 Recursive Types

Recursive types are data types that refer to themselves. CORBA does not allow *direct* recursive types (although syntactically possible) such as the one shown in fig. 4.44 (a).

```
struct foo {                  struct foo {
    long  value;                  long value;
    foo   recursive_ref;          sequence<foo> recursive_ref;
};                            };

(a)                           (b)

CMISFilter ::= CHOICE {
                   item [8] FilterItem,
                   and  [9] IMPLICIT SET OF CMISFilter,
                   or  [10] IMPLICIT SET OF CMISFilter,
                   not [11] CMISFilter
                 }
(c)
```

Figure 4.44: Recursive type definitions in GOM.

This type has to be transformed as shown in fig. 4.44 (b) according to [OMG95, 3.8.2].

In the case of ASN.1, recursive types are allowed ([ASN90, 5.5]), e.g. the definition shown in fig. 4.44 (c) is legal.

We therefore have to provide a way to represent recursive types both in the meta model and in the instance model, i.e. it should be possible to create meta information about recursive types using elements of the generic meta model, and to create generic recursive values. The two cases are described below.

Recursive Types in the Meta Model A generic example that allows to represent all sorts of recursive types in the meta model is shown in fig. 4.45.

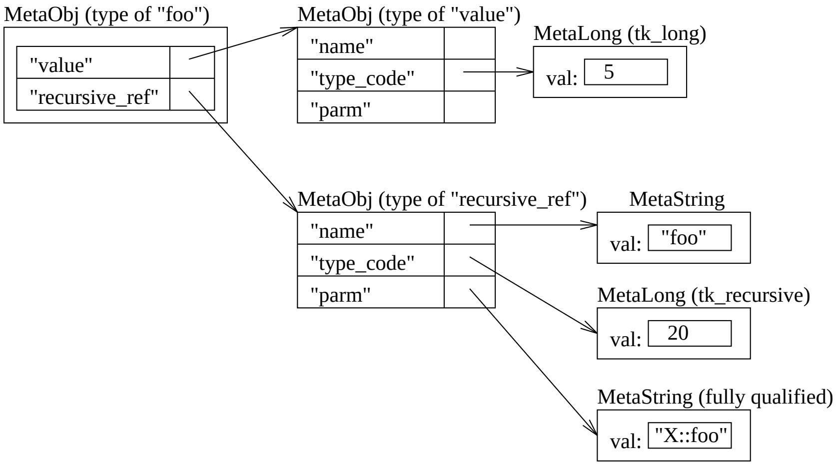

Figure 4.45: Example of a recursive type definition in the meta model

It shows how the (illegal) IDL example shown in fig. 4.44 (a) would be represented.[65] Struct `foo` is represented by an instance of `MetaObj` which describes its type. In the example, struct member `recursive_ref` recursively points to the enclosing struct. This relation is modeled by an instance of `MetaObj` which describes the type of the member variable. Property `"name"` points to the ASCII name of the struct and `"type_code"` to an instance of `MetaLong`, describing the type code constant (`20` $\rightarrow$ `tk_recursive`). Finally, property `"parm"` contains the fully qualified name of the recursive type. This allows to retrieve the type definition if needed, using the name of the type as lookup key.

An alternative for representing the recursive structure of the type would be to have property `"parm"` recursively refer back to the initial instance of `MetaObj` which describes type `foo` so that the type and `"parm"` point to the same `MetaObj` instance. This scheme has the advantage that no lookup of types using the (recursive) type name has to be performed. If this alternative is preferred, however, care has to be taken to avoid problems when for example dumping the `MetaObj` to a stream or when reading it back. Possible schemes for marshaling and unmarshaling structures containing circular references are described in [Gro93, Cra93, Shi94].

Of course, the alternative to be adopted has to be decided by the author of a layout and described in the layout definition. The flexible and generic structure of the meta model allows for a number of alternative approaches to handle recursion.

Recursive Values in the Instance Model The recursive IDL type shown in fig. 4.44 (a) would be represented as a GOM `Struct` value. The flexible nature of GOM does not require a `Struct` to have a fixed set of members. Rather, members may be added or removed at runtime using operations `Add` and `Remove`. This allows to construct

[65]Although not (yet) legal in CORBA, the example demonstrates how typical recursive types are usually defined. The same approach would be taken for recursive ASN.1 types such as the one shown in fig. 4.44 (c).

an instance of `Struct` that does not yet contain any members. Then, for example, the member variable `value` could be added by creating an instance of `Long` and adding it to the struct. Adding the recursive member variable `recursive_ref` is the same: an (initially empty) instance of `Struct` is created and added to the initial struct under member name `"recursive_ref"`. Then the members of this new struct may be added and so forth.
GOM's ability to construct values at runtime is an advantage compared to static mapping approaches in the case of recursive types. It was shown that both meta- and instance-data for recursive types can be modeled using GOM.

4.6.3 The Proxy Principle

A proxy ([Sha86] [GHJV95]) is a placeholder for a different – usually remote – target object. The proxy is generic in that it can represent instances from various target models. Proxies 'know' which targets they represent, whereas target instances do not know about their proxies (or proxies in general).
Operations invoked on a proxy will be forwarded to its corresponding target. Therefore a proxy needs to be able to contact the target. This binding is done when the target and proxy instances are created. Usually, the proxy will store a *handle* to the target which enables it to find the target whenever needed.
The handle may be an object reference[66] as in the case of CORBA, an AE title and a distinguished name in CMIP, or a host address, a port and an OID in SNMP.
A CORBA object reference uniquely identifies an object. When the object is deleted, its object reference will not be reused. Any attempt to dereference it will yield an error.
OSI distinguished names may be reused and may thus refer to different managed object over time. It is therefore theoretically possible that the managed object target instance to which a proxy refers may be deleted and a new managed object with the same distinguished name, but with a different class, may be created. The effect would be that, when the next operation is invoked on the proxy, it would most probably fail. Genilloud [Gen96, 4.5.4.1] describes a scheme how this problem could be solved using OSI notifications.
SNMP variables are rather static; there is no dynamic creation and deletion of variables, therefore the problem described above will not occur.
Proxies may cache attributes locally and return cached data according to a caching policy (described in 4.1.1). This is useful in cases of communication problems because attributes can be returned even if the target is not reachable, or for performance reasons, because no traffic is generated between proxy and target locations.
Proxies and targets may have different lifetimes. There are mainly three cases to be considered:

1. Proxy and target have the same lifetime.
2. Proxy is deleted before target.
3. Proxy is deleted after target.

[66] An *inter-operable object reference* (IOR) may be used to enable access to instances within *different* ORBs. An IOR would probably be stored when using IIOP in the CORBA adapter (see 4.4.3).

In the first case, a proxy will usually be created together with its target and when the proxy is deleted, it will delete its target as well. However, multiple proxies may refer to the same target. This may be the case when proxies to existing targets are created, e.g. by consulting the naming service (see 4.6.2.1, 4.6.2.2). In this case, we cannot simply delete the target when a proxy is deleted. *Reference counting* schemes, however, allow to maintain a reference count for a target object that is incremented with each new proxy attached to the target and decremented when a proxy is deleted. When the reference count is zero, the target object will be deleted. For a discussion of reference counted proxies see [Cop92, GHJV95].
The second case is the most frequent in GOM; a proxy lives independently from its target. That means that a proxy may attach to an existing target by accessing the naming service, and it may be deleted without deleting the target object.[67] It also means that multiple proxies may refer to the same target. This case is unproblematic as long as the target instance still exists.
The third case is the most problematic one: a target instance may be deleted by some third party, e.g. by another management application, or by another proxy. This results in a *dangling reference* of the proxy to the target. When the proxy tries to access the target, an error will occur.
This problem could be circumvented by complex schemes where adapters maintain a list of proxies created by them and, whenever deletion notifications are received by target entities, the corresponding proxy's target reference is invalidated so that access to the target will not even be attempted and an error will be returned immediately.
However, because of the complexity that this proposal would bring with it and the infrequent occurrence of this problem, it was ignored. Therefore, whenever an operation is invoked on a proxy with a dangling reference, a normal error will be returned to the caller.

Other problems involve crashed servers or inaccessibility of agents because of network partitions. Here, several possibilities exist to cope with these communication problems.
The first one is to not hide this fact to clients and return errors or throw exceptions respectively. This allows clients to cope with these errors and take corrective actions.
An alternative is to store operations to be sent to targets in a queue, and, as soon as communication is established again, to send these messages to the target instance.[68] One possible solution for retrieval of attributes would be to return them from the cache (if a cache is used) and for setting of attributes to use the queue mechanism. Operations, however, that are synchronous and need to return a result immediately will fail. Asynchronous operations (cf. 4.6.5) may be queued similar to attribute access, and be invoked only when communication is re-established.
The disadvantage of such solutions is that a client never knows whether communication with the target has taken place, or whether requests are queued. However, for certain types of application, such transparency may be desirable.

Adapter implementors may choose various strategies for coping with network partitions

[67] To delete the corresponding target, operation `Delete` of the proxy can be used.
[68] Commercial solutions exist for this problem, e.g. MQSeries [MQS95].

or communication problems, according to the target system (and the possibilities it offers) at hand.

4.6.4 Persistence

Proxy instances used in management applications represent instances in various target systems such as SNMP/CMIP agents or CORBA servers. These usually have a longer lifetime than the management applications accessing them. An ideal assumption would be that target systems in which target instances are located live 'forever', i.e. they are constantly up and running.
However, management applications are shut down and restarted and therefore lose their information about target instances (proxies). A solution to prevent this is to register instances of proxies with a naming service (see 4.6.2.1) before the management application is shut down and to retrieve them at a later time, when the manager is restarted again.
Another possibility is to make proxy instances persistent by for example saving their state to secondary storage. Whenever an application is terminated, it may save some of its proxy instances to a file and reload them when restarting the next time.
A possible persistence implementation could make use of CORBA's persistence service [COS95] which provides generic services to make CORBA instances persistent, ranging from simple flat file storage to complex database management system capability.
An alternative is to use GOM's simple externalization operations `Dump` and `Read` defined in interface `GomElement` (cf. fig. 4.4 and appendix A.1). `GomElement` is the 'mother' of all GOM interfaces, therefore all interfaces are able to be dumped to a stream and be reconstructed from it.
This characteristic allows proxy instances to be saved to a file and later be re-read from it. Also, proxies may be dumped to a stream and sent to a remote (GOM-based) management application which can resurrect them and invoke operations as if it were created there, allowing for proxies to be exchanged between several management entities.

4.6.5 Execution Model

Operations invoked on proxy objects are always synchronous in GOM. Therefore a caller is blocked until the operation returns. However, in certain cases it might be useful to have asynchronous execution semantics in which operation calls return immediately and results can be fetched at a later time.
Below a proposal is presented which extends GOM to take into account asynchronous execution.
There are three main kinds of asynchronous execution [Maf95, 3.4]:

Oneway-asynchronous Operations invoked by callers which return no result and which do not guarantee delivery of operation to receiver. An example of this execution model are `oneway` operations in CORBA; these must not have a return value and are delivered on a best-effort basis.

Adapters may dispatch oneway operations to their targets and then return without waiting for results.

Asynchronous with callback Asynchronous operations which return results can be divided into two sorts: those for which the caller has to collect the results (deferred-asynchronous) and those that invoke a *callback*[69] provided by the caller to signal execution termination (asynchronous with callback).

The semantics of callbacks can be modeled using object references containing certain operations that are *called back* by entities in the server role. A caller provides such an object reference as argument to the operation invocation. The operation returns immediately and – when the results are ready – invokes an operation on the object reference. This allows callers to be notified immediately when the operation has returned.

Deferred-asynchronous These asynchronous operations also return results, but – unlike asynchronous operations with callbacks which invoke a callback when the results are ready – callers have to *poll* for results. Deferred-asynchronous operations usually require a token from the caller on which caller and callee can synchronize. An example for such tokens are *promises* [LS88]. These are part of an asynchronous-deferred operation's parameter list and are used by the callee to store the results and signal that the operation has finished, and by the caller to determine whether the operation has terminated and to fetch the result(s).

An example of how deferred-asynchronous operations can be provided by GOM is given below.

Every operation on proxy instances can theoretically be executed asynchronously. Of course some operations will benefit more from this sort of execution than others for which it would make less sense. As an example of a deferred-asynchronous operation using a promise, consider operation `GenObj::Execute`. Because CORBA currently does not allow overloading of operations [Str91], an additional operation is provided as shown in fig. 4.46.
Compared to its corresponding synchronous version, `ExecuteAsync` returns `void` and accepts an additional parameter `Promise` which is a CORBA object. A caller can use the promise to query the entity in the server role whether the results are ready (`Done`), wait until the operation has terminated (`Wait`) and retrieve the results (`GetResult`). The latter will block until the result is available and then return it.
As promises will possibly be used concurrently by callers and callee, its operations have to be re-entrant.
An example of how to use the operation is given in fig. 4.47.
First, a promise object is created by the caller. Then a deferred-asynchronous operation is invoked using `ExecuteAsync`. Note that it will return immediately, without blocking the caller which may perform other tasks while the operation is performed. A caller may at any time query the promise object for operation termination using `Done` which returns true when the operation has terminated. To retrieve the result, `Promise::GetResult` is finally called, which blocks until a result is available, and the result will then be returned to the caller. If an operation does not return any value, `Wait` can be called on the promise object. This operation will return as soon as the operation has finished.

[69] Usually a pointer to a function in C/C++.

```
interface Promise {
    Val      GetResult(); // blocks until result is available
    boolean  Done();
    void     Wait();
};

interface GenObj {
    ...
    void ExecuteAsync(in string opname, in Arglist args,
                      in Promise p);
    ...
};
```

Figure 4.46: Adding deferred-asynchronous execution to `GenObj`.

```
Promise_ptr promise= // create promise object
Long_var    result;
Arglist_ptr args=  // create argument list
GenObj_var  proxy= // retrieve proxy
proxy->ExecuteAsync("Foo", args, promise);
...
// do something else
if(promise->Done())
  result=dynamic_cast<Long_var>promise->GetResult();
```

Figure 4.47: Example of use of `ExecuteAsync`.

Deferred-asynchronous operations are useful in situations where a caller wants to start a number of operations in short succession and retrieve their results at a later time.

4.6.6 Enabling OSI Managers to Access GOM

The topic of this work is to provide CORBA client applications access to instances of other object models such as CMIP or SNMP by means of the Generic Object Model. This explicitly excludes managers of domains different from CORBA from using GOM.

Below the discussion briefly touches on how this could nevertheless be achieved and gives an example of how OSI managers could be enabled to access the Generic Object Model (and thus instances of other models as well). A similar mechanism could be devised for SNMP managers.

The architecture is shown in fig. 4.48.

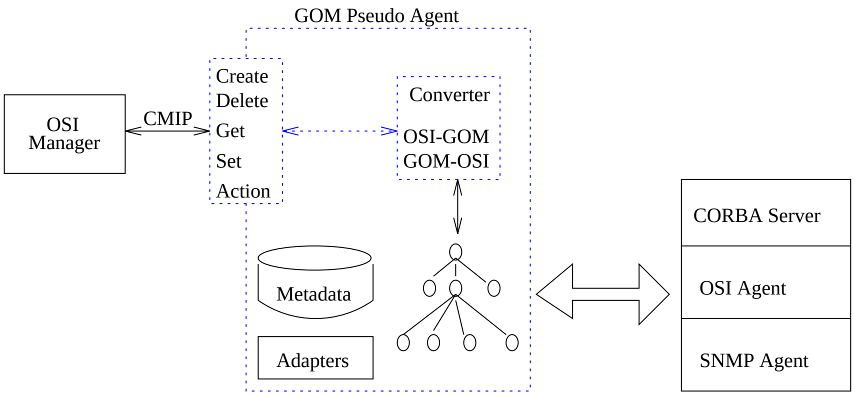

Figure 4.48: Access from OSI managers to GOM

The approach consists of a *GOM pseudo agent*, mimicking an OSI agent.[70] CMIP requests received are forwarded to a converter which has to map ASN.1 values (e.g. XOM data structures) to GOM values and back. A naming tree, consisting of GOM objects, keeps track of the associations between OSI names and GOM object references. As GOM objects can represent underlying instances of any object model, this scheme allows OSI managers to manage instances of all models for which adapters are available (CORBA, CMIP, SNMP).

Unlike other approaches that deal with OSI managers managing CORBA instances ([Spe97, Int95, Mas97, BGG94]) which require a pseudo agent to know all the CORBA interfaces it can handle at compile-time, this scheme allows any number of CORBA interfaces to be managed from OSI without modification of the pseudo agent. As a matter of fact there may be only one pseudo agent, or several pseudo agents may share the work without concern for which interfaces each agent is able to handle.

4.7 Summary

Object-oriented distributed programming techniques such as CORBA are becoming increasingly popular in systems and network management. However, older paradigms to manage networks such as CMIP and SNMP are still predominantly used: SNMP being the de-facto management standard for devices in local area networks and CMIP being popular with carriers and telcos for wide-area network management.

It is the author's assumption that no single standard will be predominant in the years to come, but that there will probably be a coexistence of multiple models, resulting in a heterogeneous 'management mix'.

Integration tools that make this mix transparent and allow end-to-end management without regard for the details of the underlying models will therefore be of major importance. The CORBA standard seems to be a candidate for such an integrated model. It is gaining acceptance in building distributed systems, possibly because of its simple object model, its language neutrality and its distribution transparency which (almost) hides the difference between local- and remote method calls.

[70] Existing OSI agent toolkits can be augmented for this purpose.

It is therefore feasible to take CORBA as the common denominator for management and try to integrate existing models such as SNMP and CMIP.
Several approaches were presented in section 3 which take CORBA as common model and provide *bridges* or *gateways* to existing management systems. Most of them are characterized by static translation of a management model specification to the CORBA model (specification translation) and runtime conversion between the CORBA- and a specific target model (interaction translation).
The disadvantages of these static approaches have been described in 3.1.2.1 and 3.2. They comprise mainly the tight binding between client and server, which requires regeneration of the client on server modification, the potential mass of generated code bloating the client, and the inability or at least awkwardness of mapping certain SNMP/CMIP idiosyncrasies to CORBA.

The *Generic Object Model (GOM)* presented in this chapter proposes ways to eliminate some of these problems. It provides a uniform programming model based on CORBA and uses metadata rather than translation-generated code-inclusion to bridge to target management models. *Bridging code* for target models such as SNMP or CMIP is concentrated in adapters for easier maintainability. Idiosyncrasies of target models can be smoothly accommodated using CORBA services or by providing own services in the form of IDL interfaces. An important piece of the proposed model is its metadata repository which allows communication with target models by means of metadata lookup and dynamic request dispatching to the corresponding system. The metadata repository can also be used by clients for metadata queries and documentation purposes.

Benefits

Uniform Model The uniform programming model of GOM allows management applications to be based entirely on GOM/CORBA with minimal (or ideally no) knowledge of the target management model. This contributes to the homogeneity of management applications, which can use a single API for all target models rather than having to use a number of different APIs. Also, learning costs for other models can be reduced, which is important especially in the telecommunications market in which new services have to be introduced rapidly to be successful. A uniform model with bridges to older management models allows to build new GOM/CORBA-based applications with immediate integration of legacy managed entities. Because significant investment in the SNMP/CMIP models has been made in the past decade, such an approach protects this investment and allows at the same time the gradual replacement of SNMP/CMIP managed entities with CORBA-based ones.

Based on CORBA The fact that GOM is based on CORBA has a number of advantages: first of all, all elements of GOM are specified in OMG IDL, which allows to use any language for which a binding exists to manipulate (and implement !) them. This enables access to GOM from any existing (or yet to be devised) language for which a binding exists.
Second, all elements of GOM can be distributed in a network, resulting in better load-balancing and availability of management services. For example the metadata repository

might be replicated on multiple nodes, effectively forming a *process group* [Maf95, Bir96]. A request sent to the CORBA (group) object reference that represents the metadata repository might use the first response returned, which makes the MR reliable, highly available and efficient.
Moreover, management of wide-area networks is becoming increasingly decentralized because applications and services are distributed themselves. Management of distributed objects requires distribution of management as well.
Third, the CORBA services are available, which offer additional standardized functionality.

Convenience Bindings A language binding for GOM's IDL interfaces may not be compliant with the host language at hand, or it may not make use of the special features a host language offers because IDL has to be a common denominator of all languages and should not make use of any specific language's features. Therefore, *convenience bindings* have been proposed (4.2.4) which are layered on top of the generated language binding in a host language X and may be used by management applications written in language X. These make use of the special features available in a host language to adapt the generated bindings to the corresponding host language.

Dynamic Access The dynamic aspect of GOM eliminates the strong dependency of clients on servers as is the case in static translation approaches where clients include classes generated as result of target model translation. Clients do not need to include knowledge of all target models' classes at compile-time; instead GOM allows clients to discover them at runtime. In addition, client applications benefit from dynamic class loading in that they are typically smaller than compile-time based approaches.

It was shown that GOM can be used as an alternative to the static approaches if flexibility, client-server independence and code size reduction are important. However, in cases where the number of classes of a target model is finite (and small) and where the flexibility offered by GOM would not have any benefits, it may be desirable to use one of the static approaches to avoid runtime type-checking and -conversions. Static and dynamic approaches complement each other and the decision concerning which one to employ should depend on the requirements of the management application at hand.
In the next chapter, an overview of two applications that make use of and benefit from the flexibility and uniformity of the model proposed by GOM will be presented.

Chapter 5

Applicability

As stated in the motivation section (1.1), the major goals of this thesis are (1) the creation of a uniform programming model, (2) flexibility of binding between client and servers and (3) reduction of client size.
Two applications are presented in the next two sections that benefit from the achievement of these goals, validating that an approach such as the one proposed for GOM can profitably be used for certain types of applications.

5.1 GOMscript

GOM allows to create instances without requiring clients to have compiled-in class knowledge. Rather, the name of the class can be given as a string, e.g. to create an instance. GOM will then look up the name in the metadata repository and – if found – retrieve metadata about the class and 'assemble' a generic GOM instance using the metadata.
Operations are invoked the same way, namely by indicating the name of the operation and a list of generic arguments.
Moreover, attributes can be retrieved by giving their name as a string.
GOM's ability to perform these things at runtime without having compile-time knowledge about the classes to be handled is an ideal basis for the creation of interpreters. These process programs – either typed-in interactively or in the form of scripts – and execute them by evaluating all statements sequentially.
To validate GOM's flexibility and uniform programming model, an interpreter called *GOMscript*[1] has been written. It is based on GOM's instance model, that is, every value is represented by a GOM value or instance.
GOMscript allows to manipulate GOM instances interactively or through scripts. Therefore, classes of those target models for which an adapter is provided may be handled[2] using GOMscript.
GOMscript has a syntax similar to C++. It has simple values such as numbers, booleans, strings, and aggregate values such as structs, unions and lists. It has the usual control

[1] The language description can be found in appendix C and further details in [Ban96c, Ban96b, Ban96e].
[2] Any prospective object model to be integrated with GOM will automatically be manageable by GOMscript.

statements, e.g. for repetition (`for`, `while`) and conditional branching (`if`, `else`). The language is object-oriented in the sense that it has classes, (single) inheritance, polymorphism and encapsulation. It consists of a small core that can be extended through user-written *extensions* located in shared libraries.
A simple example of GOMscript code that flips the attribute value of a managed object in an OSI agent is shown in fig. 5.1.

```
name_srv= // retrieve initial reference to CORBA naming service
obj=name_srv.resolve("netId=TelcoNet;circuitID=(name IBM)");
if(a != NULL) {
    ad_state=obj.administrativeState;
    if(ad_state == "unlocked") {
        obj.administrativeState="locked";
    }
    else {
        obj.administrativeState="unlocked";
    }
}
```

Figure 5.1: GOMscript sample code

The example demonstrates how two different target object models (CORBA and CMIP) can be accessed at the same time.
First, a reference to a CORBA naming service is retrieved and assigned to variable `name_srv`. This is a CORBA instance, and normal operations can be invoked on it. In the sample code, operation `resolve` is called to retrieve a managed object (CMIP target model) given its distinguished name[3] which is then assigned to `obj`.[4]
Then, the value of attribute `administrativeState` is retrieved. Depending on the value, the opposite value is set in the retrieved managed object. Note that both attribute access (`obj.administrativeState`) and operation invocation (`name_srv.resolve`) have the same simple syntax, which eliminates the need for `Get`-, `Set`- and `Execute` operations.

GOMscript enables manipulation of instances of several target models. This allows for interactive exploration of existing target system classes by creating instances, setting and getting attribute values and invoking operations. Also, newly written classes can be tested against desired behavior immediately without having to write a client test program.
This validates the author's assumption that a flexible and highly dynamic API as provided by GOM supports the creation of interpreters which could not be built using compiled-in knowledge of the target system classes.
The uniform programming model of GOM enables an interpreter writer to base the interpreter on the same homogeneous API, rather than having to use a separate API for each

[3]In *string-syntax* form as defined in [GMR94].
[4]Note that both `name_srv` and `obj` internally point to a GOM *proxy instance*, which itself has a reference to the corresponding target instance in either the CORBA- or CMIP object model.

model. Also, if a new model is integrated, the interpreter does not have to be changed because the API remains the same.
There are other interpreters (e.g. *CorbaScript* [MGG96] or *TclDII* [Tcl95]) that allow to manipulate CORBA instances interactively. Both employ the Dynamic Invocation Interface (DII) to access CORBA instances. Unlike them, GOM is not tied to managing CORBA instances. Rather, *any* (object) model should be manageable if a suitable adapter is available. GOM capsules the DII in an adapter to access CORBA instances, but this fact is transparent to clients of GOM. Actually, a CORBA adapter could use IIOP rather than DII without the clients noting this. Note that both CorbaScript and TclDII could have been written using GOM as a more abstract layer for DII than directly accessing the DII. This would have enabled them to handle instances of other models as well. Also, they would not have to be modified to handle new object models, but could just have made use of a new adapter.

5.2 Roaming Agents

A GOMscript interpreter process can be started in a special mode in which it listens on a socket, waiting for code to be sent by clients. For each connection established, a new process is spawned and the received code is interpreted. The general architecture is shown in fig. 5.2.

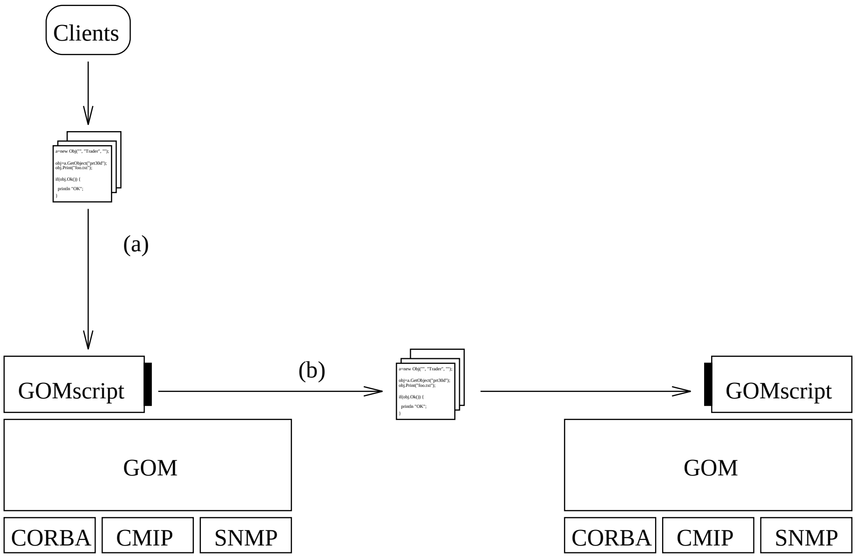

Figure 5.2: Roaming Code.

There are two ways of sending code to an interpreter process: (a) clients may open a TCP connection and send the code, or (b) a GOMscript `send` function[5] allows to specify from within GOMscript code the destination machine to which the currently running script should migrate. The latter case involves dumping the current state of the interpreter (i.e.

[5]Implemented as a GOMscript extension loaded at startup. This is a good example of how new functionality can be added to the interpreter without modifying its core.

all variables together with their values), establishing a TCP connection to the remote interpreter, sending the currently running script and its state, resurrecting the state at the destination and continuing interpretation at the new location. It is at the programmer's discretion whether the script on the old machine will continue its processing or terminate. This scheme allows to implement simple *roaming agents*. These are pieces of code with associated state that migrate from machine to machine in the network, performing certain tasks [ME96, Whi94], [Mue96, p. 299–424].

A roaming agent has two important characteristics: first, it is *stateful* because all state (i.e. all variable/value bindings) is carried along when migrating to a new location. Second, it is *active* in the sense that it can make its own decisions when and where to migrate using GOMscript's `send` primitive.

An important characteristic of GOMscript-based roaming agents is that they can handle CORBA, CMIP and SNMP instances. Actually, instances of any (object) model for which a suitable adapter exists can be handled immediately since GOMscript is based on GOM. Assuming that (physical or logical) resources such as switches, routers or customer databases will have to be accessed by roaming agents to perform their management tasks, and assuming that these resources will (at least partly) be represented by instances of models such as SNMP, CMIP or CORBA, it will be useful for roaming agents to have the capability to manipulate them.

A roaming agent can never know what types of instances representing resources will be available on a certain machine on which management chores have to be performed. It is for example possible that a roaming agent has the task of rebooting all print servers on a certain segment of the network that have been online for more than 200 hours.[6] Given a query to a *naming*- or *trading* service, the result may be a list of instances representing print servers, but of which the types (classes) are not known, i.e. not included at compile-time. The only knowledge might be that these instances offer an operation called `"Shutdown"` or `"Reboot"`. Being (indirectly) based on GOM, a GOMscript roaming agent can still handle the task by examining the metadata for each instance, retrieving the correct operation and dynamically invoking it. This can be done without having static compiled-in knowledge of the instances' classes, but by just having metadata about them in GOM.

In this example a roaming agent still has to know what the semantics of classes it encounters is and what the names of operations to invoke are. This is because instances that will be encountered are not *semantically attributed*, which means that we know nothing about the semantics of a hitherto unknown class, but only about its syntax (using metadata). Attaching semantics to classes would allow an agent to ask an instance what the semantics of an operation is and, based on the response, invoke a suitable operation. This approach, however, requires both roaming agents and instances to 'understand' a common language describing semantics.[7] Combining GOM's dynamic metadata-based manipulation capability and a facility for semantically attributing instances, the roaming agent

[6] Of course, roaming agents must have adequate privileges to perform tasks such as rebooting a machine. How these are acquired and verified is outside the scope of this work, for more information see [ME96, Whi94].

[7] To understand semantics, we first have to establish categories of concepts allowing us to agree on how to name concepts within the domain of discourse.

implementation would be yet more flexible in reasoning about unknown classes. While semantically attributed languages are still largely a research topic with many differing schemes being proposed, a standard has been evolving that proposes to merge various approaches [KQM95].

From the discussion above, it is clear that roaming agents have to be very flexible. They cannot possibly know the extent of classes they will encounter on their trips since the set of classes to handle will likely be non-finite. Therefore an approach that requires a roaming agent to have compiled-in knowledge of target classes such as the static ones proposed in section 3 cannot be employed here. GOM, however, allows to manage instances of classes that were not known when the roaming agent was written. Provided that metadata about these classes is available in the metadata repository, GOM allows to create instances of instances about which it has no compiled knowledge, get and set attributes and invoke operations.
Another advantage of not having to know all classes that will ever have to be handled (besides that this is not possible !) is the fact that clients (i.e. roaming agents) can be very small in terms of memory size since they do not have to include all generated classes as proposed by XoJIDM's approach (3.1.2). This allows them to be sent to different locations efficiently and quickly. Sending large agents around the network would involve delays because of their size.

The two applications presented in this section represent the type of applications for which GOM can profitably be used because of its flexibility and uniform programming model.
In the author's opinion, highly dynamic and rapidly moving areas – especially in the telecommunications domain in which the liberalization of the market in 1998 brings with it increased competition and thus pressure on telcos to introduce new services quickly – ask for highly dynamic and flexible systems. These must take into account heterogeneous network management worlds, be able to provide uninterrupted service and allow for rapid implementation of new services.
Dynamic management schemes such as the one presented in this thesis are a step in this direction.

Chapter 6

Conclusion and Outlook

Goals

The main objective of this thesis was to propose a *CORBA-based network management integration model* for SNMP, CMIP and CORBA. The motivation was the assumption that CORBA will gain major importance in network management, both on the managed- and management side. However, other management models have to be accommodated to be able to manage existing systems. Therefore we proposed that CORBA be taken as the common middleware layer for management and be extended to embrace other models with which managed entities are implemented, such as SNMP, CMIP and CORBA.[1] (Note that GOM focuses on the client side; as an interface to *invoke*, and *not implement* management functionality).

The target user for such a model is someone who uses CORBA and wants to access SNMP/CMIP objects without having to know SNMP/CMIP, and not necessarily someone who is already familiar with SNMP/CMIP. Therefore the idiosyncrasies of the various models (such as OSI scoping and filtering) need not be supported 1:1 in the common CORBA-based model, but rather their functionality should be *emulated* by a CORBA-conformant implementation.

Contribution

The contribution of this thesis consists of a proposal for a more dynamic and uniform interface for building CORBA-based management applications, the *Generic Object Model* (GOM). Moreover, an implementation in the form of a research prototype is provided as a feasibility study of the ideas presented in GOM.

The proposal uses well-known concepts such as metadata, a reified object model and adapters, and applies them in combination to the domain of network management in which approaches to CORBA-based management have hitherto predominantly been of a

[1] The author assumes that the distinction between managed- and management entities will fade over time as CORBA objects assume management tasks. Although CORBA is predominantly used to implement client applications, we envisage that management server- (or agent) functionality will increasingly be implemented using the CORBA model.

static nature.
The proposal consists of an object model that comprises an *instance* and *meta* model. *Convenience bindings* for the instance model facilitate the task of writing management applications. The *metadata repository* serves as central storage of metadata information. *Adapters* perform type-checking and convert requests between the generic- and specific management models. An *event handling* proposal supplements the generic management functionality required by management applications. Idiosyncrasies of different target models are integrated using *CORBAservices*.

The benefits gained from such an approach can be summarized as follows:

Uniform Programming Model Rather than having to use a number of heterogeneous management APIs, a management application requires only a single homogeneous API. However, instead of creating yet another API, the model is based on CORBA which is already established in the market. The reasons for using CORBA as common platform were as follows:

1. Language neutrality. Management clients and implementations may be written in different languages (if a language binding is available).
2. To support highly distributed managed entities, a manager has to be distributed as well.
3. The ever increasing size of resources, networks and information forces network management applications to be partitioned into smaller, distributed management 'chunks', resulting in better scalability.
4. CORBA services [COS95] allow for reuse of functionality.

Flexibility In network management there are usually a large number of managed entities, compared to only a few managers. The chance that one or more of the managed entities is modified is high. The flexible model based on metadata as proposed in this thesis avoids the need for management applications to be recompiled when one of the agents it manages is changed, but allows for uninterrupted operation. Also, management applications do not have to include managed classes at compile time, but can do so with the help of metadata at run time.

Small clients As client management applications do not have to include 'the world' at compile-time, their size will not be large. This is a benefit, considering that an application usually also has to include other libraries which increase its size, such as GUI or database functionality. Using the proposed model, clients only 'pay' for what they actually use.

Language Bindings

As GOM is based on reification (everything is an object), it fits best with languages that are reified as well, such as Smalltalk or Lisp [GR89, Ste90a]. With these languages,

GOM objects can be manipulated in a way that is natural to Smalltalk using convenience bindings, without having to resort to auxiliary methods such as `Get-`, `Set` or `Execute`.
With statically typed languages such as C++, GOM's model fits less nicely with the overall language philosophy because auxiliary methods have to be used (`Get`, `Set`, `Execute`), memory management of returned values (e.g. result of a `Get` method) has to be taken care of, returned values have to be narrowed to the actual class and arguments have to be assembled in an argument list object before operations are invoked. Convenience bindings may hide only a part of the difficulties of the model mismatch.
Therefore, with statically typed languages such as C++, GOM's intended use can be regarded primarily as a *value-adding toolkit* – to be used for example by interpreter writers – which is not directly accessed by client applications, but which serves as a foundation for other application-specific layers on top of it.

Security

Security is important nowadays, especially in the light of distribution, which introduces a number of security issues not present in non-distributed systems. However, given the scope of this thesis, security problems have not been tackled here. It is hoped that they can be integrated after the fact ...

Prototype

A prototype implementation of some of the concepts presented has been made. It currently comprises the generic object model, metadata repository and two adapters (for CORBA [SOM94] and CMIP [GMR94]).
On the basis of this prototype, a few test applications have been developed for feasibility study. These comprise (1) a CGI [CGI94] based topology application, which displays the managed objects within OSI agents and which can be accessed using any Web browser [TBLP92], (2) Java bindings [Sun95], which can be included by clients to manipulate GOM instances, (3) the GOMscript interpreter as described in 5.1, and (4) a simple roaming agent facility (5.2). See [Ban96g, Ban96b, Ban96a, Ban96e, Ban96c, BD97] for more details.

Outlook

Currently, the situation on the Web [TBLP92] can be compared to that of structured programming in the seventies: information (data) and behavior (programs) are separated; programs are used to manipulate data, and the associations that determine which program is to be invoked for which datum[2] are maintained in a file using MIME [BF92]. This brings with it all the disadvantages of structured programming, which will not be dealt with here.

[2]For example which application has to be started when an MPEG file is received by the browser.

Second, the information on the Web is untyped (in the sense of types as used in programming languages)[3] and has no inherent structure; most of the information available on the Web is in the form of ASCII or binary files.
Third, access to the distributed information space is performed via an ASCII-based protocol (HTTP) [BLFF96].
It is the author's assumption that these three points may change in the future as follows.
First, the Web may move from the 'structured programming' philosophy to an object-oriented paradigm in which data and programs are encapsulated together, yielding autonomous data entities which have a behavior; that is, they know how to manipulate themselves. For advantages of an object-oriented approach over a structured one, refer to the respective literature (e.g. [Mey88]).
Second, information on the Web may become typed, that is, instances of classes.
Third, HTTP as a protocol may be superseded by either IIOP [OMG95] as an 'object-protocol' on the protocol level, or, better, with an RPC-like CORBA API, which is not concerned with protocols, but offers a higher level of abstraction.
In a line, in the future the universal request for information in the form of:
`http://www.zurich.ibm.com/~bba/gom.ps`
might be replaced by
`orb://com/ibm/zurich/ban/papers/gom`.
The form of information on the Web may move from flat files to objects, which would be *specified* using CORBA interfaces and *instantiated* by creating instances of those interfaces. The above request typed within a browser would therefore, for example, call a CORBA naming service to request an object reference to an instance of class `Paper` called `gom` in the hierarchical naming context `com/ibm/zurich/ban/papers`.
CORBA objects can be classified into two categories: those that know how to display themselves within a container and those that do not. The paper instance retrieved above would have to know how to display itself in the browser's context, otherwise it could not be displayed (maybe only the data members would be shown). But, of course, any 'displayable' object may use other non-graphical objects to fulfill its task.
If this vision of a 'typed object-oriented' Web becomes reality, then the need to access objects distributed across the Web will arise. It is impossible for a browser to know all the types (interfaces) of all the objects it will ever encounter during a session, since the objects to be manipulated are typically chosen by a user pointing his/her browser to a random URL. Although it is envisaged that a few frequently used types such as files, images, audio and video data, directories etc. will be known by a browser, the entire range of other types available on the Web cannot be known.
Therefore, a model based on the concept of generic types / classes and runtime instance-creation / operation-dispatching using metadata as exemplified by GOM will be needed to handle information on the Web. The association between an instance that represents a piece of information and its type, which is located in an interface repository, can be compared to that of a piece of information and its MIME 'type' as determined in a configuration file in the Web server. But unlike information in the current Web, 'object-oriented information' is structured in the form of classes and have behavior.

[3] MIME associations can hardly be called types.

Research on how object-oriented concepts can be integrated with the Web, and what potential benefits can be achieved has just started. It is hoped that some of the ideas presented in this thesis may contribute to that research.

Appendix A

OMG IDL Definition of GOM Interfaces

The OMG IDL definitions of all interfaces the the Generic Object Model defines are given in the following sections. To prevent name space collisions, all interfaces are contained in an IDL module called `GOM`. However, for space and readability reasons, the module definition was omitted from the code.

A.1 Instance Model

```
#ifndef INSTANCE_MODEL
#define INSTANCE_MODEL

interface Adapter;
interface Val;
interface Arglist;
interface Dictionary;
interface InputStream;
interface OutputStream;

#include <MetaModel.idl>
#include <EventModel.idl>

const string ttl_policy="ttl=60";
const string never_cache="never_cache";
const string always_cache="always_cache";

enum GomKind {
    GenObjKind, ValKind, NILKind, ObjRefKind, BoolKind, CharKind,
    ShortKind, IntKind, LongKind, DoubleKind, StrKind, EnumKind,
    StructKind, SequenceKind, UnionKind, ArrayKind, AnyKind,
    MetaObjKind, MetaLongKind, MetaFloatKind, MetaStringKind, MetaListKind
};
```

```
interface GomElement {
    GomKind             GetKind();
    string              AsString();
    GomElement          Copy();
    boolean             Read(in InputStream is);
    boolean             Dump(in OutputStream os);
};

enum GomErrcode { ATTR_NOT_FOUND, ATTR_READONLY, TYPE_MISMATCH, INDEX_ERR };
enum ExType {GOM_EX, TARGET_EX};

exception GenEx {
    string              name;
    ExType              ex_type;
    Dictionary          members;
};

interface Dictionary {
    Val                 Get(in string key);
    Val                 GetAt(in long index);
    void                Set(in string key, in Val new_val);
    boolean             SetAt(in long index, in Val new_val);
    long                Size();
};

interface Arglist {
    long                Size();
    void                Add(in string parameter_name, in Val v);
    Val                 Get(in string parameter_name);
    Val                 At(in long index);
};

interface GenObj : Val {  // An object can also be a value
    attribute Adapter    adapter;
    attribute string     classname;
    attribute string     instance_name;
    attribute Dictionary properties;

    Val                 Get(in string attrname) raises(GenEx);
    void                GetN(in Dictionary values) raises(GenEx);

    void                Set(in string attrname, in Val new_val) raises(GenEx);
    void                SetN(in Dictionary values) raises(GenEx);

    Val                 Execute(in string opname, in Arglist args) raises(GenEx);
    void                Delete() raises(GenEx);

    MetaObj             GetClassDef() raises(GenEx);
```

```
    MetaObj             GetAttributeDef(in string attrname) raises(GenEx);
    MetaObj             GetOperationDef(in string opname) raises(GenEx);
    MetaObj             GetElementDef(in string element_name) raises(GenEx);

    Val                 GetProperty(in string name);
    boolean             SetProperty(in string name, in Val new_val);

    string              GetPolicy();
    void                SetPolicy(in string new_policy);
};


interface Adapter {
    attribute string    object_model;
    GenObj              Create(in string classname, in string inst_name,
                               in string target_location, in Arglist args);

    Val                 Get(in GenObj objref, in string attrname);
    void                GetN(in GenObj objref, in Dictionary values);

    void                Set(in GenObj objref, in string attrname, in Val new_val);
    void                SetN(in GenObj objref, in Dictionary values);

    Val                 Execute(in GenObj objref, in string opname,
                                in Arglist args);
    void                Delete(in GenObj objref);

    MetaObj             GetClassDef(in GenObj objref);
    MetaObj             GetAttributeDef(in GenObj objref, in string attrname);
    MetaObj             GetOperationDef(in GenObj objref, in string opname);
    MetaObj             GetElementDef(in GenObj objref, in string element_name);

    ProxyFilter         CreateFilter(in string type, // class or struct
                                     in string target_location,
                                     in string target_name,
                                     in Arglist attrs,
                                     in ConsumerList consumers);

    void                SendEvent(in EventInfo event_info,
                                  in string destination_address);
};


interface Factory {
    GenObj              Create(in string object_model, in string classname,
                               in string inst_name,
                               in string proxy_location,
                               in string target_location,
```

```
                          in Arglist args) raises(GenEx);

    Val                 GetConstant(in string object_model, in string const_name);
    string              GetPolicy(in string object_model);
    void                SetPolicy(in string object_model, in string new_policy);
};


interface Val    : GomElement {};
interface NIL    : Val {};
interface Bool   : Val { attribute boolean  val;      };
interface Char   : Val { attribute char     val;      };
interface Short  : Val { attribute short    val;      };
interface Int    : Val { attribute long     val;      };
interface Long   : Val { attribute long     val;      };
interface Double : Val { attribute double   val;      };
interface Str    : Val { attribute string   val;      };
interface Enum   : Str { attribute long     long_val; };

interface Struct : Val {
    Val                 Get(in string key);
    Val                 GetIndex(in long index);
    boolean             Set(in string key, in Val val);
    boolean             Add(in string key, in Val val);
    boolean             Remove(in string key);
    long                Size();
};

interface Sequence : Val {
    boolean             Add(in Val new_val);
    long                Size();
    Val                 At(in long index);
};

interface Union : Val {
    attribute string    name;
    attribute Val       val;
};

interface Array : Val {
    Val                 Get(in long index);
    boolean             Set(in long index, in Val new_val);
    long                Size();
};

#endif
```

A.2 Meta Model

```
#ifndef META_MODEL
#define META_MODEL

interface GomElement;
interface MetaElement;
interface MetaLong;
interface MetaFloat;
interface MetaString;
interface MetaList;
interface MetaObj;
interface Dictionary; // associations of <string , MetaObj>

#include "InstanceModel.idl"

typedef sequence<MetaElement>  MetaElementList;

interface MetaElement : GomElement {
    attribute string          name; // e.g. "ConstantDef", "ClassDef" etc.
};

interface MetaLong : MetaElement {
    attribute long            val;
};

interface MetaFloat : MetaElement {
    attribute double          val;
};

interface MetaString : MetaElement {
    attribute string          val;
};

interface MetaList : MetaElement {
    attribute MetaElementList val;
};

// ------------------------------------------------------------
//                       MetaObj:
//
// Generic container for metadata. It contains a dictionary
// with strings as keys and MetaElement instances as values.
// The values can be instances of MetaLong, MetaString and
// (recursively) MetaObj.
// ------------------------------------------------------------

interface MetaObj : MetaElement {
```

```
    attribute Dictionary      dict;

    MetaElement               Get(in string name);
    boolean                   Set(in string name, in MetaElement new_el);
    long                      Size();
};

#endif
```

A.3 Event Model

```
#ifndef EVENT_MODEL
#define EVENT_MODEL

interface GenPushConsumer;
interface ProxyFilter;
interface EventService;
interface Struct;

typedef sequence<GenPushConsumer> ConsumerList;
typedef sequence<ProxyFilter>     FilterList;
typedef Struct                    EventInfo;

#include "InstanceModel.idl"

interface GenPushConsumer {
    void HandleEvent(in EventInfo event_info);
};

interface ProxyFilter {
    attribute ConsumerList consumers;
    attribute Val          target_filter;
    void                   AddConsumer(in GenPushConsumer consumer);
    void                   RemoveConsumer(in GenPushConsumer consumer);
};

interface LocalFilter {
    attribute string       filter_construct;
    attribute ConsumerList consumers;
    void                   AddConsumer(in GenPushConsumer consumer);
    void                   RemoveConsumer(in GenPushConsumer consumer);
    boolean                Evaluate(in EventInfo event_info);
};

interface EventService {
    ProxyFilter       CreateFilter(in string object_model,
                                   in string type,
```

```
                                in string target_location,
                                in string target_name,
                                in Arglist args,
                                in ConsumerList consumers);

    LocalFilter       CreateLocalFilter(in string object_model,
                                     in string oql_expr,
                                     in ConsumerList consumers);

    void              AddFilter(in string object_model,
                                 in Val new_filter);

    FilterList        GetFilters(in string object_model);

    EventInfo         PullEvent(in string object_model, in boolean wait);

    void              PushEvent(in string object_model,  // used by adapters
                                 in FilterList filters,
                                 in EventInfo event_info);

    void              SendEvent(in string object_model,
                                 in EventInfo event_info,
                                 in string destination_address);
};

#endif
```

A.4 Metadata Repository

```
#ifndef MR_REP
#define MR_REP

#include "MetaModel.idl"

interface MetadataRepository;
interface MetadataAdapter;
interface MetadataCache;
interface Dictionary; // associations of <string , MetaObj>
interface InputStream;
interface OutputStream;

struct TypeCodeConstant {
    long    type_code;
    string  name;
};

typedef sequence<MetadataAdapter>     AdapterList;
```

```
typedef sequence<MetadataCache>       CacheList;
typedef sequence<TypeCodeConstant>    TypeCodeList;

// ------------------------------------------------------------
//                    MetadataRepository
// ------------------------------------------------------------
interface MetadataRepository {
    attribute AdapterList  adapters;
    attribute CacheList    caches;

    void                   AddAdapter(in MetadataAdapter new_adapter);
    void                   AddCache(in MetadataCache new_cache);

    boolean                Read(in InputStream is);  // Reads the db
    boolean                Dump(in OutputStream os); // Dumps the db

    MetaObj                Find(in string object_model,   // e.g. "X.700"
                                in string key,            // e.g. "classes"
                                in string property_name); // e.g. "circuit"

    // Methods specifically provided for the one instantiation of the
    // generic meta model: CORBA
    // ------------------------------------------------------------
    MetaObj                FindConstant(in string object_model,
                                        in string constant_name);

    MetaObj                FindTypedef(in string object_model,
                                       in string typedef_name);

    MetaObj                FindException(in string object_model,
                                         in string exception_name);

    MetaObj                FindModule(in string object_model,
                                      in string module_name);

    // Fully scoped name, e.g. "X700:op1vol4/circuit:1.1"
    MetaObj                FindClass(in string object_model,
                                     in string classname);

    // The attribute name has to be fully scoped,
    // that is the classname has to be given as well, e.g.:
    // "CORBA:/MyProject/Person:2.0:salary"
    MetaObj                FindAttribute(in string object_model,
                                         in string attrname);

    // The operation name has to be fully scoped,
    // that is the classname has to be given as well, e.g.:
    // "CORBA:/MyProject/Person:2.0:IncreaseSalary"
```

```
MetaObj                FindOperation(in string object_model,
                                     in string opname);

// Write interface: add data to MR. Existing metadata is replaced
// ---------------------------------------------------------------
boolean                Add(in string object_model, in string key,
                           in string property_name, in MetaObj data);

boolean                AddConstant(in string object_model,
                                   in string constant_name,
                                   in MetaObj data);

boolean                AddTypedef(in string object_model,
                                  in string typedef_name,
                                  in MetaObj data);

boolean                AddException(in string object_model,
                                    in string exception_name,
                                    in MetaObj data);

boolean                AddModule(in string object_model,
                                 in string module_name,
                                 in MetaObj data);

boolean                AddClass(in string object_model,
                                in string classname,
                                in MetaObj data);

boolean                AddAttribute(in string object_model,
                                    in string attrname,
                                    in MetaObj data);

boolean                AddOperation(in string object_model,
                                    in string opname,
                                    in MetaObj data);


// Gets all metadata from a specific source using adapters.
// This is important in the case where e.g. all classes in
// a system have to be discovered. All methods will call
// LoadAllXXX() of the adapter for the object_model given.
// All cache data will be deleted before copying of the new
// information to the cache.

long                   LoadAllConstants(in string object_model);
long                   LoadAllTypedefs(in string object_model);
long                   LoadAllExceptions(in string object_model);
long                   LoadAllModules(in string object_model);
```

```
    long                   LoadAllClasses(in string object_model);
    TypeCodeList           GetTypeCodes(in string object_model);
    string                 GetTypeCodeName(in string object_model,
                                           in short type_code);
};


// ---------------------------------------------------------------
//                MetadataAdapter (Abstract):
//
// For each specific object model, a separate subclass of
// MetadataAdapter has to be created which retrieves metadata
// from a specific source (e.g. CORBA's interface repository (IR)
// and returns a copy of the desired information in the form
// dictated by the MetadataAdapter interface. The methods 'Find'
// and 'GetAll' have to be overwritten.
// ---------------------------------------------------------------

interface MetadataAdapter {
    attribute string       object_model;
    attribute string       location; // of specific metadata source

    MetaObj                Find(in string key,            // e.g. "classes"
                                in string property_name); // e.g. "circuit"
    MetaObj                GetAll(in string key);

    // ---------------------------------------------------------------
    // Specific functions
    // ---------------------------------------------------------------
    MetaObj                FindConstant(in string constant_name);
    MetaObj                FindTypedef(in string typedef_name);
    MetaObj                FindException(in string exception_name);

    // fully scoped name, e.g. "IBM::Zurich::bba"
    MetaObj                FindModule(in string module_name);

    // fully scoped name, e.g. "IBM::Zurich:bba::MyProject::Customer"
    MetaObj                FindClass(in string classname);

    MetaObj                GetAllConstants();
    MetaObj                GetAllTypedefs();
    MetaObj                GetAllExceptions();
    MetaObj                GetAllModules();
    MetaObj                GetAllClasses();
    TypeCodeList           GetTypeCodes();
    string                 GetTypeCodeName(in short type_code);
};
```

```
// -----------------------------------------------------------
//                          MetadataCache:
//   -----------------------------------------------------------
interface MetadataCache : MetadataAdapter {
    attribute Dictionary   dict;

    MetaElement            Get(in string key);
    // create new key if not present
    void                   Set(in string key, in MetaElement new_val);
    long                   Size();

    // override operations 'Find', 'GetAll' and 'GetTypeCodeName'

    boolean                Read(in InputStream is);  // Reconstructs the dict
    boolean                Dump(in OutputStream os);  // Dumps the dict
    void                   Flush();
};

#endif
```

A.5 OSI Agent Specific Code

```
#ifndef OSI_SPEC
#define OSI_SPEC

#include <InstanceModel.idl>
#include <CosNaming.idl>

module OsiSpecific {

  typedef sequence<GenObj> GenObjList;
  typedef sequence<Val>    ValList;

  interface ProxyAgent {
      attribute string agent_address; // AE-title

      boolean          SyncNamingService(in CosNaming::NamingContext nc,
                                         in string distinguished_name,
                                         in short level) raises(GenEx);
      GenObjList       DiscoverManagedObjects(in string distinguished_name,
                                              in short level) raises(GenEx);
  };

  interface GroupIterator {
      long             Size();
```

```
    GenObj           Next();
    void             Reset();
    boolean          Remove();
    boolean          More();
};

interface GenGroupObj : GenObj {
    void             SetAgentAddress(in string agent_address);
    void             SetScope(in short scope);
    void             SetFilter(in Val filter); // e.g. GenObj or Str

    void             PerformSelection() raises(GenEx);
    void             ClearResultSet();

    ValList          GroupGet(in string attrname) raises(GenEx);
    void             GroupSet(in string attrname, in Val new_val)
                       raises(GenEx);
    ValList          GroupExecute(in string opname, in Arglist args)
                     raises(GenEx);
    GroupIterator    GetResultIterator();

    boolean          Add(in GenObj    new_obj);
    boolean          Remove(in GenObj old_obj);
    long             Size();

    // Operations Get and Execute are overridden to use the first result
    // that is returned (failure safety)
};

};

#endif
```

Appendix B

Layout Definitions

B.1 CORBA Layout

```
1. Type code constants
----------------------
0        tk_null          10       tk_octet
1        tk_void          11       tk_any
2        tk_short         12       tk_objref
3        tk_long          13       tk_struct
4        tk_ushort        14       tk_union
5        tk_ulong         15       tk_enum
6        tk_float         16       tk_string
7        tk_double        17       tk_sequence
8        tk_boolean       18       tk_array
9        tk_char          19       tk_except
                          20       tk_recursive


2. Elements
-----------

2.0 Metadata Repository (MR)

  Keys:             Values:

  "modules"         Modules (MetaObj) -> c.f. Modules
  "classes"         Classes (MetaObj) -> c.f. Classes
  "constants"       Constants (MetaObj) -> c.f. Constants
  "typedefs"        Type definitions (MetaObj) -> c.f. Typedefs
  "exceptions"      Exceptions (MetaObj) -> c.f. Exceptions


2.1 Modules (MetaObj)

  Keys:             Values:
```

```
  "<module name>"  Module (MetaObj)

    Module (MetaObj):

      Keys:           Values:
      "name"          Name of module (MetaString)
      "constants"     Constants (MetaObj) -> c.f. Constants
      "typedefs"      Type definitions (MetaObj) c.f. Typedefs
      "exceptions"    Exceptions (MetaObj) -> c.f. Exceptions
      "modules"       Modules (MetaObj) -> c.f. Modules
      "classes"       Classes (MetaObj) -> c.f. Classes

Descr: Models a name space in CORBA. Can contain other modules or
classes.


2.2 Classes (MetaObj)

  Keys:             Values:
  "<class name>"    Class (MetaObj)

    Class (MetaObj):

      Keys:           Values:
      "name"          Name of class (MetaString)
      "superclasses"  Superclasses (MetaObj) -> c.f. Classes
      "subclasses"    Subclasses (MetaObj) -> c.f. Classes
      "attributes"    Attributes (MetaObj) -> c.f. Attributes
      "operations"    Operations (MetaObj) -> c.f. Operations
      "constants"     Constants (MetaObj) -> c.f. Constants
      "typedefs"      Type definitions (MetaObj) -> c.f. Typedefs
      "exceptions"    Exceptions (MetaObj) -> c.f. Exceptions

Descr: Models a class (e.g. CORBA interface or GDMO template)
       (GDMO Notifications are not yet modeled)


2.3 Attributes (MetaObj)

  Keys:            Values:
  "<attrname>"     Attribute

    Attribute (MetaObj):

      Keys:           Values:
      "name"          Name of attribute (MetaString)
      "type"          Type (MetaObj) -> c.f. Type
      "access_mode"   Access mode, e.g. read-only, writable (MetaLong)
```

```
                      (0 == read-only, 1 == writable)

Descr: Models an attribute (e.g. CORBA attribute or ASN.1 attribute)


2.4 Type (MetaObj)

  Keys:            Values:
  "name"           Name of type
  "type_code"      Type code constant (MetaLong)
  "parm"           Parameter(s) (subclass of MetaElement)

Descr: Models a type. Depending on the type code constant, the
       parameter may be empty, simple of constructed (c.f. CORBA
       TypeCodes). Examples:

       type 'short':

          "name"      --> NULL (only constructed types have names)
          "type_code" --> MetaLong (5)
          "parm"      --> <null>

       type 'struct'

          "name"      --> Name of struct
          "type_code" --> MetaLong (13)
          "parm"      --> MetaObj (keys   == names of members,
                                   values == Subclasses of MetaElement


2.5 Operations (MetaObj)

  Keys:             Values:
  "<opname>"        Operation (MetaObj)

    Operation (MetaObj):

      Keys:            Values:
      "name"           Name of operation (MetaString)
      "mode"           Execution mode, e.g. synchronous, asynchronous
                       (MetaLong)
      "return_type"    Return value. Same members as 'Type'. (MetaObj)
      "parameters"     List of parameter descriptions
                       (Parameter). (MetaObj) -> c.f. Parameters
      "exceptions"     MetaList of MetaStrings. Lists all
                        exceptions that may be thrown. Note that CORBA
                        stadard exception may not be listed, but can be
                        thrown as well.
```

Descr: Describes a list of parameters (c.f. below).

2.6 Parameters (MetaObj)

```
  Keys:            Values:
  "<parm name>"    Parameter (MetaObj)

    Parameter (MetaObj):

      Keys:           Values:
      "name"          Name of parameter (MetaString)
      "mode"          Parameter mode (e.g. in, out, in/out) (MetaLong)
      "type"          Type of parameter (c.f. 'Type') (MetaObj)
```

Descr: Describes a parameter.

2.7 Constants (MetaObj)

```
  Keys:            Values:
  "<const name>"   Constant (MetaObj)

    Constant (MetaObj):

      Keys:           Values:
      "name"          Name of constant (MetaString)
      "type"          Type of constant (MetaObj)
      "value"         Value of constant. (GomElement; MetaElement or Val)
```

Descr: The "value" property points to a subclass of MetaElement or Val which both inherit from GomElement. Thus, the kind of the element can be retrieved using method GetKind(). If the constant has a simple value, probably subclasses of MetaElement are used, otherwise subclasses of Val can be used.

2.8 Typedefs (MetaObj)

```
  Keys:            Values:
  "<typedef name>" Typedef (MetaObj)

    Typedef (MetaObj):

      Keys:           Values:
      "name"          Name of type definition (MetaString)
      "type"          Type of type definition (MetaObj)
```

```
Descr: Models a type definition, e.g. in OMG IDL:
       struct Info { short  age; string name; };


2.9 Exceptions (MetaObj)

  Keys:                 Values:
  "<exception name>"    Exception (MetaObj)

    Exception (MetaObj):

    Keys:               Values:
    "<member name>"     Type of member (MetaObj) -> c.f. Type


Descr: Describes an exception
```

B.2 GDMO Layout

```
1. Type code constants
----------------------
1        BOOLEAN              21       PrintableString
2        INTEGER              22       TeletexString
3        BITSTRING            23       VideotexString
4        OCTETSTRING          24       IA5String
5        NULL                 25       UTCString
6        OBJECT IDENTIFIER    26       GeneralizedTime
7        ObjectDescription    27       GraphicString
8        EXTERNAL             28       VisibleString
9        REAL                 29       GeneralString
10       ENUMERATED           30       CharacterString
11       ENCRYPTED            31       CHOICE
12-15    <reserved>           32       ANY
16       SEQUENCE             33       ANY DEFINED BY
17       SEQUENCE-OF          34       SELECTION
18       SET                  35       TAGGED
19       SET-OF               36       recursive
20       NumericString


2. Elements
-----------

2.0 Metadata Repository (MR)

  Keys:            Values:
```

```
  "templates"      Templates (MetaObj).
  "name_bindings"  NameBindings (MetaObj).
  "packages"       Packages (MetaObj).
  "parameters"     Parameters (MetaObj).
  "attributes"     Attributes (MetaObj).
  "attr_groups"    AttrGroups (MetaObj).
  "behaviors"      Behaviors (MetaObj).
  "actions"        Actions (MetaObj).
  "notifications"  Notifications (MetaObj).
  "oid_mappings"   Mappings
  "asn1_types"     Asn1Types (MetaObj).


2.1 NameBindings (MetaObj)

  Keys:                        Values:
  "<OID of subordinate class>"  Subordinate class (MetaObj).

    Subordinate Class (MetaObj):

    Keys:                Values:
    "name"               Symbolic name of name binding (MetaString)
    "oid"                OID of name binding (MetaString)
    "superior_classes"   List of OIDs (MetaList)
    "naming_attr"        Symbolic name of naming attribute (MetaString)
    "behavior"           All behavior clauses are merged (MetaString)
    "creation"           Creation modifier clause (MetaObj).
    "deletion"           Deletion modifier clause (MetaObj).

      Creation Modifier (MetaObj):

      Keys:                       Values:
      "with_reference_object"           List of parameter OIDs (MetaList)
                                        or NULL if absent.
      "with_automatic_instance_naming"  List of parameter OIDs (MetaList)
                                        or NULL if absent.

      Deletion Modifier (MetaObj):

      Keys:                              Values:
      "only_if_no_contained_objects" |
      "deletes_contained_objects"        List of parameter OIDs
                                         (MetaList) or NULL.


Descr: Defines the constraints under which managed objects can be
created and deleted. Metadata contains a number of name bindings which
```

```
are keyed by the OID of the subordinate class for faster lookup.


2.2 Templates (MetaObj)

  Keys:                 Values:
  "<templ. name (OID)>" Template (MetaObj)

    Template (MetaObj):

    Keys:            Values:
    "name"           Symbolic name of template (MetaString)
    "oid"            OID of template (MetaString)
    "document"       Document name in which template is defined (MetaString)
    "superclasses"   List of superclass OIDs (MetaList)
    "mand_packages"  List of mandatory package OIDs (MetaList of MetaStrings)
    "cond_packages"  List of conditional packages (MetaObj)

      Conditional packages (MetaObj):

      Keys:                     Values:
      "<package name (OID)>"    Condition under which package is to be
                                included (MetaString)

Descr: Defines a GDMO template.


2.3 Packages (MetaObj)

  Keys:                  Values:
  "<package name (OID)>" Package (MetaObj)

    Package (MetaObj):

    Keys:            Values:
    "name"           Symbolic name of package (MetaString)
    "oid"            OID of package (MetaString)
    "behavior"       All behavior clauses of a package are merged into
                     one single string (MetaString)
    "attributes"     Attributes of package (MetaObj).
    "attr_groups"    Attribute groups (MetaObj). Keys are symbolic names
                     of attribute groups
    "actions"        List of action OIDs (MetaList)
    "notifications"  List of notification OIDs (MetaList)

      Attribute of package (MetaObj):

      Keys:          Values:
```

```
    "<attrname>"   Attribute (MetaObj)

      Attribute (MetaObj):

      Keys:        Values:
      "oid"          OID of attribute (MetaString)
      "name"         Symbolic name of attribute (MetaString)
      "properties"   Properties of atttribute (MetaObj)
      "parameters"   List of parameter OIDs (MetaList)

        Properties of attribute (MetaObj):

        Keys:                  Values:
        "replace_with_default" NULL
        "default_value"        Default value (GomElement).
                               Instance of ASN.1 value
        "initial_value"        Default value (GomElement).
                               Instance of ASN.1 type
        "permitted_values"     ASN.1 type (MetaString)
        "required_values"      ASN.1 type (MetaString)
        "get_replace"          NULL
        "add_remove"           NULL

    Attribute groups of package (MetaObj):

    Keys:              Values:
    "<attrgroup name>" AttributeGroup (MetaObj)

      AttributeGroup (MetaObj):

      Keys:         Values:
      "name"        Symbolic name of attribute group (MetaString)
      "oid"         OID of attribute group (MetaString)
      "attributes"  List of attribute OIDs (MetaList)

      Descr: If the attributes of an attribute group are defined inline,
             then "oid" will have a NULL value.


Descr: Defines mandatory or conditional packages.


2.4 Parameters (MetaObj)

  Keys:                Values:
  "<parmname (OID)>"   Parameter (MetaObj)

    Parameter (MetaObj):
```

```
  Keys:             Values:
  "name"            Symbolic name of parameter (MetaString)
  "oid"             OID of parameter (MetaString)
  "context"         Context (MetaString). Values are one of the
                    following strings: "action_info", "action_reply",
                                       "event_info", "event_reply" or
                                       "specific_error"
  "syntax"          Name of ASN.1 type (MetaString). In the form
                    "<documentname>.<typename>"
  "behavior"        All behavior clauses of a package are merged into
                    one single string (MetaString)
```

```
2.5 Attributes (MetaObj)

 Keys:                 Values:
 "<attrname (OID)>"    Attribute (MetaObj)

  Attribute (MetaObj):

  Keys:             Values:
  "name"            Symbolic name of attribute (MetaString)
  "oid"             OID of attribute (MetaString)
  "syntax"          Name of ASN.1 type (MetaString). In the form
                    "<documentname>.<typename>"
  "matches_for"     List of matching specifiers (MetaList of MetaStrings).
                    Specifiers are one of the following strings:
                    "equality", "ordering", "substrings", "set_comparison"
                    or "set_intersection"
  "behavior"        All behavior clauses of a package are merged into
                    one single string (MetaString)
  "parameters"      List of parameter OIDs (MetaList)
```

```
2.6 AttrGroups (MetaObj)

 Keys:                     Values:
 "<attrgroup name (OID)>"  AttributeGroup (MetaObj)

  AttributeGroup (MetaObj):

  Keys:             Values:
  "name"            Symbolic name of attribute group (MetaString)
  "oid"             OID of attribute group (MetaString)
  "attributes"      List of attribute OIDs (MetaList)
  "fixed"           NULL. If this key is present, the attribute group
                    cannot be extended, otherwise it is extensible.
```

```
  "description"    Description of attribute group (MetaString)


2.7 Behaviors (MetaObj)

 Keys:               Values:
 "<beh. name (OID)>" Behavior (MetaObj)

   Behavior (MetaObj):

   Keys:            Values:
   "name"           Symbolic name of behavior (MetaString)
   "oid"            OID of behavior (MetaString)
   "description"    Description of behavior (MetaString)


2.8 Actions (MetaObj)

 Keys:                  Values:
 "<action name (OID)>" Action (MetaObj)

   Action (MetaObj):

   Keys:            Values:
   "name"           Symbolic name of action (MetaString)
   "oid"            OID of action (MetaString)
   "behavior"       All behavior clauses of a package are merged into
                    one single string (MetaString)
   "mode"           NULL. If this key is present, mode is confirmed,
                    otherwise, it may be confirmed or unconfirmed.
   "parameters"     List of parameter OIDs (MetaList)
   "info_syntax"    Name of ASN.1 type (MetaString). In the form
                    "<documentname>.<typename>"
   "reply_syntax"   Name of ASN.1 type (MetaString). In the form
                    "<documentname>.<typename>"


2.9 Notifications (MetaObj)

 Keys:                  Values:
 "<notif. name (OID)>" Notification (MetaObj)

   Notification (MetaObj):

   Keys:            Values:
   "name"           Symbolic name of notification (MetaString)
   "oid"            OID of notification (MetaString)
   "parameters"     List of parameter OIDs (MetaList)
```

```
    "info_syntax"    Name of ASN.1 type (MetaString). In the form
                     "<documentname>.<typename>"
    "reply_syntax"   Name of ASN.1 type (MetaString). In the form
                     "<documentname>.<typename>"


2.10 Mappings (MetaObj)

  Keys:             Values:
  "<OID name>"      Mapping (MetaObj)

    Mapping (MetaObj):

    Keys:            Values:
    "name"            Symbolic form (MetaString)
    "oid"             OID form (MetaString)
    "document"        Name of document in which element is defined (MetaString)

Descr: Mappings are keyed by OID. Maybe an additional mapping table
that is keyed by symbolic name has to be introduced.


2.11 Asn1Types (MetaObj)

  Keys:               Values:
  "<document name>"   ASN.1 types defined in this document
                      (MetaObj). Keys are ASN.1 type names

    ASN.1 type (MetaObj):

    Keys:          Values:
    "name"         Name of ASN.1 type (MetaString)
    "document"     Name of document (MetaString)
    "type_code"    Type code constant (MetaLong)
    "parm"         Parameter(s). See next section (Mapping of ASN.1 Types)
```

B.3 SNMP Layout

```
1. Type code constants
----------------------
0        NULL                6        NetworkAddress
1        INTEGER             7        IpAddress
2        OCTET STRING        8        Counter
3        OBJECT IDENTIFIER   9        Gauge
4        SEQUENCE           10        TimeTicks
5        SEQUENCE-OF        11        Opaque
```

```
2. Elements
-----------

2.0 Metadata Repository (MR)

  Keys:             Values:
  "<MIB name>"      MIB (MetaObj)

2.1 MIB (MetaObj)

  Keys:                     Values:
  "<symb. variable name>"   Variable (MetaObj)

    Variable (MetaObj):

      Keys:            Values:
      "name"           Symbolic name of variable (MetaString)
      "oid"            OID of variable  (MetaString)
      "syntax"         Type of variable (MetaObj) -> c.f. Type
      "access"         Access restrictions (MetaLong)
                       0: read-only
                       1: read-write
                       2: write-only
                       3: not accessible
      "status"         Status os object (MetaLong)
                       0: mandatory
                       1: optional
                       2: obsolete

2.2 Type (MetaObj)

  Keys:          Values:
  "name"         Name of ASN.1 type (MetaString)
                 (e.g. "OCTET STRING", "INTEGER" etc.)
  "type_code"    Type code constant (MetaLong) (see pt. 1)
  "parm"         Parameter(s). Describes aggregate types such as
                 SEQUENCE, SEQUENCE-OF etc. The use of the parameter
                 is the same as for the GDMO layout (c.f GDMO layout)
```

Appendix C

GOMscript Language Overview

C.1 Overview

GOMscript is an interpreter for a simple C++-like language. It contains the usual control flow statements such as `if-then-else`, `while` and `for`, and allows to define functions and classes. GOMscript may be extended by adding functions and classes located in shared libraries. These are called *extensions*.
Unlike C++, GOMscript is untyped. This means that values will only be assigned an (implicit) type at runtime and not at compile time as in C++. This eliminates the necessity of a compiler, resulting in faster development, but also adds overhead to the running program. However, GOMscript was not written to compete with C++, but to produce a tool that allows to interactively put together small applications and to write simple scripts.
GOMscript is publicly available and can be downloaded in binary- (AIX 3.2.5, 4.1.1, Linux 2.X) and source code form from

`http://sunsite.unc.edu/pub/Linux/devel/lang/gomscript/`.

or

`http://SAL.KachinaTech.COM/F/1/GOMSCRIPT.html`

In the following sections the language is described.

C.2 Types

GOMscript recognizes the following built-in types:

Null Represents the 'nil' value. It is often returned in the case of a failure.

Number Both real and integer values can be represented by this type. Example: `3`, `3.145`.

String Example: `"Hello world"`

List A collection of elements, e.g. `#(1, 2, 3)`, `#("Bela", "Janet", 3.23, null)` or `#(1, 2, #(-34, 4.3), 56)`. Lists can also be created using the `new` operator:

`a=new List(1,2,3);`. As a list is a class, its member functions can be inspected using `inspect`, e.g. `inspect MyClass;`. New elements can be added to a list using method `Add`. Sample code for lists is shown below (the '>' sign is the prompt):

```
> l=#(1,2,3);
> l;
> #(1, 2, 3);
> l.Length();
> 3
> inspect l;
  #(1, 2, 3)
  Methods:
  --------
  Length
  Add
  At
  AtPut
  AddFirst
  Delete
> l.Add(34);
> l.AtPut(1, 100);
> l;
> #(1, 100, 3, 34);
```

Struct Represents a struct that contains other named values. Each member has a name and a value: `a=new Struct("name", "Bela", "age", 31);`. Each member of the struct can be accessed using the dereference operator '`.`', e.g. `mystruct.age=23;`.

Union Represents a C-like union with a name and a value: `a=new Union("age", 31);`.

C.3 Identifiers

Identifiers denote variables, function- and class names and have to consist of the characters a-zA-Z plus the underscore ('_') and period ('.') characters, as well as digits. Digits, however, may not be in the first position. Valid identifiers are e.g. `a`, `number_of_bytes`, `a.Length` and `person.age`.

C.4 Expressions

The following expressions are present in GOMscript:

Expression	**Description**
expr `+` expr	Addition
expr `-` expr	Subtraction
expr `*` expr	Multiplication
expr `/` expr	Division
expr `MOD` expr	Modulo division
`-` expr	Unary minus
`(` expr `)`	Parentheses
identifier	Value of identifier
`NUMBER`	Number
`STRING`	String
`NULL`	Null
List	List
Function call	Value of function call
`NEW`	New expression

C.5 Statements

Each program consists of a (possibly empty) list of statements. Each statement is terminated by a semicolon ';' character. GOMscript starts at the first statement and processes all statements in the order they are read until no more statements are left (or the user terminates the program). The following statements are available in GOMscript:

Statement	**Description**
ident `=` expr	Assignment
expr	Expression
vars	Definition of variables in class or function scope
function def	Function definition
class def	Class definition
`quit`	Exits the interpreter
`symbols`	Shows all symbols that are currently defined in the symbol table
`classes`	Shows all the classes currently defined
`inspect` expr	Shows variables and methods of an instance
`reload`	Reloads all extensions
`load` string	Interprets the contents of the string
`print(ln)`	Prints a variable or value
`dump` string	Dumps the contents of the symbol table to file *string*
`read` string	Populates the symbol table with the values stored in file *string*
while	while statement
if	if statement
for	for statement
return	return statement
break	break statement
`eval` string	Interprets the contents of *string*

`Symbols` Statement

This statement prints the contents of the symbol table, i.e. all bindings between identifiers and their values. Identifiers may denote both variables and functions. Classes are *not* stored in the symbol table, but are located in a separate table. The contents of that table can be printed using the `classes` statement.
Note that both the `symbols`- and the `classes`-statements do *not* have a terminating semicolon !

`While` Statement

while (*condition*) **{** *block* **}**

The `while` statement continuously executes *block*, which is a list of statements, until *condition* becomes false.
Example:

```
a=0;
while(a < 10) {a=a+1; println a;}
```

`If` Statement

if (*condition*) **{** *block* **}** [**else** **{** block **}**]

The `if` statement evaluates *condition*. If true, the block immediately following it is executed, otherwise the block after a potential `else` is executed. The else-part is optional.
Example:

```
if(a < 5) {println "smaller";}
else      {println "greater";}
```

`For` Statement

for (*identifier* = *expr* (**to** | **downto**) *expr*) **{** *block* **}**

The `for` statement iterates through a range of numbers between two expressions. Both expressions have to evaluate to numeric values. With each iteration, *block* is executed and variable *identifier* is incremented or decremented.
Example:

```
for(i=0 to 100) {println "Number: " + i;}
```

Return statement

return [*expr*]

The `return` statement allows to return from a function of method.
Example:

```
define sqr(x) {return x*x;}
```

Break Statement

break

The `break` statement allows to leave the current block in `for`- and `while` statements.
Example:

```
for(i=0 to 100) {
  println i;
  if(i == 10) {
      break;
  }
}
```

Eval Statement

eval *expr*

The `eval` statement allows to interpret a string which must contain GOMscript code. Note that strings within a string have to be escaped.
Example:

```
> code="println \"Hello, world !\";";
> eval code;
> Hello, world !
```

Function Definition

define *identifier* (*parameters*) **{** *block* **}**

The function definition statement allows to define new functions. It must have an identifier, a number of parameters, and a function body, which is a block (list of statements).
Example:

```
> define fib(x) {
      if(x == 1 or x == 2) { return 1; }
      else {
          return fib(x-1) + fib(x-2);
      }
  }
> fib(12);
> 144
```

Class Definition

> **class** *identifier* [**inherits from** *identifier*] **{**
> [**vars:** *variable declarations*]
> [**methods:** *method definitions*]
> **}**

The class definition statement allows to define new classes. It must have a class name, optional parent class, a number of variables (optional), and a number of method definitions (optional as well).
Variables defined after `vars` will be visible only within instances.[1] These are usually called *local-* or *instance* variables.
Method definitions have the same syntax as functions and can additionally access member variables.
When a class inherits from a parent class, all variables and methods of the parent class are available as well.[2]
Variables and methods inherited from a parent class can be overridden, and do not even have to have the same type or signature, respectively. Instances are created using keyword `new`.
Example:

```
class Person {
    vars:
        salary, name, age, personnel_number;
    methods:
        define Person(s, n, a, p) {  // Constructor
            salary=s; name=n; age=a; personnel_number=p;
        }
        define WhoAreYou() {println "I am a normal person";}
}

class Manager inherits from Person {
    vars: manages;
    methods:
```

[1]There are currently no *class variables* available, i.e. variables visible by all instances of a class.
[2]Unlike C++, there are no inheritance restrictions such as `private` or `protected`.

```
        define Manager(s, n, a, p, managed_persons) {
            super.Person(s, n, a, p);  // Call parent's ctor
            manages=managed_persons;
        }
        define WhoAreYou() {
            print "I am a manager";
            if(manages != null) {
                print " and I manage "
                for(i=0 to manages.Length()-1) {
                    print manages.At(i); print " ";
                }
                println;
            }
        }
}

> m=new Manager(120000, "John Smith", 32, 322649, #("Frank", "Ann"));
> m.salary=m.salary*1.2;
> m.salary;
> 120000
> m.WhoAreYou();
  I am a manager and I manage Bela Ann George
```

Constructors

Constructors are methods that are invoked when a new instance is created, before it is returned to the caller. They must have the same name as the class and may accept none, one or several parameters.

Keyword `super`

If a method of the parent is overridden in the subclass, and the parent's method should be invoked, then keyword `super` can be used. In the example above, the constructor of `Manager` calls the constructor of `Person` before it performs its own processing.

Object-Oriented Features

Encapsulation Variables and methods together form a unit. Methods are defined in the class, whereas each instance gets its separate copy of the instance variables. There are currently no access modifiers for member variables; therefore each variable can be modified at will.[3]

Inheritance and Overriding Only single inheritance is supported. Methods of super-classes can be overridden in subclasses (see method `WhoAreYou` in the example above), allowing to specialize behavior.

[3] It is planned to introduce private variables, however, which would better enforce encapsulation.

Polymorphism Depending on the type of the class, a method invoked may behave differently. As shown above, method `WhoAreYou` sent to an instance of `Manager` triggers behavior different from `Person`.

Overloading Overloading is the definition of one or more methods within a class which have the same name, but different parameters (both in terms of types and number of parameters). This is currently not allowed in GOMscript (similar to CORBA !). The current behavior is that this is not impossible to do (syntactically), but when the method is looked up in the instance's class, the first method found that matches the method name will be taken.

C.6 Extensions

Extensions allow to add user-defined functions and classes to GOMscript. They are *shared libraries*[4] which contain a number of functions and classes. Extensions are located in a certain directory (default or designated by the user) and loaded when GOMscript is started.

The concept of extensions is very important in GOMscript, since these practically represent the equivalent of a runtime library (e.g. `libc.a` in C). All functions of a runtime library such as `printf` or `write` in C would be found in extensions in GOMscript. As will be shown below, an extension has been written that allows to create and manipulate GOM instances.

An extension example is shown below that adds a number of mathematical functions to GOMscript. For sample class extensions, refer to the source code (see the beginning of this chapter).

C.6.1 Example

The common definitions for extensions are shown below (shortened):

[4] Also called *dynamic link libraries (DLLs)* on certain operating systems.

```
#include "Grammar.h"

typedef CL_List<Expr*> Elist;
typedef NativeInstance* (*Constructor)(Elist&);
typedef void (*FuncPtr)();

struct FuncDescr {
    char*          name;
    void*          addr;
    int            number_of_args;
    BType*         parm_types;     // Array of BTypes...
};

struct MethodDescr {
    char*          name;
    MethodPtr      mptr;
    int            number_of_formal_parms;
    BType*         parm_types;     // Array of BTypes
};

struct ClassDescr {
    char*          name;
    char*          superclass;
    Constructor    ctor;
    MethodDescr*   methods;
};

struct Contents  {
    FuncDescr*     functions;
    ClassDescr*    classes;
    FuncPtr        load_func;    /* called when DLL is loaded   */
    FuncPtr        unload_func;  /* called when DLL is unloaded */
};
```

Every *entry point*[5] of a shared library has to be a function which returns the address of a struct of type `Contents`. This contains a list of function description structures, a list of class description structures, a function that is to be executed when the library is loaded into memory, and a function that is to be executed when it is unloaded again.

The `FuncDescr` struct contains the name of the function, its address, the number of parameters and an array of types (one for each parameter). The latter two members can be NULL, in which case argument type-checking has to be done by the author of the extension rather than by the GOMscript runtime.

The code below shows how two functions are defined in an extension:

[5]An entry point is returned by the operating system function loading the shared library into memory. How this is done is OS-dependent.

```
#include <math.h>
#include "defs.h"

/* ---- Function prototypes ---- */
Expr* Fibonacci(Elist& arglist); BType Fibonacci_types[]={INT_TYPE};
Expr* Exp(Elist& arglist); BType Exp_types[]={INT_TYPE};

/* ---- Function list and entry point ---- */

static FuncDescr functions[]={
    /* name        address    num of parms   parm types  */
    /* --------    ---------  ------------   ----------  */
    {"Fibonacci",  Fibonacci,     1,         Fibonacci_types},
    {"Exp",        Exp,           1,         Exp_types},
    0
};

static Contents contents={functions, 0, 0, 0};
Contents* EntryPoint() {return &contents;}

/* ---- Function implementations ---- */
long fib(long x) {
    return (x == 1 || x == 2)? 1 : fib(x-1)+fib(x-2);
}

Expr* Fibonacci(CL_List<Expr*>& arglist) {
    Expr* ex=arglist.GetFirst()->GetElement();
    long x=(long)((Int*)ex)->GetInt();
    return new Int(fib(x));
}

Expr* Exp(CL_List<Expr*>& args) {
    Expr* ex=args.GetFirst()->GetElement();
    long inp=((Int*)ex)->GetInt();
    long res=exp(inp);
    return new Int(res);
}
```

The sample extension defines functions `Fibonacci` and `Exp`. These will be available in the GOMscript interpreter and can be used like any other function (built-in or user-defined).

Every function has to accept a list of expressions, where each expression has to be downcast to the actual type, e.g. when an integer value is expected, the expression value has to be narrowed to a value of type integer.

Variable `contents` contains all functions, classes, and the entry- and exit-points of the shared library. The address of function `EntryPoint` is returned by the OS function that loads a shared library into memory. This address can then be dereferenced and executed

to obtain the address of struct `contents`. This is exactly what is done when GOMscript loads the shared libraries. Additionally, all function- and class-descriptions are added to GOMscript's internal function- and class-list, respectively.
Variable `functions`, which is an array of `FuncDescr` structs, is the first member of `contents` and contains all the description of all functions in the library. `FuncDescr` structs are used to associate the name of a function with its address, number of parameters and types.
`Fibonacci` is an example of a function. It expects one numeric argument. Since the number and types of parameters was provided above, the GOMscript runtime performs type-checking, and the argument can be directly converted to a C++ type (an `int`). The the result is computed and converted to a GOMscript numeric type, which is subsequently returned to GOMscript.

C.6.2 Writing an Extension for GOM

The ability of GOMscript to manage GOM instances is provided as a GOMscript extension (cf. file `GomObj.C`). Class `Gom` represents GOM proxies and contains methods corresponding to the ones defined in `GenObj` (cf. 4.7). Instances of `Gom` may be created, their attributes modified, and methods may be called interactively, triggering the corresponding actions on the target object.
The details of the GOM extension are not described here due to space limitations. However, registration of class `Gom` and its methods is done similarly to the way functions are registered (see above). The difference between classes defined in extensions and classes defined using GOMscript is hidden: built-in classes and user-defined ones can be manipulated in the same manner.

Bibliography

[ACH93] Sebastian Abeck, Alexander Clemm, and Ulf Hollberg. Simply Open Network Management. In INM [INM93], pages 361–375.

[ADF+94] Tom Atwood, Joshua Duhl, Guy Ferran, Mary Loomis, and Drew Wade. *The Object Database Standard: ODMG-93.* Morgan Kaufmann Publishers, release 1.1 edition, 1994.

[ANS93] Architecture Project Management, Poseidon House, Castle Park. Cambridge. *The ANSAware 4.1 manual set*, 1993.

[ASN90] ISO / IEC 8824. *Specification of Abstract Syntax Notation One (ASN.1)*, April 1990.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[Bac86] Maurice J. Bach. *The Design of the UNIX Operating System.* Prentice Hall, 1986.

[Ban96a] Bela Ban. Design of a CORBA-Based GOM Prototype. Internal paper, IBM Research Division, IBM Zurich Research Laboratory, Säumerstr. 4, 8803 Rüschlikon, June 1996.

[Ban96b] Bela Ban. Extending CORBA For Multi-Domain Management. In *Proceedings of Distributed Object-Oriented Computing For Telecom (DOCT'96) Workshop, ObjectWorld'96, Frankfurt, Germany*, 1996.

[Ban96c] Bela Ban. *GOMscript User's Guide.* IBM Corporation, 1996. Internal Paper.

[Ban96d] Bela Ban. Open Distributed Processing: A Reference Model For Distributed Computing. Technical report, Institute of Computer Science, University of Zurich, 1996.

[Ban96e] Bela Ban. Towards A Generic Object-Oriented Model For Multi-Domain Management. In Muehlhaeuser [Mue96], pages 272–276. Workshop Reader of the 10th European Conference on Object-Oriented Programming (ECOOP'96), Linz.

[Ban96f] Bela Ban. Towards an Object-Oriented Framework For Multi-Domain Management. Technical Report RZ 2789 (#89267), IBM Research Division, IBM Zurich Research Laboratory, Säumerstr. 4, 8803 Rüschlikon, February 1996.

[Ban96g] Bela Ban. Using Java For Dynamic Access to Multiple Object Models. Technical report, IBM Zurich Research Laboratory, October 1996.

[BD97] Bela Ban and Luca Deri. Static vs. Dynamic Network Management Based on CORBA. In *Proceedings of IS&N, Como*, May 1997.

[BF92] N. Borenstein and N. Freed. *MIME (Multipurpose Internet Mail Extensions): Mechanisms for Specifying and Describing the Format of Internet Message Bodies*, November 1992. RFC 1343.

[BGG94] Karim Berrah, David Gay, and Guy Genilloud. Accessing ANSA Objects from OSI Network Management. In *Proceedings of the Fifth IFIP / IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'94)*. IFIP / IEEE, 1994. Toulouse, France.

[Bir96] Kenneth P. Birman. *Building Secure and Reliable Network Applications*. Manning Publications Co., 1996.

[BK96] Nat Brown and Charlie Kindel. Distributed Component Object Model Protocol – DCOM/1.0. Internet draft, Microsoft Corporation, November 1996.

[Bla92] Uyless Black. *Network Management Standards. The OSI, SNMP and CMOL Standards*. Mc Graw-Hill, 1992.

[BLFF96] T. Berners-Lee, R. Fielding, and H. Frystyk. *Hypertext Transfer Protocol – HTTP/1.0*, May 1996.

[BN84] A. D. Birrell and B. J. Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2:39 – 59, February 1984.

[Box95] Don Box. Building C++ Components Using OLE2. *C++ Report*, 7(3):28–34, April 1995.

[Bro94] K. Brockschmidt. *Inside OLE2*. Microsoft Press, Redmont, WA, 1994.

[Cat91] Roderic G. G. Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley, 1991.

[CB94] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security*. Addison-Wesley, 1994.

[CFSD90] J. Case, M. Fedor, M. Schoffstall, and C. Davin. The Simple Network Management Protocol. RFC 1157, May 1990.

[CGI94] NCSA. *The Common Gateway Interface*, 1994. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html.

[Cha93] Chang. ISO/CCITT to Internet Management Proxy. Technical Report Issue 1.0, Network Management Forum, October 1993.

[Cha94] Siva Challa. *Towards an Interoperable, Reflective Common Object Model for Statically-Typed Object-Oriented Languages.* PhD thesis, Dept. of Computer Science, Blacksburg, Virginia, 1994.

[CMI] International Standards Organization. *Common Management Information Protocol (CMIP).* ISO / IEC 9596.

[CMRW96] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser. Structure of Management Information for Version 2 of the Simple Network Management Protocol (SNMPv2). RFC 1902, January 1996.

[Cop92] James O. Coplien. *Advanced C++ Programming Styles and Idioms.* Addison-Wesley, 1992.

[COS95] Object Management Group. *CORBAservices: Common Object Services Specification*, document 95-3-31 edition, March 1995. Revised Edition.

[Cra93] Daniel H. Craft. A study of pickling. *Journal of Object-Oriented Programming*, 5(8):54–66, January 1993.

[Der96] Luca Deri. Surfing Resources Across the Network. Technical report, IBM Zurich Research Laboratory, 1996.

[Der97] Luca Deri. *A Component-based Architecture for Open, Independently Extensible Distributed Systems.* PhD thesis, University of Berne, 1997.

[DHR92] De Meer, J., V. Heymer, and R. Roth, editors. *Open Distributed Processing.* IFIP Transactions C-1. Elsevier Science Publishers (North-Holland), 1992.

[DSO96] *Proceedings of the Seventh IFIP / IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'96).* IFIP / IEEE, 1996. L'Aquila, Italy.

[GDM92] ISO / IEC 10165-4. *Guidelines for the Definition of Managed Objects*, 1992.

[Gen96] Guy Genilloud. *Towards A Distributed Architecture For Systems Management.* PhD thesis, Ecole Polytechnique Federale de Lausanne, 1996. Thesis No. 1588.

[GG95] Genilloud Guy and David Gay. Accessing OSI Managed Objects From ANSAware. In *Proceedings of the Sixth IFIP / IEEE International Workshop on Distributed Systems: Operations and Management (DSOM'95).* IFIP / IEEE, 1995. Ottawa, Canada.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GMR94] G. Geiger, A. Majtenyi, and P. Reder. IBM cmipWorks. Technical report, IBM Corporation, March 1994.

[GP95] Guy Genilloud and Marc Polizzi. Managing ANSA Objects with OSI Network Management Tools. Technical report, EPFL, 1995.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80. The Language.* Addison-Wesley, 1989.

[Gro93] Mark Grossman. Object I/O and runtime type information via automatic code generation in C++. *Journal of Object-Oriented Programming*, 6(4):34–42, July 1993.

[Hie94] Juan J. Hierro. Architectural Issues for Using CORBA Technology in OSI Systems Management. Append to XoJIDM mailing list, July 1994.

[Hie96a] Juan Hierro. Management information repository. Submitted by Telefonica I&D to the NMF – X/Open Joint Inter-Domain Management Taskforce as proposal for Interaction Translation, March 1996.

[Hie96b] Juan J. Hierro. Common Facilities for OSI Management. Submitted to XoJIDM mailing list, February 1996. Submitted as proposal for XoJIDM Interaction Translation.

[HPSK96] James Won-Ki Hong, Jong-Tae Park, Joong-Gu Song, and Sung-Bum Kim. Implementation and Performance of a TMN SMK System. In DSOM [DSO96]. L'Aquila, Italy.

[IBM96] IBM Corporation. *IBM SOMobjects Java Client*, 1996.

[INM93] *Information Systems Integrated Network Management III.* Elsevier North-Holland, 1993.

[Int95] Joint Inter Domain Management Working Group, X/Open and Network Management Forum. *Inter Domain Management Specifications: Preliminary CORBA / CMISE Interaction Translation Architecture*, April 1995. http://www.rdg.opengroup.org/mem_only/tech/sysman/jidm/it.htm.

[Ion96] Iona Technolgies. *OrbixWeb For Java*, 1996.

[ITU92a] ITU-T — ISO/IEC. *ITU-T X.700 — ISO/IEC 7498-4: Management Framework For Open Systems Interconnection*, September 1992.

[ITU92b] ITU-T — ISO/IEC. *ITU-T X.701 — ISO/IEC 10040: Information Technology - Open Systems Interconnection - Systems Management Overview*, 1992.

[ITU92c] ITU-T — ISO/IEC. *ITU-T X.720 — ISO/IEC 10165-1: Information Technology - Open Systems Interconnection - Structure Of Management Information: Management Information Model*, January 1992.

[ITU93] ITU-T. *ITU-T X.734: Data Communication Networks. Information Technology – Open Systems Interconnection – Systems Management: Event Report Management Function*, 1993. Recommendation X.734.

[Joe96] Sun Microsystems. *Joe: Developing Client/Server Applications for the Web*, 1996.

[Kam90] Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach.* Addison-Wesley, Reading, MA, 1990.

[KQM95] *Knowledge Query and Manipulation Language*, 1995. http://www.cs.umbc.edu/kqml/kqml95/kqml95.html.

[KS93] Pramod Kalyanasundaram and Adarshpal S. Sethi. An Application Gateway Design for OSI-Internet Management. In INM [INM93], pages 389–401.

[LaB93a] Lee LaBarre. Translation of Internet MIB-II (RFC1213) to ISO/CCITT GDMO MIB. Technical Report Issue 1.0, Network Management Forum, October 1993.

[LaB93b] Lee LaBarre. Translation of Internet MIBs to ISO/CCITT GDMO MIBs. Technical Report Issue 1.0, Network Management Forum, October 1993.

[LF93] Allan Leinwand and Karen Fang. *Network Management. A Practical Perspective.* Addison-Wesley, 1993.

[LS88] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. *ACM SIGPLAN Notices*, 23(7), July 1988.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, pages 147–155, December 1987. Published as Proceedings OOPSLA '87, ACM SIGPLAN Notices, volume 22, number 12.

[Maf95] Silvano Maffeis. *Run-Time Support for Object-Oriented Distributed Programming.* PhD thesis, Institute of Computer Science, University of Zurich, 1995.

[Mas97] Introduction to Mascotte. White Paper, May 1997. Esprit Project: 20804, version 2.0.

[Maz96] Subrata Mazumdar. Inter-Domain Management between CORBA and SNMP: CORBA/SNMP Gateway Approach to WEB-based Management. In DSOM [DSO96]. L'Aquila, Italy.

[MBL93] Subrata Mazumdar, Stephen Brady, and David L. Levine. Design of Protocol Independent Management Agent to Support SNMP and CMIP Queries. In INM [INM93], pages 377–389.

[ME96] T. Magedanz and T. Eckardt. Mobile Software Agents: A New Paradigm for Telecommunications Management. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 360–369, Kyoto, Japan, April 1996. IEEE Press. ISBN 0-7803-2518-4,.

[Mey88] Betrand Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[MGG96] Philippe Merle, Christoph Gransart, and Jean-Marc Geib. CorbaScript and CorbaWeb: A Generic Object-Oriented Dynamic Environment upon CORBA. Technical report, Universite de Lille, 1996.

[MOA94] IBM Corporation. *The Managed Object Agent Composer User's Guide*, 1994.

[MPBS95] Kevin McCarthy, George Pavlou, Saleem Bhatti, and Jose N. De Souza. Exploiting the power of OSI Management for the control of SNMP-capable resources using generic application level gateways. In *Information Systems Integrated Network Management IV*, pages 440–453. Elsevier North-Holland, 1995.

[MQS95] IBM Corporation. *MQSeries*, 1995. http://www.hursley.ibm.com/mqseries/.

[Mue96] Max Muehlhaeuser, editor. *Special Issues in Object-Oriented Programming.* dpunkt.verlag, 1996. Workshop Reader of the 10th European Conference on Object-Oriented Programming (ECOOP'96), Linz.

[New93] Owen (ed.) Newman. Translation of ISO/CCITT MIBs to Internet MIBs. Technical Report Issue 1.0, Network Management Forum, October 1993.

[Obj95] Object Management Group. *Object Property Services*, June 1995. OMG TC Document 95-6-1.

[Obj97] Object Management Group. *Notification Service RFP (Telecom RFP3)*, March 1997. Document 97-01-03.

[ODP95] ITU-T — ISO/IEC. *ITU-T X.901 — ISO/IEC 10746-1: ODP Reference Model Part1: Overview*, draft international standard edition, May 1995.

[OMG95] Object Management Group. *The Common Object Request Broker: Architecture And Specification*, July 1995. Revision 2.0.

[Pav93] G. Pavlou. The OSIMIS TMN Platform: Support for Multiple Technology Integrated Management Systems. In *Proceedings of the 1st RACE IS&N Conference*, Paris, November 1993.

[PHHS96] Jong-Tae Park, Su-Ho Ha, James W. Hong, and Joong-Goo Song. Design and Implementation of a CORBA-based TMN SMK System. In *Proceedings of NOMS96*, 1996.

[QP93] Peggy Quinn and George Preoteasa. Reconciling Object Models for Systems and Network Management. Technical report, UNIX Systems Laboratories, Inc., 1993.

[RFC91a] OSI Internet Management, April 1991.

[RFC91b] Management Information Base for Network Management of TCP/IP-based Internets: MIB II, March 1991.

[RM88] M. Rose and K. McCloghrie. Structure and Identification of Management Information for TCP/IP-based Internets. RFC 1065, August 1988.

[RMI96] Sun Microsystems Inc. *Java Remote Method Invocation Specification*, 1.1 edition, November 1996. Draft.

[Ros90a] Marshall T. Rose. *The Open Book. A Practical Perspective on OSI*. Prentice Hall, 1990.

[Ros90b] Marshall T. Rose. Transition and Coexistence Strategies for TCP/IP to OSI. *IEEE Journal on Selected Areas in Communications*, 8(1):57 – 66, January 1990.

[Rut93] Tom Rutt. Comparison of the OSI management, OMG and Internet management Object Models. Technical report, AT&T Bell Labs, December 1993.

[Sat96] Hirano Satoshi. *HORB: Distributed Execution of Java Programs*, 1996.

[SG94] Ute Schürfeld and Dieter Gantenbein. Bilingual Agent. DSOM Access to X.700 Agent. Technical report, IBM Zurich Research Laboratory, May 1994.

[Sha86] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *International Conference On Distributed Computing Systems*, Massachusetts, Boston, May 1986.

[Shi94] John J. Shilling. How to roll your own persistent objects in C++. *Journal of Object-Oriented Programming*, 7(4):25–32, July 1994.

[Slo94] Morris Sloman, editor. *Network and Distributed Systems Management*. Addison-Wesley, 1994.

[SOM94] IBM. *SOM Developer's Toolkit: An Introductory Guide To The System Object Model And Its Accompanying Frameworks*, 1994.

[Sou94a] Nader Soukouti. Inter Domain Management. Append to XoJIDM mailing list, 1994. draft.

[Sou94b] Nader Soukouti. Managing CORBA Objects Using an OSI Manager. Append to XoJIDM mailing list, 1994.

[Sou94c] Nader Soukouti. Managing OSI Objects Using a CORBA Manager. Append to XoJIDM mailing list, 1994. draft.

[Sou94d] Nader Soukouti. Toward Managing CORBA Objects Via OSI Network Management Mechanisms. Append to XoJIDM mailing list, July 1994. draft.

[Sou94e] Nader Soukouti. Using CORBA Technology for SNMP Management. Append to XoJIDM mailing list, 1994. draft.

[Spe97] Joint Inter-Domain Management Working Group. *Inter-Domain Management: Specification Translation*, April 1997. http://www.rdg.opengroup.org/mem_only/tech/sysman/jidm/st.htm.

[Ste90a] Guy L. Steele. *Common LISP. The Language.* Digital Press, 1990.

[Ste90b] W. Richard Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 1991.

[Sun95] Sun Microsystems. *The Java Language Environment: A White Paper*, 1995.

[Sun96] Sun Microsystems Inc. *Java Core Reflection. API and Specification*, October 1996.

[SV97] Douglas C. Schmidt and Steve Vinoski. The OMG Events Service. *C++ Report*, 9(2):37 – 46, February 1997.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 1992.

[TBLP92] J. Groff T. Berners-Lee, R. Cailliau and B. Pollermann. World-Wide Web: The Information Universe. *Electronic Networking*, 1(2), 1992.

[Tcl95] West Virginia University. *Tcl Dynamic Invocation Interface*, 1995. http://www.cerc.wvu.edu/dice/iss/TclDii/TclDii.html.

[Tel96] *OMG White Paper: CORBA-Based Telecommunication Network Management System*, May 1996.

[TIN95] Telecommunications Information Networking Architecture Consortium. *Overall Concepts And Principles Of TINA*, 1.0 edition, Feb 1995.

[Tiv95] Tivoli Systems Inc. *Object-Orientation Brings Advantages To Distributed Systems Management*, 1995.

[TMN95] ITU-T. *M.3010: Principles of Telecommunication Management Network*, 1995.

[TMN96] *TMN/C++*, 1.0 draft 7a edition, May 1996. cmiswg@nmf.org.

[UCCH91] Dave Ungar, Craig Chambers, Bay-Wei Chang, and Urs Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation: An International Journal*, 4(3), 1991.

[US91] Dave Ungar and Randall B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation: An International Journal*, 4(3), 1991.

[Weg90] Peter Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, 1(1):7 – 87, August 1990.

[Whi94] J. E. White. *Telescript Technology: The Foundation for the Electronic Marketplace*. General Magic, 1994.
http://www.genmagic.com/WhitePapers.

[WMBL92] S. F. Wu, S. Mazumdar, S. Brady, and D. W. Levine. On Implementing a Protocol Independent MIB. Technical Report RC 18248, IBM T. J. Watson Research Center, Yorktown Heights, NY, August 1992.

[X7593] ITU-T. *X.750: Management Knowledge Management Function*, 1993.

[X/O96] X/Open Company Ltd. *Systems Management: Event Management Service*, draft v0.3 edition, June 1996. Preliminary Specification.

[XOM94] X/Open Company Ltd. *OSI Abstract Data Manipulation API (XOM)*, c315 edition, February 1994.

Glossary and Acronyms

The glossary defines terms and acronyms. Where appropriate, a section number at the end refers to the location where the term or acronym is used or defined.

Adapter Converts between generic- and specific object models. 4.4.

AE-title Application entity title. Used to denote the address of an OSI manager or agent.

Agent Entity in the OSI model which represents resources for the purpose of management. 2.2.1, 1.2.

ANSA Advanced Networked Systems Architecture. Another model for creation of distributed applications, similar to CORBA, but predating it.

ASN.1 Abstract Syntax Notation One. 2.2.3.

Bridge See Adapter.

CGI Common Gateway Interface. Used by HTTP servers to invoke external programs for request handling.

Client Program requesting services from a server. 1.2.

CMIP Common Management Information Protocol. Used between OSI managers and agents. Used synonymously with 'OSI' in a pars pro toto fashion in this thesis. 2.2.1.

COM Component Object Model. Microsoft's core document-centric model underlying OLE. See DCOM, OLE.

CORBA Common Object Request Broker Architecture. 2.1.

CORBA services See COSS.

COSS Common Object Services Specification. Set of IDL interfaces defining frequently used services in the CORBA world. Used to extend the functionality of CORBA without modification of the core architecture.

DCE Distributed Computing Environment. Open Software Foundation's procedural distribution model.

DCOM Distributed Component Object Model. See COM. 1.3.7.

DII See Dynamic Invocation Interface. 2.1.

DIR Dynamic Invocation Routine. Part of CORBA's DSI. This routine can be installed by the programmer and is called every time an implementation (e.g. a server) receives a request.

Distinguished name (DN) Name to identify a managed object in an OSI agent. Consists of several relative distinguished names (RDNs). 2.2.4.

Dynamic Invocation Interface (DII) Used in the CORBA model to dynamically dispatch requests to objects. Used extensively by GOM. 2.1.

Dynamic Skeleton Interface (DSI) Equivalent of DII on the implementation side. Allows to handle requests received by a CORBA implementation in a dynamic and generic manner. 2.1.

Domain Used in the meaning given by XoJIDM [Spe97]: (object) model such as CMIP, CORBA, SNMP etc. 3.1.

DSI See Dynamic Skeleton Interface.

DSOM Distributed System Object Model. IBM's CORBA implementation.

EFD Event Forwarding Discriminator. 4.5.1.2.

GDMO Guideline for the Definition of Managed Objects. OSI-defined language to specify managed object classes. Equivalent in the CORBA world is IDL. 2.2.

GIOP General Inter-ORB Protocol. Defined by OMG. Used for communication between different ORB implementation. The version of TCP/IP is called IIOP. 4.4.3.

HTTP Hypertext Transfer Protocol. Used in the WWW between browsers and HTTP servers.

IAB Internet Architecture Board.

IDL Interface Definition Language. Used to specify CORBA classes. 2.1.3.

IDM See Inter-Domain Management. 3.1.

IIOP Internet Inter-ORB Protocol. See GIOP. 4.4.3.

Inter-Domain Management Access of model B from model A in a transparent way: model B is translated to A. 3.1.

Interface (a) Set of operations offered by a class. (b) CORBA class (interface). In unclear cases the term (OMG) 'IDL interface' is used for disambiguation.

IOR Interoperable Object Reference. CORBA object reference used to ensure uniqueness among several ORBs. See IIOP.

IR Interface Repository. Used in CORBA to store metadata information. 2.1.1.

ISO International Standards Organization.

ITU-T International Telecommunications Union.

JIDM Joint Inter-Domain Management task force. Subgroup within X/Open working on CORBA-CMIP interoperability. 3.1.2.

KQML Knowledge Query and Manipulation Language. A proposal for electronic representation of knowledge. 5.2.

Language binding Code in a specific language generated from CORBA IDL interfaces. Used by clients to access and by servers to implement functionality (i.e. objects). 4.2.4.

Manager Entity in the OSI world which accesses services offered by agents. 1.2.

Metadata Layout A layout defines which elements are available in an object model, the semantics of each element and the relations between elements, e.g.: a *class* element is a template for the creation of instances (definition and semantics) and contains *attribute*- and *operation* elements (relation). 4.2.3.2.

MIB Management Information Base. Denotes the set of classes in the OSI- and SNMP models that comprise a certain domain, e.g. a MIB for ATM, for Internet management (MIB II) etc. Sometimes also used in the OSI world to describe the set of instances in an OSI agent (see also MIT). 2.2.

MIME Multipurpose Internet Mail Extensions. Standard for attaching different sorts of data to a mail message, defined in RFC 1343 [BF92]. 6.

MIR Management Information Repository. 4.3.3.2.

MIT Management Information Tree. The containment tree of managed objects in an OSI agent. 2.2.

MO Managed Object. Instance in an OSI agent representing a GDMO class. 2.2.1.

MR Metadata Repository. Central piece of GOM which stores metadata about target models (CORBA, CMIP and SNMP). 4.3.2.

Network management Set of tasks dealing with management of network-specific devices, such as routers, printers etc. Subset of systems management.

NMF Network Management Forum.

Object reference Client's local proxy instance (handle) to a (possibly) remote CORBA instance. Any request invoked on it is transparently forwarded to the target instance. Whether the target instance is local, in a different process, or on a different machine, is transparent. 2.1.1.

ODL Object Definition Language. Superset of OMG IDL, used in TINA and ODP. 1.3.6.

ODP Open Distributed Processing. Framework standardized by OSI to describe distributed systems.

OID Object Identifier. Used in SNMP to denote variables (SNMP) and in the OSI model to denote GDMO classes, ASN.1 types etc. 4.6.1.

OLE Object Linking and Embedding. Microsoft's compound document architecture. 4.1.1.

OMG Object Management Group. Industry consortium to develop the CORBA standard.

OODP Object-Oriented Distributed Processing.

OQL Object Query Language. 4.5.2.4.

ORB Object Request Broker. Central distributed message bus of CORBA. 2.1.1.

OSF Open Software Foundation.

OSI Open Systems Interconnection. ISO's 7-layer standard for communication between open systems. See CMIP.

PDU Protocol Data Unit.

PIM Protocol-Independent MIB (see [WMBL92]). 3.1.1.

Proxy object A local instance of `GenObj` acting as place-holder for a (remote) target instance. All requests sent to a proxy will be transparently forwarded to its corresponding target object. 4.1.1.

Reification Representation of concepts of a system by elements of the system itself. For example, concepts such as method dispatching or inheritance can be modeled and implemented using objects (Object-Oriented Reification). 4.2.2.

RDN Relative Distinguished Name. Parts of a Distinguished Name. 2.2.4.

Relative Distinguished Name (RDN) See RDN.

RFC Request For Comments. Proposal in the Internet community for a new standard.

RMI Remote Method Invocation. Java's mechanism of dispatching methods to remote objects. 1.3.7.

RPC Remote Procedure Call. See [BN84].

RTTI Run Time Type Identification. Used in C++ to maintain a certain degree of type information about an object. Mainly used for safe narrowing of instances. 4.2.4.1.

Server Program providing services to clients. 1.2.

SMF Systems Management Functions. OSI's equivalent of CORBA's services. 2.2.

SMK Shared Management Knowledge. OSI SMF, maintaining information about classes and instances of an OSI agent. 4.3.3.1.

SNMP Simple Network Management Protocol. 2.3.

Systems management Set of tasks dealing with application management, e.g. user administration, starting- and shutting down of servers, software distribution etc.

Target instance Instance in a target system, e.g. a managed object in an OSI agent, a CORBA instance in a CORBA server, or a variable in an SNMP agent. 1.2.

Target system (or specific system) A system that is to be managed by GOM (e.g. a CMIP/SNMP agent or a CORBA server). 1.2.

TINA Telecommunications Information Networking Architecture. Standard for systems management based on ODP.

TMN Telecommunications Management Network. Standard refining the OSI network management standards.

URL Universal Resource Locator. Symbolic address of a Web page.

XEMS Extended Event Management Services. Standard defined by X/Open for event management. 4.5.1.4.

Curriculum Vitae

Personalia

Last name	Ban
First name	Bela
Address	Stählistr. 33a 8280 Kreuzlingen Switzerland
Nationality	Swiss
Date of birth	Jan 26 1965
Place of birth	Münsterlingen (Switzerland)
Email	`BelaBan@acm.org`

Education

1972–1978	Primary school, Kreuzlingen
1978–1980	Secondary school, Kreuzlingen
1980–1984	Gymnasium (Type B), Kreuzlingen
1985	Military Service (17 weeks)
1986–1992	University of Zurich Studies in English, Computer Science and Economics Graduation in 1993, summa cum laude
1988	Exchange student Detroit, MI, and Sta. Cruz, CA (8 months)
1993–1997	Ph.D. candidate University of Zurich and IBM Zurich Research Laboratory

Work Experience

1988–1991	IBM Switzerland, Zurich (SW engineer)
1992–1993	ISE AG, Tägerwilen, Switzerland (SW/Knowledge engineer)
1993–1997	IBM Research Laboratory, Rüschlikon, Switzerland 60% Pre-Doc contract Work on IBM's TMN/6000 OSI agent product

Publications

- Ban, Bela and Luca Deri. Abstract Factory Pattern Revisited. Research Report RZ2787 (#89265). IBM Research Division, IBM Zurich Research Laboratory, Rüschlikon, 1996.

- Ban, Bela. Towards a Generic Object-Oriented Model for Multi-Domain Management. Workshop reader, *10th European Conference on Object-Oriented Programming (ECOOP'96)*, Linz, Austria (dpunkt, CITY, 1996).

- Ban, Bela. Extending Corba for Multi-Domain Management. In *Proceedings of Distributed Object-Oriented Computing For Telecom (DOCT'96)*, ObjectWorld'96, Frankfurt.

- Ban, Bela. Using Java for Dynamic Access to Multiple Object Models. Internal Research Paper. IBM Research Division, IBM Zurich Research Laboratory, Rüschlikon, 1996.

- Ban, Bela. Towards an Object Oriented Framework for Multi Domain Management. Research Report RZ2789 (#89267). IBM Research Division, IBM Zurich Research Laboratory, Rüschlikon, 1996.

- Ban, Bela. Design of a CORBA-Based GOM Prototype. IBM Research Division, IBM Zurich Research Laboratory, Rüschlikon, 1996.

- Ban, Bela. *GOMscript User's Guide.* IBM Research Division, IBM Zurich Research Laboratory, Rüschlikon, 1996.

- Ban, Bela. Open Distributed Processing: A Reference Model for Distributed Processing. Technical Report. Institute of Computer Science, University of Zurich, 1996.

- Ban, Bela and Luca Deri. Static vs. Dynamic CMIP/SNMP Network Management Using CORBA. In *Intelligence in Services and Networks: Technology for Cooperative Competition*, ed. by A. Mullery et al. (Springer, Berlin, 1997).