3. Mapping the OSI-SM / TMN Model Onto Object-Oriented Programming Environments

3.1 Introduction

Chapter 3 of this thesis proposes a novel approach for the realisation of the OSI-SM/TMN framework, based on object-oriented software platforms. While the TMN is object-oriented in information specification terms, it is a communications framework and, as such, it does not address software realisation aspects. The richness and complexity of the overall framework in conjunction to the fact that non object-oriented approaches were initially adopted for its realisation, resulted in doubts about its implementability, performance and eventual deployment.

In this chapter we demonstrate how the inherent object-oriented aspects of the OSI-SM/TMN framework can be exploited through an object-oriented realisation model that hides protocol aspects through abstractions similar to those of emerging distributed systems frameworks. The resulting environment is an easy to use object-oriented distributed software platform that enables the rapid development and deployment of TMN systems. The software architecture of the proposed environment is presented while its power, expressiveness, usability and similarity to recently emerging distributed object frameworks is demonstrated through examples. The environment in which the relevant concepts and abstractions were validated is the OSIMIS TMN platform which predated similar products by some years and influenced a number of subsequent commercial developments.

Having demonstrated the mapping of the abstract OSI-SM/TMN framework to object-oriented programming environments in the form of an object-oriented distributed software platform, we subsequently demonstrate that the resulting framework has good performance characteristics. We demonstrate in particular that the main performance cost is due to the Q_3 protocol stack rather than the proposed application framework. This is particularly important since we show in Chapter 4 that it is possible to retain the TMN application aspects over a distributed object framework such as OMG CORBA.

Chapter 3: Mapping the OSI-SM /TMN Model Onto
Object-Oriented Programming Environments

This chapter is organised as a "super-chapter", in a similar fashion to chapters 2 and 4 of this thesis. Related research work is presented in the various sub-sections before the author's research work, in a similar style to the rest of the thesis.

Section 3.2 presents first an introduction to object-oriented software systems and subsequently identifies a number of key properties of object-oriented distributed software frameworks against which the proposed OSI-SM/TMN realisation framework will be measured.

Section 3.3 presents key issues in realising the protocol part of the Q_3 interface, discusses possible policies for the relevant API and investigates alternative mappings over different, lightweight transport mechanisms. Section 3.4 discusses issues behind object-oriented ASN.1 manipulation, which is key to any OSI upper layer infrastructure and an essential ingredient of the proposed TMN application framework.

Section 3.5 discusses the manager or client mappings of the proposed TMN application framework. Two approaches are presented, one modelling whole remote agents and another one modelling individual managed objects. The latter includes a manager mapping of GDMO to O-O programming languages. A mapping to the Tcl/Tk interpreted scripting language is also presented, being suitable for the rapid realisation of TMN WS-OS applications.

Section 3.6 discusses the agent or server mappings of the proposed TMN application framework. It proposes an agent mapping of GDMO to O-O programming languages, discusses interaction models between managed objects and associated resources, presents realisation aspects of the OSI-SM SMFs and shows that the perceived "difficult" aspects of the OSI-SM/TMN framework, i.e. scoping, filtering, event reporting and logging, are in fact easy to realise.

Section 3.7 discusses aspects of synchronous "remote procedure call" and asynchronous "message passing" paradigms, which are both supported in the proposed environment. Section 3.8 presents a performance analysis and evaluation in terms of response times, application sizes and the amount of management traffic incurred.

Section 3.9 examines the proposed framework against the desired properties of object-oriented distributed frameworks identified in section 3.2. It also shows how the functional decomposition of the TMN OS presented in Chapter 2 is mapped onto the proposed object-oriented realisation framework. Since the ultimate validation of the latter was accomplished through research and development work based on the proposed environment, such work is presented in Appendix A.

Finally, section 3.10 highlights the research contributions in this chapter.

3.2 Object-Oriented Distributed Systems

One of the key contributions of this thesis is that it demonstrates how the OSI-SM / TMN model can be mapped onto object-oriented programming environments using abstractions similar to those of emerging object-oriented distribution frameworks. It is thus important to define first the terms *object-oriented programming environment* and *object-oriented distribution framework*.

3.2.1 Object-Oriented Development Principles

Object-orientation has been a cultural achievement of software engineering in the mid and late eighties. It proposes a new approach for specifying, designing and implementing software systems which takes further the structured approach of the past and achieves new levels of software reusability, extensibility and genericity. Object-oriented concepts and principles have already been mentioned when describing the OSI-SM information model in section 2.2.3 of Chapter 2. Here we attempt a more systematic definition.

In traditional or *structured* software engineering, programs comprise data structures and logic which are loosely coupled. Designers and programmers think of their programs in terms of the required logic first and add data structures later in order to support the needs of that logic. The relevant data structures are globally available and accessible by different program procedures which manipulate them. Program logic is developed by "stepwise refinement" [Wirth71] while the basic building block is the procedure. A typical programming language that supports this paradigm is Pascal [Wirth75].

An evolution of the structured approach has led to the *modular* paradigm. A module implements an abstraction that becomes the basic building block of complex programs. A module has well defined functionality, e.g. it implements an abstract data type such as a linked list, and comprises both procedures and data, in a similar fashion to a modular program. Data is hidden inside the module so that it becomes invisible to procedures of other modules. This principle is known as data-hiding or encapsulation and guarantees the internal consistency and integrity of the module. The module's functionality is made available to other modules through well-defined entry points, implemented as "public" procedure calls.

A module can be thought as some form of object since it supports encapsulation. In fact, some refer to this approach as *object-based*. A key drawback is that a module may only have one "instance" since it contains a single copy of the private data. In addition, a module's functionality cannot be modified or extended without having access to its source code. As such, this approach

supports only limited reusability and extensibility. Programming languages that support this paradigm are C [Kern78], Modula [Wirth82] and Ada. It should be noted that the C programming language supports this paradigm implicitly only, since there are no explicit language constructs to support modules.

The evolution of the modular framework has led to the *object-oriented* paradigm. An object is similar to a module since it contains procedures and data, but the two are tightly coupled and constitute an object type or *object class*. The procedures of an object are known as *methods* and its data as *variables*. Many *instances* of the same class may exist at any time, with their own copies of instance variables. Access to the latter is allowed only through an object's methods. Some of those are *private* and cannot be accessed from outside. *Public* methods can be accessed by other objects in order to perform certain functions. A method may change the state of an object, operate on some of its variables or act on other objects. In an object-oriented system, all interactions among object instances take place through method calls or *messages*. This paradigm provides better support for software reusability and system integrity.

The concept of an object is taken further through inheritance and polymorphism. *Inheritance* allows a new class, called a *subclass*, to be an extension, modification or even restriction of the original class, called the *superclass*. A subclass may include additional methods not present in the superclass (extension), may override existing ones (modification) or may even prevent existing ones from exercising their functionality (restriction). These features can be supported without access to the source code of a superclass and provide excellent support for software re-usability and extensibility.

Polymorphism is an intriguing characteristic which enables one to treat instances of derived classes as instances of a generic superclass (from the Greek words *poly:* multi and *morphe*: shape or form). A typical example demonstrating the use of polymorphism is that of a window manager object which treats displayed objects in the same way, regardless of their particular specialisation e.g. word processor window, task-bar, pointer etc. The window manager is programmed to interact with instances of a generic class, e.g. displayableObject, which could be moved, resized, iconified, brought forward or backward and so on. It can then interact with instances of particular specialisations of that class and trigger associated behaviour without even knowing what the relevant classes are.

Programming languages that support the object-oriented paradigm are Smalltalk [Gold83], C++ [Strau86], Objective C [Cox86], Eiffel [Meyer88] and more recently Java [Sun96]. The most popular of those is C++ because of its compatibility with C and the fact it is highly efficient.

Polymorphism in C++ is supported by *virtual* methods which may be redefined in derived classes. A call to virtual method results in triggering the *leaf-most* method implementation in the inheritance hierarchy of that instance, despite the fact that the caller "sees" the latter as an instance of a generic superclass. This feature achieves polymorphic behaviour.

Object-oriented programming should be based on a sound object-oriented design. The latter breaks away from the structured and even modular design practices and proposes a new approach to the decomposition of complex systems. There exist a number of books addressing object-oriented decomposition methodologies. [Booch91] and [Rumb91] are the classical references, discussing both issues of object-oriented decomposition and proposing modelling techniques for documenting an object-oriented design, the Object-Oriented Design (OOD) and the Object Modelling Technique (OMT) respectively. [Cox86] and [Meyer88] address mainly O-O programming languages, Objective C and Eiffel respectively, but they also contain useful material on object-oriented design. This thesis uses OMT, C++ class specifications and object instance diagrams to demonstrate aspects of object-oriented design.

The OSIMIS platform, which is the environment in which the ideas presented in this thesis have been validated, was designed using object-oriented design principles and making extensive use of concepts such as inheritance and polymorphism. The goal behind the design was to allow reusability, extensibility and access to sophisticated features through simple-to-use object-oriented APIs. The approach was baptised *harness-and-hide* [Pav94b]. C++ [Strau86] was chosen as the programming language, the reasons being at the time (1989) compatibility with C, ubiquity, strong type checking and performance.

3.2.2 Object-Oriented Distribution Frameworks

Distributed systems have been addressed since the early eighties by the research community and have become a reality since the mid to late eighties through the advent of local area networks and inexpensive workstations and personal computers. Distributed systems exhibit component remoteness, component concurrency, lack of precisely determinable global state and potential for partial failures. On the other hand, they offer potential advantages in availability, performance, dependability and cost optimisation resulting from distribution. A key issue in distributed systems is masking the heterogeneity of the hardware, operating systems and programming languages used to build them. [Coul88] addresses the concepts and design of distributed systems in detail while [Kram94] provides a concise introduction to the relevant issues. It should be noted that most of the literature on distributed systems assumes silently a highly reliable and "fast" local

area network as the supporting communications infrastructure. This is not the case with a TMN which can be distributed over a wide area network, with parts of it communicating over slower, less reliable links.

Since the early days of research in distributed systems, a key requirement has been the extension of programming languages with constructs to support distributed computation. There exist two different paradigms for those extensions: unidirectional asynchronous message passing or bi-directional synchronous Remote Procedure Call (RPC), the latter having semantics similar to a local procedure call. There exist both differences and complementary aspects in the two approaches which are discussed in more detail in section 3.7 of this thesis.

The RPC paradigm is described in the seminal [Birr84]. Since its inception, a number of distribution frameworks based on it appeared, providing support for the development and deployment of distributed systems. Sun Microsystems' RPC [Sun88] comes bundled with their SunOS and Solaris operating systems and has been widely used. The UCL RPC environment is described in [Wilb87] and included a binding service to support location transparency, introducing aspects of an elementary platform. The ANSA platform [ANSA89a] introduced the concept of trading and was more than an RPC environment, influencing the development of the whole ODP framework [ODP]. The OSF DCE was a industrial approach, bearing more similarities to the first two systems than to ANSA. OMG CORBA [CORBA] is another, more recent, industrial approach, embracing for the first time true object-orientation. The author has experimented with all of those frameworks apart from DCE. The ANSA, DCE and CORBA frameworks are examined in more detail in Chapter 4 of this thesis.

Since we will later need to evaluate the proposed OSI-SM/TMN realisation approach against the properties of *object-oriented distribution frameworks*, it is important to define what these properties are. In an ideal distribution framework, one could take a non-distributed object-oriented program, derive abstract specifications for the object interfaces, produce distributed "stub" objects through relevant tools and re-use most of the existing implementation to fill-in the stub objects with behaviour. After this reverse-engineering process, the system could be deployed in a distributed fashion.

The observant and cognisant reader may remark that this is what Java's [Sun96] Remote Method Invocation (RMI) mechanism tries to achieve, without the need for an intermediate step of abstract interface specification. The latter is exactly the point: the Java RMI assumes an homogeneous environment where all distributed objects are programmed in Java. A distribution framework should mask the heterogeneity of components, acting as a unifying "glue". It should

also *allow* for it in the first place and be able to *cope* with it. This is exactly what the Java RMI does not do and this is why it is not considered as a distribution framework in this thesis. It should be noted that distributed objects could be programmed in Java in any other framework.

Distributed objects need to be specified in an abstract language, which should be programming language independent. That language should be object-oriented, supporting inheritance and polymorphism, since the latter are key properties of object oriented systems as explained. Distributed objects could be developed in different programming languages through multiple language mappings. These should include mappings to object-oriented languages, which would be most natural given the object-oriented nature of the abstract language itself. In complex distributed systems there is a need for generic applications which can operate without statically built-in knowledge of the objects they access. Such applications need to use a *dynamic invocation* facility. Finally, the relevant environments should be easy to use by hiding communication details. They should also be performant and scaleable in order to encourage distribution.

ODP [ODP], which is examined in more detail in Chapter 4, identifies a number of properties of distributed systems. *Openness* addresses both software portability through standard APIs and requires interoperability through agreed communications protocols. *Distribution transparencies* mask the details of the mechanisms used to overcome distribution problems. These include among other *access transparency*, which masks differences in data representations and remote execution, and *location transparency*, which masks the location of a distributed component providing a service.

We have thus identified the following key properties of object-oriented distribution frameworks:

- an abstract, object-oriented specification language that supports inheritance and polymorphism
- mappings of the abstract language to object-oriented and also procedural/modular programming languages
- user friendly APIs that hide communication and protocol details
- dynamic access facilities that obviate the need for static (i.e. pre-compiled) knowledge of object specifications in client applications
- good performance and scalability so that distribution is encouraged and exploited
- openness in terms of both standard APIs and communication protocols

- distribution transparencies, and in particular access and location

The rest of this chapter explains the issues behind a C++-based software architecture that realises the OSI-SM / TMN model in a distributed object-oriented framework fashion. We will examine the proposed framework against the above properties at the end of this chapter, in section 3.9.1. We will also examine ANSA, the OSF DCE and OMG CORBA against the properties set above in Chapter 4.

3.3 Issues in Realising the Protocol Part of the Q_3 Interface

In this section, we consider issues associated to the mapping of the OSI-SM/TMN Common Management Information Service and protocol (CMIS/P) [X710][X711] onto object-oriented environments through suitable APIs. A brief introduction to CMIS/P has already been given in section 2.1.4 of Chapter 2. We will start this section by examining CMIS/P and the supporting OSI protocol stack in more detail. We will then discuss relevant research work and will present our approach, discussing also alternative design possibilities.

3.3.1 The Q_3 Protocol Stack

As discussed in section 2.2.1 of Chapter 2, TMN traffic may use the telecommunications network being managed. In addition, parts of the TMN operate in other networks attached to the telecommunications network. This implies that the TMN Q_3 protocols need to operate over a number of diverse lower layer data network technologies, spanning from X.25 and the Signalling System No. 7 (SS7) to the Internet TCP/IP, which is rapidly becoming the dominant data network technology. The lower layer stack profile for the Q_3 interface is specified in [Q811]. This comprises a number of sub-profiles as depicted in Figure 3-1.[1]

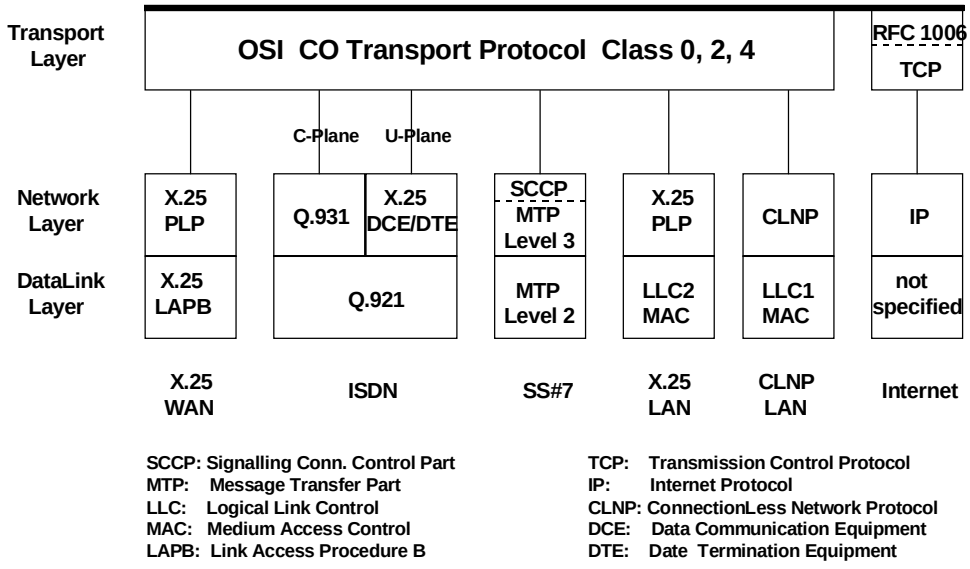


Figure 3-1 Lower Layer Protocol Profile for the Q3 Interface (from [Q811])

[1] Familiarity is assumed with the OSI 7 layer reference model [X200] and data network technologies in general. A good introduction can be found in [Tanen96].

The X.25 wide area profile, the ISDN and the SS7 profiles are the natural candidates when managing X.25, ISDN and SS7 networks respectively. It should be noted that the performance of the ISDN and SS7 signalling protocols tends to be very high in a wide area, which may not be the case with traditional data network technologies such as X.25 and the Internet TCP/IP. When managing plain transmission networks such as SDH and SONET, any of the previous technologies may be used over the "embedded communications channel". In the case of the B-ISDN which will be based on ATM technology, additional mappings will be defined over both the relevant signalling [Q2931] and user planes. Finally, for parts of the TMN operating in local area networks, it is possible to run either X.25 or the OSI ConnectionLess Network Protocol (CLNP).

In all the above combinations of network and data link protocols, the OSI Connection-Oriented Transport Protocol (COTP) provides the end-to-end Connection-Oriented Transport Service (COTS). The use of COTP class 0, 2 or 4 depends on the reliability characteristics of the underlying network service. Interoperability between subnetworks of different network technologies can be achieved either through network layer relaying, which involves protocol conversion, or through transport service bridging. We have discussed briefly the issues of protocol conversion and service bridging in section 2.3.2.2 of Chapter 2, while explaining aspects of mediation functions.

All the above technologies are pure OSI ones. Over the last years though, the Internet TCP/IP has undoubtedly become the dominant data network technology. As such, the ITU-T recognised the need to support a TCP/IP-based profile for the Q_3 interface. This can be done by treating TCP as a reliable network protocol, in a similar fashion to X.25, and operating over it a convergence protocol that provides the OSI COTS. The key difference between the COTS and the service offered by TCP is that the former is packet-oriented while the latter is stream-oriented. As such, the convergence protocol consists of two parts: a small "packetisation" protocol over TCP, which makes it appear as an OSI network protocol; and the OSI TP class 0 over the packetisation protocol that offers the COTS. This approach was standardised through the RFC 1006 [Rose87]. The intention is to enable OSI upper layer protocols and applications to operate over the Internet lower layer protocols. A good discussion of the relevant issues can be found in [Rose90].

Using this approach, the Q_3 upper layer protocols may operate over TCP/IP in a completely transparent fashion. Note though that this approach is *different* and not interoperable to the CMOT [Besa89] approach which will be discussed in section 3.3.2.4. Interoperability between Q_3 stacks based on TCP/IP and Q_3 stacks based on any of the other OSI lower layer technologies can take place through transport service bridging [Rose90]. The transport service bridge should

run on the network node that interconnects the subnetworks of the two different technologies, e.g. on the node that connects a TCP/IP local area network to the SS7 telecommunication network.

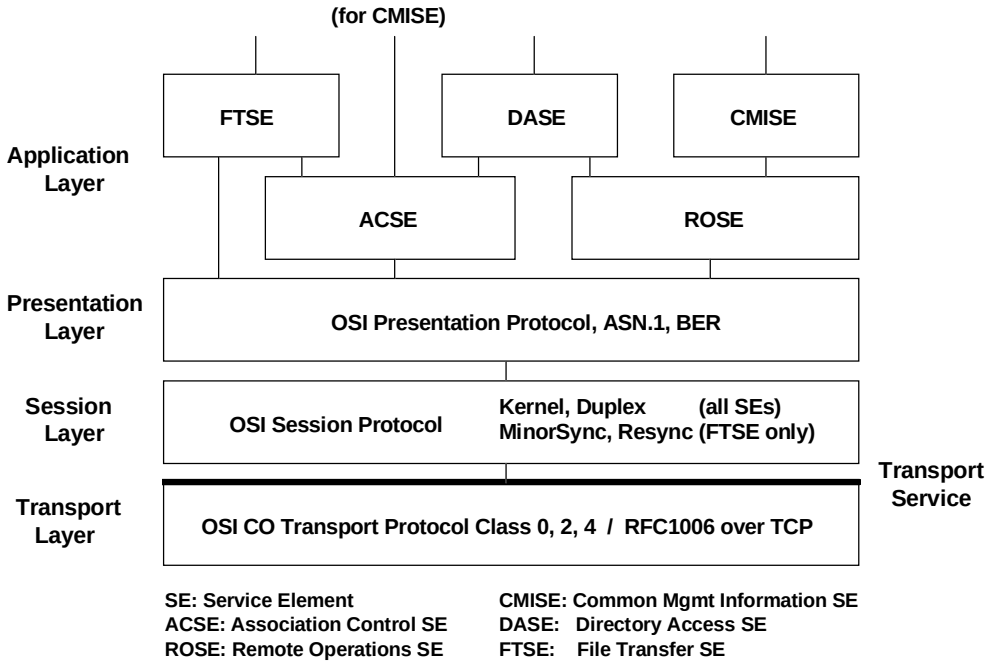


Figure 3-2 Upper Layer Protocol Profile for the Q3 Interface

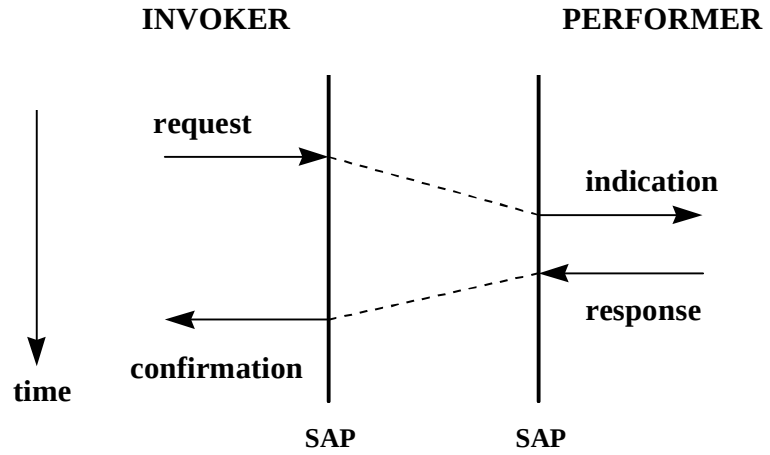
While the lower layer Q3 profile may vary, the upper layer profile is always the same as shown in Figure 3-2. The main Application Service Elements (ASEs) that are part of the Q_3 interface are CMISE [X710] and DASE [X511], while FTSE [FTAM] may be also used in the future. The service provided at the Transport Service Access Point (TSAP) is a reliable, packet-based service that does not support graceful connection release. The OSI Session Protocol adds graceful connection release, half-duplex exchanges through token management, dialogue control through checkpointing and synchronisation, activity management and exception reporting. Both CMISE and DASE need none of the sophisticated functionality of the session layer and use only the basic kernel and duplex services. The File Transfer SE needs also the Minor Synchronisation and Re-synchronisation services.

While the Session Service Access Point (SSAP) supports data exchanges with no structure, the Presentation Protocol adds structure to the data through the Abstract Syntax Notation One (ASN.1) [X208] language. ASN.1 implements an *abstract syntax* whose data structures need to be converted to byte streams and transmitted across the network and vice-versa. This functionality is provided by various sets of Encoding Rules (ER) that implement different *transfer syntaxes*. The mapping of an abstract syntax to a transfer syntax is termed a *presentation context*. The various ASEs may use different presentation contexts which are

negotiated at connection establishment time. The presentation layer keeps track of the presentation contexts and provides syntax matching functions that serialise and de-serialise the relevant data structures. The Q_3 interface specification [Q3] suggests the use of the Basic Encoding Rules (BER) [X209] as the transfer syntax.

The application layer structure consists of a number of layered ASEs. ACSE [X217] manages application layer connections which are termed *associations*. It provides a combined interface to the PSAP and SSAP connection management services but adds also Application Entity Title (AET) parameters for the calling and called parties. A AET in its complete form is the directory name of the application as explained in section 2.3.1 of Chapter 2, e.g. {c=GB, o=UCL, ou=CS, cn=ATM-NM-OS}. A simpler form for an AET is the value of the application process relative name, e.g. ATM-NM-OS. OSI applications use the ACSE services either directly, e.g. for establishing CMISE associations, or indirectly through other ASEs, e.g. for establishing DASE and FTSE associations (see Figure 3-2).

ROSE [X719] realises the OSI mechanism for building distributed applications based on a *request/response* paradigm. Though asynchronous in nature, it can also support synchronous Remote Procedure Call (RPC) semantics [Birr84], which is what many distributed applications are built on. Both CMISE [X710] and DASE [X511] use ROSE to implement management information and directory access operations respectively. We are going to discuss ROSE and CMISE in more detail while addressing their realisation, since they are the main components of the upper layer Q_3 profile. The picture of the latter is completed by FTSE for file transfer [FTAM], which uses directly the presentation layer services.



SAP: Service Access Point

Figure 3-3 OSI Invoker and Performer Interactions

In general, interactions between adjacent OSI layers or application layer ASEs take place at Service Access Points (SAPs), which offer the services of the subordinate layer or ASE. Exchanges between peer entities across the network follow a request-indication cycle, followed by a response-confirmation cycle for confirmed exchanges. The requesting user of a service is termed an *invoker* while the accepting user is termed a *performer*. This interaction model is depicted in Figure 3-3.

3.3.2 Issues in Realising the Upper Layer Part of Q_3

3.3.2.1 General Issues in Realising Upper Layer Infrastructures

Realising an upper layer Q_3 stack profile requires lower layer protocol infrastructure. Lower layers based on TCP/IP, X.25 and TP4/CLNP exist for most multi-purpose operating systems such as UNIX and WindowsNT. It should be noted that TCP/IP support comes typically bundled at no additional cost, while one has to pay extra for OSI lower layer protocols. Upper layer infrastructure, including Q_3 support for CMISE at least, can be bought today from many vendors of OSI and TMN systems. Back in the mid eighties there were no products available while the provision of efficient and reusable upper layer OSI stack infrastructure was a research issue. A major research effort in realising OSI upper layer protocols and applications and validating the relevant specifications has been the ISO Development Environment (ISODE) [ISODE][Rose90]. This provided support for the upper layer stack including ACSE, ROSE, DASE and FTSE and was used as the basis for the OSIMIS platform.

When designing software abstractions for ASEs based on a particular upper layer stack, one has the freedom to be different from the supporting infrastructure since the latter can be hidden using encapsulation. For example, ISODE is based on the structured or modular paradigm with APIs in the C programming language [Kern78] while OSIMIS is based on the object-oriented paradigm, with APIs in C++ [Strau86][Ellis91]. The ISODE APIs are not at all visible when using OSIMIS since they are encapsulated in the OSIMIS infrastructure. An important aspect related to the supporting environment though is that it might not be possible to hide all its aspects completely. This concerns in particular ASN.1 manipulation, as explained next.

Presentation layer support comes typically through ASN.1 compilers which produce concrete programming language representations for the relevant types. They also produce relevant logic for converting those representations to and from a generic representation that is understood by the presentation layer; the latter converts those to and from byte streams according to the relevant transfer syntax. An ASN.1 compiler with C mappings produces C data structures and separate

encode, decode and print functions while a compiler with C++ mappings produces C++ classes with relevant encode, decode and print behaviour. A good discussion on ASN.1 compilers and relevant issues can be found in [Neuf90].

The mapping of ASN.1 to a programming language realises a ASN.1 API. This can be modified to reflect the taste of the designer of an application layer infrastructure. For example, the ISODE ASN.1 API is procedural while the author designed and implemented an additional "wrap-up" compiler that produces encapsulating C++ classes, which will be described in section 3.4. The functionality of the latter though depends on the conventions behind the encapsulated C structures produced by the native ISODE ASN.1 compiler. This observation can be generalised as follows: the designer of application layer infrastructure is somewhat "restricted" by the native ASN.1 API. This restriction can be completely removed only if a new ASN.1 compiler and associated API is designed and implemented.

3.3.2.2 Related Work on CMISE APIs

Before we move on to discuss issues behind the realisation of CMISE and relevant APIs, let's look at related work in this area and position the work presented here. The author's design and implementation of an ISODE-based CMISE that constituted the initial component of OSIMIS dates back to 1989. At that time, there was no related research work or a similar commercial product. In fact, the OSIMIS CMISE implementation served for some time as the only available reference implementation and was subsequently used as the basis for a number of products. The OSIMIS CMISE design decisions and relevant abstractions are described in [Pav93a], a tutorial on "Implementing OSI Management". The relevant API is documented in [Pav93b], the OSIMIS-3.0 manual. Brief descriptions are also given in [Pav95a] and [Pav96b] which describe the OSIMIS platform as a whole.

The only other work in the literature that discusses CMISE realisation issues is [Dens91], [XMP] and [Chat97]. The first one [Dens91] discusses the realisation of DEC's CMIS services while the latter two present work of standards bodies. The first of those is X/Open's Management Protocols API specification (XMP) [XMP], released in 1992. The second is a recent attempt by the NMF to provide object-oriented TMN APIs [Chat97], including a CMIS API known as CMIS/C++.

[Dens91] describes DEC's approach for a CMIS API in their Enterprise Management Architecture (EMA) [Strut94]. The particularly interesting aspect of their approach is that their API is a generic protocol-independent one, which can be mapped onto particular protocols

through different Access Modules (AMs) [Struct89]. The interface consists of a single procedure which takes as parameters the *verb* or directive, the *in_entity* or object to access, the *attributes* for get and set directives, the *in_q* for additional qualifiers (e.g. access control) and the *in_p* for additional input arguments (e.g. action argument). The *out_p* contains the results/errors while the *out_entity* parameter contains information on the object(s) on which the directive was performed e.g. the class and name of the object. Scope and filter information are part of the in_entity parameter but only single level scoping is possible while not all the aspects of CMIS filtering are possible. This interface was obviously designed before CMIS and cannot cope fully with the richness of the latter. On the other hand, it is an interesting attempt on a generic, polymorphic, dynamic invocation interface that can be mapped onto different protocols.

The X/Open XMP interface was the first attempt from a standards body to standardise a CMIS API. The intention behind such an API is to separate OSI-SM/TMN applications from the underlying CMIS/P protocol stack so that portability across different vendors' stacks is possible. This API was first introduced in 1992 and has similarities to the OSIMIS one which had been publicly available since 1990. This is a procedural API in the C language. Every CMIS request and response primitive maps to a corresponding procedure that can be called asynchronously e.g. Get-req() and Get-rsp(). A Receive() procedure needs to be called to receive the result. A synchronous call model with RPC semantics is also supported e.g. Get(). Management "sessions" need to be established before sending and receiving messages through the Bind() call while they may be terminated through the Unbind() call. Finally, automatic name to address resolution is provided that maps application names to addresses.

While all this design makes sense and is in fact extremely similar to that of the OSIMIS CMIS, it has two serious drawbacks. First, the API tries to cater both for CMIS and SNMP and this creates unnecessary complexity. CMIS and SNMP follow very different philosophies as explained in [Pav94d][Pav97a] and also in [Geri94] and elsewhere. As such, there is no tangible benefit from unifying their access APIs while additional complexity is introduced for dealing with the different object models, parameters to common primitives etc. A second and more important drawback concerns the use of the associated X/Open ASN.1 API [XOM]. This takes an object-oriented view of structural information but does not incorporate the characteristics of object-oriented systems as explained in section 3.2.1. In particular, the functions for manipulating objects are separate from the definitions of those objects and there is no notion of encapsulating or hiding the information associated with objects. We could characterise both XMP and XOM as object-based instead of object-oriented. In summary, the combined XOM/XMP API is complex and daunting to use.

The object-based nature and complexity of XOM/XMP has led the NMF to define recently the NMF/C++ API, which comprises ASN.1, CMIS and GDMO APIs [Chat97]. This work is very much related to the work described in this chapter and has been produced by a group of experts over a 2-3 year period. The author has initially participated in that group and the OSIMIS APIs have been one of the relevant inputs. The CMIS API is known as CMIS/C++. This offers a set of C++ classes for modelling the CMISE and ACSE primitives and their parameters in an asynchronous fashion only. It also offers a set of objects for referring to outstanding operations (invocation handles) and two different mechanisms in order to receive operation indications and confirmations: a callback facility through a callback class and a queue facility through a queue class. A "convenience" API is also available for automatic association management but applications can avoid using this and can take explicit control of association establishment and release.

3.3.2.3 Issues in Realising CMISE Over ROSE

The OSIMIS CMISE implementation is based on the ISODE environment and uses the relevant ASN.1 compiler known as *pepsy* and the associated ASN.1 API. An early implementation based on the still evolving CMIS/P ISO documents was produced by S. Walton of UCL under the auspices of the ESPRIT INCA project, in 1988. A management system for monitoring the activity of the OSI transport protocol was developed based on it, as described in [Knig89]. The CMIS/P standards [X710][X711] achieved a state of maturity in 1989 and the author re-designed and re-implemented completely CMISE in late 1989. This became the fundamental building block for the OSIMIS platform and has remained fairly stable ever since, used subsequently in a number of commercial products.

Since CMISE is based on ROSE, a ROSE implementation is necessary while ACSE is also necessary for association management. ISODE provided both ACSE and ROSE implementations. In cases where the available OSI stack provides presentation layer services only, implementing ACSE and ROSE is fairly straightforward. ACSE is essentially a wrap-up of the PSAP connection management features. ROSE implements a simple, generic request/response protocol for distributed OSI applications. It also provides a facility of operations *linked* to another operation, which can be thought as remote callbacks. This facility is used by CMISE for operations resulting in multiple replies through scoping.

ro-invoke	invokeId,	linkedId,	operation,	argument
ro-result	invokeId,		operation,	result
ro-error	invokeId,		error,	parameter

Table 3-1 ROSE Primitives and Associated Parameters

Table 3-1 shows the ROSE primitives (apart from *ro-reject*) and associated parameters. Because of its asynchronous nature, a unique identifier needs to be associated with every outstanding request (invokeId). Callback invocations are linked to the initial operation through a linked identifier (linkedId) which should have the value of the original invoke identifier. The operation code, argument, result, error code and error parameter are defined by higher level protocols e.g. CMISE.

Implementing a ROSE protocol machine is not difficult. The simplest policy for an associated API is a procedural asynchronous one, with a procedure modelling each of the primitives and their parameters, e.g. RoInvoke(), RoResult(), RoError(), and a separate procedure for receiving indications and confirmations, e.g. RoWait(). This is exactly the API policy ISODE implements. A relevant design decision is if the user of ROSE will be given responsibility for the uniqueness of the invokeId parameter or if the latter will be assigned by ROSE, passing it back to the caller as a "voucher" in order to be matched against the reply and linked invocations. ISODE has decided to leave this responsibility to the caller.

The state information required by a ROSE protocol machine is very little i.e. the outstanding request and indication invokeId's for a session so that further invocations, results and errors can be checked for consistency. ROSE implementations support typically *at most once* reliability characteristics, with an operation requested exactly once and the performer keeping no state of previous invokeId's. *Exactly once* reliability characteristics are also possible, with the invoker requesting repeatedly the operation with the same invokeId until a result/error or a "duplicate operation" rejection is received. In this case, the performer needs to keep additional state of the invokeId's of operations in a session from an epoch date. ROSE supports *total* distributed operations: for any given operation, the result and all exceptions (errors and rejections) are well-defined and distinguishable. The concept of totality is important for reliable distributed systems.

m-create	invokeId, access, objClass, objName, referenceName, attrList
m-cancelGet	invokeId, getInvokeId
m-get	invokeId, access, objClass, objName, scope, filter, sync, attrIdList
m-set	invokeId, access, objClass, objName, scope, filter, sync, setReqList
m-action	invokeId, access, objClass, objName, scope, filter, sync, actionInfo
m-delete	invokeId, access, objClass, objName, scope, filter, sync
m-eventRep	invokeId, objClass, objName, eventTime, eventType, eventInfo

m-createRes	invokeId, < objClass, objName, time, attrList | error, errorInfo >
m-cancelGetRes	invokeId, < | error, errorInfo >
m-getRes	invokeId, linkId, < objClass, objName, time, getAttrList | error, errorInfo >
m-setRes	invokeId, linkId, < objClass, objName, time, setAttrList | error, errorInfo >
m-actionRes	invokeId, linkId, < objClass, objName, time, actionReply | error, errorInfo >
m-deleteRes	invokeId, linkId, < objClass, objName, time, | error, errorInfo >
m-eventRepRes	invokeId, < objClass, objName, time, eventReply | error, errorInfo >

Table 3-2 CMIS Primitives and Associated Parameters

Table 3-2 shows the CMIS request and response primitives and associated parameters. *The m-get*, *m-set*, *m-action* and *m-delete* primitives may operate on many managed objects through the *scope, filter* and *sync* parameters. The base object for the search is identified by the *objName* parameter. When these primitives are applied to a single object instance (i.e. without scope and sync), the optional *objClass* parameter may be used to request allomorphic behaviour. In the case of the *m-create* primitive, the objClass parameter is mandatory while objName is optional for classes with "automatic instance naming" properties. The setReqList parameter of m-set is a list of `{attrId, attrVal, modifyOperator}` tuples. The modify operator can take the values *replace*, *setToDefault*, *add* and *remove*, the latter two for multi-valued attributes [X720]. The access parameter is reserved for access control [X741] but its use has not yet been defined. The rest of the parameters are self-explanatory.

The response primitives model both result and error conditions. The relevant result parameter is passed back together with the objName, objClass and a timestamp. In case of an error, the error

code is passed back together with relevant error information. CMIS/P defines a comprehensive set of errors [X711]. It also allows for object-specific errors through the processingFailure error.

Implementing a CMISE protocol machine over ROSE is not difficult, despite the fact that CMISE [X711] is a much more complex protocol than ROSE [X219]. While a ROSE protocol machine can be implemented without the need for an ASN.1 compiler due to the reduced primitive set and the simple parameter types (ASN.1 INTEGER and ANY), CMISE needs ASN.1 compiler support because of its complexity. In the case of requests and responses, the main task of a CMISE protocol machine is to assemble the API parameters, create and encode a CMISE Protocol Data Unit (PDU) and use the relevant ROSE primitive. In the case of indications and confirmations, the CMISE PDU should be decoded and the API parameters should be populated. The only state information that needs to be kept concerns outstanding m-get requests so that m-cancelGet requests are validated at source.

3.3.2.3.1 Association Management

A CMISE API should provide access to the relevant services in an efficient, flexible and easy-to-use manner. CMISE services can only be used after an association has been established through ACSE. An important design decision to make is whether the CMISE user will be given control of establishing and releasing ACSE associations or such activities will be handled transparently by the infrastructure. This decision has an impact on the API and there are three possible design decisions:

a) association management becomes an explicit part of the CMIS API; this is the approach followed in OSIMIS [Pav93b] and the NMF CMIS/C++ [Chat97];

b) Bind and Unbind facilities to management applications are part of the API but association management takes place transparently while in the bound state; this is the approach followed by XMP [XMP]; and

c) all the CMIS operations accept some form of global names, with the prefix part denoting the management application.

The third one is the most abstract. In the case of a procedural API with a procedure for each primitive, an additional API parameter is required for those CMIS primitives that do not include a name in the remote system, i.e. m-cancelGet, m-create and m-eventRep. This parameter should be the distinguished name of the target application. An important drawback of this approach is that it hides completely the relevant negotiation capabilities at association establishment, which can only take place in a pre-packaged fashion behind the API. This is fine only as far as the

applications' requirements are in accordance with the pre-packaged policy. But as [Chat97] points out, "a convenience interface is only convenient if it does what you need"!

In the case of b), association options to a particular destination can be specified through the Bind() primitive. Some form of identifier is passed back from Bind() which should be used as a prefix to all the primitives. Note that this is *not* an association handle but denotes the binding with that system. Associations will be opened and closed transparently by the infrastructure thereafter. Finally, a) is the most "low level" approach but also most powerful, since it allows explicit control of associations. Connect and disconnect primitives are available, with Connect() typically returning an association handle to be used as a prefix in the other primitives.

In any of the schemes presented above, the destination can be specified through a logical application name e.g. NM-OS or the full name {c=UK, o=UCL, ou=CS, cn=NM-OS}. The latter is typically required only when crossing domain boundaries since the local domain name is known by the infrastructure. If location transparency is supported through the OSI Directory [X750] or any similar mechanism, the infrastructure will map this name to an address. If location transparency is not supported, the location name needs to be passed together with the application name e.g. NM-OS@athena or {c=UK, o=UCL, ou=CS, cn=NM-OS, cn=athena}. In this case, some form of local database is used to map this name to an OSI presentation address. The problem with this approach is twofold, as already discussed in section 2.3.1 of Chapter 2: first, there is no location transparency; and second, it is very difficult to keep those local databases consistent in a large-scale distributed system.

From the three schemes presented above, the author chose to implement a) because it offers the maximum expressive power. In addition, it models explicitly the ACSE [X217] specification and its use dictated by CMISE [X711]. The latter is an important reason since the relevant standard documents or typical textbook explanation of the OSI application layer structure could serve as reference documents on the structure and semantics of the API. The same reasons hold also for the NMF CMIS/C++ [Chat97], though the latter also offers an additional "convenience" API for automatic association management. In OSIMIS, such an API is only offered at a higher level as described in section 3.5.

The author also chose to unify the ACSE and CMISE APIs under one common API. This API supports location transparency through the OSI Directory as dictated by [X750], but it can also be used in a non-transparent fashion through a local database. In the former case, directory access takes place "underneath" the combined CMISE / ACSE API.

3.3.2.3.2 Procedural vs. Object-Oriented APIs

Another important decision regarding the API concerns the use of a structured, object-based approach or an object-oriented one. The decision here should be clear-cut: an object-oriented approach offers important advantages in terms of reusability, simplicity, easier state and memory management, etc. For example, one could model the CMISE protocol machine as an object, with methods corresponding to the relevant service primitives. This object would also encapsulate ACSE features as described above. An instance of that object would model a (remote) management interface, encapsulating the relevant binding and association information.

The X/Open XMP [XMP] has chosen an object-based as opposed to an object-oriented approach. The NMF CMIS/C++ [Chat97] follows a fully object-oriented approach; this is also the case with the OSIMIS high-level manager API known as Remote MIB (RMIB) [Pav94b] that will be described in section 3.5. On the other hand, the OSIMIS CMIS API, known as the Management Service Access Point (MSAP) API [Pav93b] follows a procedural approach and is implemented in C, in a similar fashion to the XMP one.

The main reasons for the decision not to follow an object-oriented approach, at least for the CMIS API, were political rather than technical. At that time (second half of 1989), the intention was to make the CMISE protocol machine part of the ISODE distribution. This would increase the popularity and acceptance of OSI-SM as a whole due to the wide deployment of ISODE in the research community. ISODE is written in C and follows a procedural approach throughout, so the same approach should be followed for CMISE. It should be noted that the main ISODE contributor, M. Rose, was at the time involved in the standardisation of SNMP [SNMP] and his views were pretty vitriolic regarding OSI-SM; an amusing tale of his can be found in [Rose91]. Because of his views, the OSIMIS CMISE implementation was never incorporated in ISODE, which meant that the original CMISE design and implementation *could* have been object-oriented. In fact, after the wide deployment of OSIMIS in the mid-90's, the fact that OSIMIS required the ISODE stack and its ASN.1 tools was considered by many as a liability.

The OSIMIS CMISE implementation follows an asynchronous procedural paradigm, with every request and response primitive in the Table 3-2 mapped to a separate procedure. The parameters of those primitives are mapped directly to those in Table 3-2, with the addition of an association descriptor parameter. Responsibility for invokeId consistency is left to the user of CMISE, in a similar fashion to the ISODE ROSE. A "m-wait" procedure models indications and confirmations, following a queue model as opposed to a callback or upcall model [Clark85]. The m-wait request primitive may be instructed to simply inspect the queue i.e. return immediately, to

wait for a specified period or to wait indefinitely until an indication or confirmation arrives. A relevant application could be organised in a single or multi-threaded fashion. In the case of a single-threaded execution paradigm, the incoming indications and confirmations need to be managed. The relevant mechanism is orthogonal to the MSAP API and is discussed in more detail in section 3.7.

A part of the MSAP CMIS API is shown in Table 3-3. The approach taken corresponds very closely to the CMIS standard [X710], which can serve as relevant documentation. The reader may observe the similarity between the programmatic CMIS interface of Table 3-3 and the abstract CMIS primitives of Table 3-2.

int M_Get (int assocId, int invokeId, External* access, MIDentifier* objClass, MNane* objName, CMISScope* scope, CMISFilter* filter, CMISSync sync, int nattrs, MIDentifier attrIdList[], MSAPIndication* mi);
int M_GetRes (int assocId, int invokeId, int linkedId, MIDentifier* objClass, MNane* objName, char* currentTime, int nattrs, CMISGetAttr attrList[], CMISErrors error, CMISErrorInfo* errorInfo, MSAPIndication* mi);
int M_Wait (int msd, int waitPeriod, MSAPIndication* mi);

Table 3-3 (Part of) The MSAP CMIS API

3.3.2.3.3 Attribute, Action, Event and Specific Error Values

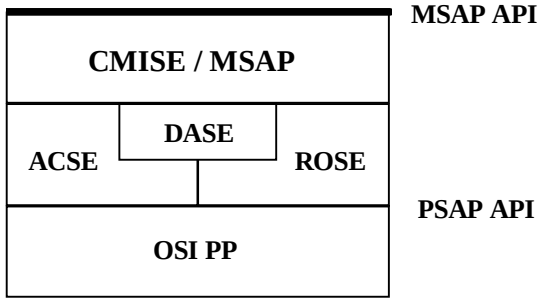
Another important decision behind a CMIS API concerns the representation of parameters with dynamic nature whose exact type is not known by the CMIP protocol. These are the attribute value, action information and reply, notification information and reply and object-level error information in the case of a processingFailure error. All these are specified as ASN.1 ANY parameters by CMIP [X711], acting essentially as place holders for information that will be defined at a higher-level, by managed object classes. Their exact definition at the CMIP level uses the ANY DEFINED BY ASN.1 construct, which associates a name to an ASN.1 type e.g. an attribute name to the corresponding type. For example, the specification of the *uxObj* class in Appendix C associates the *nUsers* attribute to the *ObservedValue* ASN.1 type defined in [X721] and the *echo* action information and reply arguments to the *GraphicString* type [X209].

When designing a CMISE API, the key question is if responsibility for encoding and decoding those values will be left to the application, as the type ANY implies, or it will be undertaken by CMISE. The second approach is obviously more user-friendly since it results in additional information hiding. It implies though that the CMISE layer should be able to determine the exact data type for an ANY value and invoke the appropriate encode / decode method. This can be accomplished through access to *meta-data* produced from the GDMO and ASN.1 specification for a particular information model. These should map attribute, action, event and specific error names to ASN.1 types and associated manipulation logic i.e. encoding and decoding procedures. An object-oriented API can unify the ASN.1 data structures and relevant manipulation logic through object classes, allowing for a natural representation of the ANY type in higher layer APIs. In fact, this is what the OSIMIS object-oriented ASN.1 API does as will be described in section 3.4.

The procedural MSAP API follows the ISODE policy, which always passes control to the API user for dealing with the ANY type. XMP [XMP] is fairly flexible, accommodating both approaches: the programmer can instruct the CMISE infrastructure to either encode/decode ANY values or leave them to be manipulated by the application. Finally, the CMIS/C++ API [Chat97] uses the separate ASN.1/C++ API which is object-oriented, supporting a natural manipulation of the ANY type.

An associated design decision has to do with the API representation of the attribute, action, notification and specific error names. These are defined at the CMIP level as ASN.1 OBJECT IDENTIFIER (OID) types, with their values defined in a GDMO specification through "REGISTER AS" clauses. For example, the *objectClass* attribute of the *top* class [X721] is registered as {joint-iso-ccitt(2) ms(9) smi(3) part2(2) attribute(7) objectClass(65)} or, more concisely, as 2.9.3.2.7.65 . It is this value that is communicated across the Q_3 interface, encoded according to the transfer syntax in use, and *not* the user-friendly string representation "objectClass". The simplest API policy is to pass the actual OID in a concrete representation e.g. a C data structure. A better API policy that results in more information hiding is to pass the user-friendly string and let the CMISE layer map it to the associated OID. In the latter case, the CMISE layer should have access to meta-data associated with a particular GDMO information model. The OSIMIS MSAP API follows the former, more "low-level" approach in order to be consistent with the ISODE API policy.

3.3.2.3.4 The OSIMIS CMISE



MSAP: Management Service Access Point
PSAP: Presentation Service Access Point

Figure 3-4 The OSIMIS CMISE Realisation

Figure 3-4 shows the layered OSIMIS CMISE realisation. When this figure is contrasted with Figure 3-2, which depicts the upper layer protocol profile for the Q_3 interface, it shows two important design decisions which have already been described above. First, the incorporation of association control primitives in the CMISE API; and second, the incorporation of location transparency features in the CMISE API through the use of DASE for accessing the OSI Directory. These decisions make the CMISE API self-contained i.e. the user does not need to either learn and or access a separate ACSE or DASE API. In OSIMIS the name MSAP describes both for the CMISE implementation, i.e. the relevant library, *and* the CMISE API.

From an engineering perspective, all the upper layer protocols, including CMISE, are realised as libraries linked with a management application. This means that each management application contains its own "instance" of the upper layer protocol stack. The lower layers are typically part of the operating system's kernel, so all the applications use a single instance of the lower layers. In a general purpose operating system such as UNIX, TCP/IP, TP4/CLNP and X.25 are part of the kernel. This means that RFC1006 in the case of TCP/IP and TP0 / TP2 in the case of X.25 run in user space, together with the upper layer stack. An evaluation of the impact of the "tightly-coupled" upper layers to the size of management applications is presented in section 3.8.

In summary, realising a CMISE API and protocol machine involves a number of important design decisions as described in this section. It is not difficult though, assuming the existence of ASN.1 tools and disregarding, at least initially, location transparency features. The author spent around 3 months for the design, implementation and testing of the bare-bone OSIMIS CMISE in late 1989. The API paradigm was procedural instead of object-oriented in order to maintain ISODE compatibility, which finally proved to be unnecessary. C. Stathopoulos of ICS

implemented the location transparency features first in early 1993 and re-implemented them in 1995 to track the [X750] standard.

The popularity of this procedural CMISE implementation proved to be remarkable, used in many research projects and products and promoting the concept of OSI-SM as a whole. It should be finally noted that the discussion and the issues raised in this section, though targeted at CMIS/P, can be generalised for any other application service element.

3.3.2.4 Alternative Mappings for CMISE

In the previous sections we discussed the issues behind realising CMISE over a full upper layer Q_3 protocol stack [Q812]. While CMIP [X711] is specified in ASN.1 and uses the OSI ROSE [X219], it is a general management protocol that can be put over a different transport infrastructure. In fact, the whole of OSI-SM including GDMO, the SMFs and the manager-agent application framework can be adapted and used over environments other than OSI. In this section we examine the issues behind alternative mappings for CMISE.

The first key requirement for CMISE is reliable transport infrastructure. This can be provided either by the OSI TP over pure OSI lower layer protocols or by the Internet TCP/IP, in the latter case with or without the RFC 1006 packetisation protocol. The second key requirement is a presentation facility of similar expressive power to ASN.1.

As it was already mentioned, CMISE and ROSE-based ASEs do not use any of the sophisticated functionality of the session protocol. This means it should be possible to provide a lightweight mapping for an upper layer stack by using a modified version of the OSI presentation protocol operating directly over a reliable transport mechanism. This was exactly the thinking behind the mapping specified in [Rose88], which is known as the Lightweight Presentation Protocol (LPP). That particular mapping exploits the fact that BER streams are “self-delimited” because of the *tag-length-value* approach [X209], so it maps LPP directly over the Internet TCP which provides a stream-oriented reliable transport service. The LPP “hardwires” a number of parameters which are generally negotiated at association establishment and restricts the transfer syntax to be the BER.

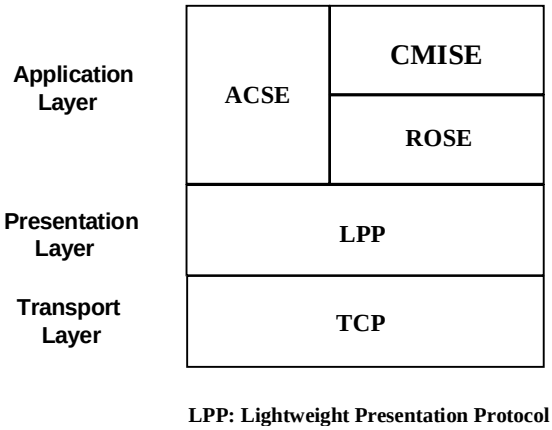


Figure 3-5 The CMOT Protocol Stack

The mapping of CMISE over the LPP is shown in Figure 3-5 and is known as CMOT - CMIP Over TCP/IP [Besa89]. Since ISODE supports the LPP, OSIMIS subsequently supports the CMOT stack. This mapping is a Q_x protocol in TMN terms but it has not had much use in telecommunications environments. The key reason is that it sacrifices interoperability in comparison to the full Q_3 stack while it does not bring significant improvements to the size and performance of relevant applications, as it will be discussed in section 3.8. The LPP approach as a whole has remained mostly a paper exercise, without any real deployment.

Another approach towards lightweight mappings of OSI application layer protocols has been taken by the Lightweight Directory Access Protocol (LDAP) [LDAP]. LDAP has been recently very popular because of the commercial interest in OSI directory technology [X500] over the Internet. Its key aspects are:

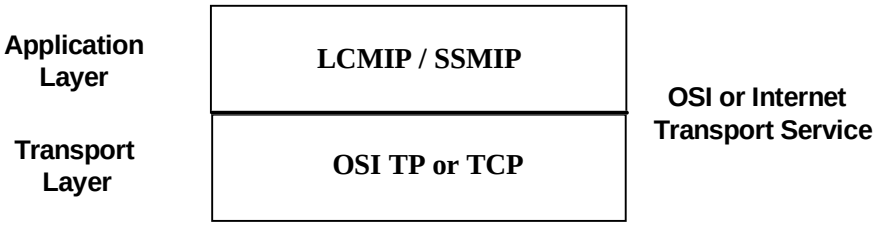
- protocol data units are carried directly over the TCP or the OSI transport service, bypassing completely both the presentation and session overhead;
- many parameters of the protocol primitives are encoded as strings e.g. distinguished names, attribute types and values, etc.; and
- the protocol data units themselves are specified in ASN.1 and encoded in BER which is used in a restricted form in order to simplify implementations.

LDAP-based applications do not need presentation facilities since they communicate attribute values in pretty-printed *string* form. LDAP is typically used for lightweight DUAs and it is relevant PDUs are converted to DAP through LDAP-DAP gateways or service relays. Many recent commercial DSAs support LDAP directly, in addition to DAP, in which case there is no need for gateways.

The author was intrigued by the principles behind LDAP and adopted similar concepts for the specification of the Lightweight CMIP (LCMIP) protocol [Pav95c]. The latter uses the same principles described above for LDAP but introduces a number of additional simplifications. While ASN.1 and BER are used to describe and encode the LCMIP PDUs, only a limited set of ASN.1 types are allowed, namely *NULL*, *INTEGER*, *OCTET STRING*, *SEQUENCE*, *SET OF* and *SEQUENCE OF*. The LCMIP PDU specification was structured in such a way as to allow maximum reusability of encoding and decoding procedures. There are no ASN.1 optional elements while numeric tags have been kept to a minimum, since they result in different encodings. The idea was to be able to implement LCMIP by hand, without the need for ASN.1 compilers which inevitably introduce inefficiencies.

Distinguished names are communicated as strings using the ISODE string convention, used also in LDAP e.g. "`logId=1@logRecordId=5`". CMIS filters use the string format described in Appendix D. Attribute, action, notification and specific error values are communicated as pretty-printed strings whose structure should be agreed. OSIMIS provides already "standard" string representations for the DMI types [X721]. New GDMO/ASN.1 specifications should always define the string representations for the ASN.1 types they introduce. For example, the MeanStdDev type defined in Appendix C could have the string representation
"`{ mean: <val> stdDev: <val> }`".

LCMIP includes a number of other optimisations that simplify the structure of the CMIP PDUs. The LCMIP structure of the GetArgument and GetResult LCMIP types is presented in Appendix E, highlighting some of the major design decisions and simplifications. The LCMIP approach was never implemented, mainly because of lack of resources but also because the use of full Q_3-capable applications proved to be less expensive than widely believed, as explained in section 3.8. It would be interesting though to be able to quantify the savings of this approach compared to the full Q_3 one, the author intends to pursue this in the future. The - very simple - LCMIP protocol stack is shown in Figure 3-6.



LCMIP: Lightweight Common Management Information Protocol
SSMIP: Simple String-based Management Information Protocol

Figure 3-6 The Lightweight and String-based CMIP Protocol Stack

While LCMIP is much simpler than CMIP and uses strings for attribute, action, notification and specific error values, its PDUs are still specified in ASN.1 while BER is used for their encoding. Taking the concept of a string-based representation further, a possibility would be to communicate the whole of the lightweight CMIP PDUs in a pretty-printed string form. The key issue in this case is the definition of this pretty-printed PDU format.

When the author initially implemented the OSIMIS CMISE and generic agent infrastructure, he also implemented a number of generic command line manager programs that provided the full functionality of the CMIS primitives, namely *mibdump* (or *mget*), *mset*, *maction*, *mcreate*, *mdelete* and *evsink*. Their syntax, which follows the UNIX convention for command line arguments, is described in [Pav93b] and realises essentially a string form for the CMIS request primitives. Since these programs parse their input, the relevant logic can be reused as part of a CMIP protocol machine. A full string-based CMIP requires also the specification of the string form for the reply and error PDUs, which can be based on the same principles.

Table 3-4 presents the PDU structure of a string-based lightweight CMIP protocol which is largely based on the syntax of the OSIMIS generic command-line manager programs [Pav93b]. The author named this Simple String-based Management Information Protocol (SSMIP). Figure 3-6 shows the protocol mapping which exactly the same as that of LCMIP. The SSMIP should be more lightweight than LCMIP while its big advantage is that it can be implemented without the need for ASN.1 compiler support. Its PDUs can be encapsulated in protocols such as HTTP so that it can drive WWW displays. The SSMIP was never implemented in OSIMIS but a variation of it was implemented in a commercial product which is based on OSIMIS.

mcre invId [-A access] -c class [-n name | -s superiorName] [-r referenceName] [-a attrName=value ...]
mcgt invId
mget invId [-A access] [-c class] [-n name] [[-s scope] sync] [-f filter] [-a attrName ...]
mset invId [-A access] [-c class] [-n name] [[-s scope] sync] [-f filter] [-[w|d|a|r] attrName[=value] ...]
mact invId [-A access] [-c class] [-n name] [[-s scope] sync] [-f filter] -a attrName[=value]
mdel invId [-A access] [-c class] [-n name] [[-s scope] sync] [-f filter]
mevr invId [-c class] [-n name] [-t time] -a eventName[=value]

mres <op>[2] invId [-l linkId] [-c class] [-n name] [-t time] [-a [-e error] [name[=value] [-m modify]] …]
merr invId error [<errInfo>[3]]

Table 3-4 String-based CMIP PDUs

In summary, comparing the LCMIP and SSMIP approaches, it is worth going all the way and adopting the SSMIP approach as opposed to the LCMIP one since it uses a simpler, fully string-based approach. Such protocols are useful for driving TMN WS applications [Pav96d] which typically manipulate management information in string form. They may not be particularly good for applications that examine management information and perform numerical calculations. In those cases, a significant amount of processing time will be spent in converting management information from numeric to string form and vice-versa. It should be finally mentioned that when these protocols drive TMN WS applications, they may be thought as "proprietary" F protocols. One of the reasons we have proposed not to standardise the TMN F interface in Chapter 2 is because there exist many different styles of string-based CMIP protocols over various different transport mappings. WS applications that use SSMIP should communicate with the rest of the TMN through service relays that convert SSMIP to the Q_3 protocol stack.

While in this section we have considered alternative mappings for CMISE that use simple string-based representations of attribute, action, event and specific error values, the applicability of those protocols is typically restricted to workstation applications as explained above. In Chapter 4 we examine a more general mapping to OMG CORBA and distributed object technologies.

---

[2] The operation type is the name of the operation i.e. mcre, mcgt, mget, mset, mact, mdel, mevr.

[3] The error information has structure that is specific to the particular error code.

3.3.2.5 Summary

In this section we described in detail the issues behind realising the protocol part of the TMN Q_3 interface and examined possible policies for the relevant API. Since CMIS/P requires only a reliable transport service and a presentation facility, it can be mapped onto alternative transport infrastructures as discussed in section 3.3.2.4.

In summary, CMIP is a modestly complex protocol that can be relatively easily realised. The necessary infrastructure should be an OSI development environment that includes a procedural ASN.1 compiler. An aspect that makes the protocol more complex than necessary is the use of object identifiers instead of user-friendly string names. This is something pertinent to all the OSI applications and introduces unnecessary complexity. Proponents of the solution claim that it provides guaranteed uniqueness of names, which is true. On the other hand, such uniqueness could be policed by a central authority that would endorse new GDMO specifications.

While CMIS services are relatively easy to provide, the real difficulty lies in providing a development environment that hides CMIS/P and provides an object-oriented distributed platform that supports the rapid development of TMN applications by developers with little or no knowledge of network programming. The relevant issues are examined in sections 3.5 and 3.6, after we examine issues on object-oriented ASN.1 manipulation in the next section.

3.4 Issues in Object-Oriented ASN.1 Manipulation

A important aspect of OSI upper layer is the manipulation of ASN.1 data structures. Typically, ASN.1 compilers map those abstract data structures to concrete programming language representations, as already discussed in section 3.3.2.1. This mapping can be procedural, with separate data structures and syntax manipulation procedures, or object-oriented, with classes mapped to ASN.1 types and relevant syntax manipulation methods. An important issue in the latter case is the polymorphic design of the relevant API. OSIMIS is based on ISODE which supports a procedural ASN.1 manipulation style through the *pepsy* ASN.1 compiler, in a similar fashion to most ASN.1 infrastructures of the late eighties and early nineties [Neuf90]. As such, it has been necessary to define object-oriented ASN.1 abstractions in OSIMIS and to provide an object-oriented ASN.1 compiler with C++ mappings. The issues behind high-level object-oriented ASN.1 manipulation are discussed in this section.

There is very little work in the literature on issues related to flexible high-level ASN.1 APIs, as opposed, say, to work on ASN.1 performance measurements and comparisons. One well-known approach to ASN.1 manipulation is X/Open's XOM API [XOM] that has been described in the previous section. Its key drawback is that it is object-based as opposed to object-oriented. One of the main reasons behind the fact that the XOM/XMP API has been rather unpopular has to do mostly with the XOM rather than the XMP part.

Another more recent and much more promising approach is the NMF ASN.1/C++ API, which is part of the overall TMN/C++ series of APIs [Chat97]. This maps ASN.1 types to C++ classes that derive ultimately from the abstract class `AbstractData`; the latter heads the C++ class hierarchy for ASN.1 types. This class provides functionality inherent in ASN.1 data types, such as encode, decode, print, compare, discover its type information etc. For each ASN.1 built-in type, a C++ subclass of AbstractData provides a type-specific representation, e.g. Boolean, Integer, Sequence, etc. Other ASN.1 types map to classes derived from those. The whole approach is in fact extremely similar to the one designed by the author and described in this section. It should be noted that the OSIMIS approach was passed as input to the NMF TMN/C++ team.

Before we describe our approach, we need to clarify further some issues behind the representation and use of the ASN.1 ANY type in upper layer infrastructures. The ANY type is typically used to pass "unknown" types between layered ASEs. For example, the CMISE m-get PDU is of

ASN.1 type GetArgument [X711] but is passed to ROSE [X219] as an ANY type. This is because ROSE is unaware of higher-level ASEs, so it specifies the arguments and results of its operations as of type ANY. The extensive use of the ANY type in upper layer infrastructures implies that a relevant API representation is necessary. One may suggest that this should be a byte stream, encoded according to the transfer syntax in use. The problem with this approach is that it increases the complexity for the relevant applications since they have to deal explicitly with encoding and decoding. It also violates the layering principle since the application layer becomes explicitly aware of the transfer syntax, which is normally a function of the presentation layer. In addition, the logic produced by the ASN.1 compiler becomes dependent on the particular transfer syntax. This reduces flexibility, in the sense that a different transfer syntax can be supported only after recompiling the application software.

A generic procedural approach for representing the ASN.1 ANY type was first pioneered in ISODE [ISODE][Rose90]. According to this, a special data structure can represent any ASN.1 type in a transfer syntax independent fashion. This can be generally termed an "intermediate ASN.1 representation" and in ISODE it is specifically called a Presentation Element (PE). ASN.1 compilers produce logic that converts a concrete representation to an intermediate one and vice-versa (*encode* and *decode* a type). Such generic structures can be passed through the upper layer APIs and can be *serialised* (and *de-serialised*) in the presentation layer, according to the relevant presentation context. This is exactly the policy followed in the ISODE ASEs and in many other non object-oriented OSI infrastructures.

The key ingredient of high-level TMN APIs is the object-oriented manipulation of ASN.1. The author realised this early in the initial design of OSIMIS (around 1990). This led to the design of the generic *Attr* class, whose polymorphic interface defines the rules for generic object-oriented ASN.1 manipulation in OSIMIS [Pav93a][Pav93b]. Every ASN.1 type is modelled by a class that derives either directly from Attr, or indirectly through another generic class such as Enumerated, Integer, String, List, etc. In addition, the *AnyType* class models specific types in a generic fashion and is typically used by generic manager applications. Finally, the generic *AVA* class (Attribute Value Assertion) was added later to model the *ANY DEFINED BY* ASN.1 construct, which associates attribute, action, event, and specific error names to ASN.1 types. An O-O ASN.1 compiler wraps up the output of the ISODE pepsy compiler and produces C++ classes for specific ASN.1 types.

```
class Attr
{
protected:
    virtual PE      _encode ();
    virtual void*   _decode (PE);
    virtual char*   _print ();
    virtual void*   _parse (char*);
    virtual void    _free ();
    virtual void*   _copy ();
    virtual int     _compare (void*, void*);
    virtual void**  _getElem (void*);
    virtual void**  _getNext (void*);
    // . . .
    Attr ();        // abstract class

public:
    virtual char*   getSyntax ();
    Bool            isMultiValued ();

    PE              encode ();
    char*           print ();
    void            ffree ();
    void*           copy ();
    // . . .
    void*           getval ();
    void            setval (void*);
    int             setstr (char*)

    virtual Bool    filter (int, void*);

    void            clear ();
    virtual         ~Attr ();
    // . . .
};
```

Code 3-1 The Generic Attr Class that Models an ASN.1 Type

We will start discussing the aspects of object-oriented ASN.1 manipulation by examining the features of the Attr class, which realises the fundamental aspects of the ASN.1 API. Attr is an abstract class which is never instantiated but serves as the root of the relevant C++ class hierarchy, in a similar fashion to the AbstractData class in the ASN.1/C++ API [Chat97]. The O-O ASN.1 compiler produces automatically derived classes that the model specific ASN.1 types e.g. Integer, OperationalState, etc.. The Attr class encapsulates the relevant data type while derived classes redefine the associated manipulation functions. It comprises the following polymorphic manipulation methods:

- _encode and _decode, which convert to and from the intermediate representation;
- _print and _parse, which convert to and from a pretty-printed string;
- _free, which releases memory and _copy, which makes a copy;
- _compare, which compares two instances of the encapsulated data type; and
- _getNext and _getElem which can be used to walk through a multi-valued type (ASN.1 SET OF or SEQUENCE OF) and access the contained elements.

All these methods can be produced automatically through an ASN.1 compiler. The produced print and parse functions may use a "not-so-pretty" string representation while the _compare method may not be able to exploit "buried in" semantics of a particular type , e.g. order comparisons for a "time" type. The user might want to overwrite those methods through manually supplied ones or even add new methods. Such a facility needs to be supported by the relevant O-O ASN.1 compiler. It should be noted that CMIS filtering is automatically supported through the filter method. Finally, it is possible to use string representations to construct and manipulate a type e.g. Integer("5"), AdminidtrativeState("locked"). It is also possible to use intermediate representations, e.g. ISODE PEs, which is important for constructing relevant objects when ascending the protocol stack, i.e. in indications and confirmations.

A number of additional generic classes model generic properties of a "family" of types, such as enumerated, null-terminated string, list, etc. An example inheritance hierarchy is depicted in Figure 3-7 in OMT notation. The *count*, *gauge*, *threshold* and *tide-mark* classes model the relevant types defined in [X721]. It should be noted that those types have well-defined behaviour which has to be hand-written. They can be implemented once though and be subsequently re-used.

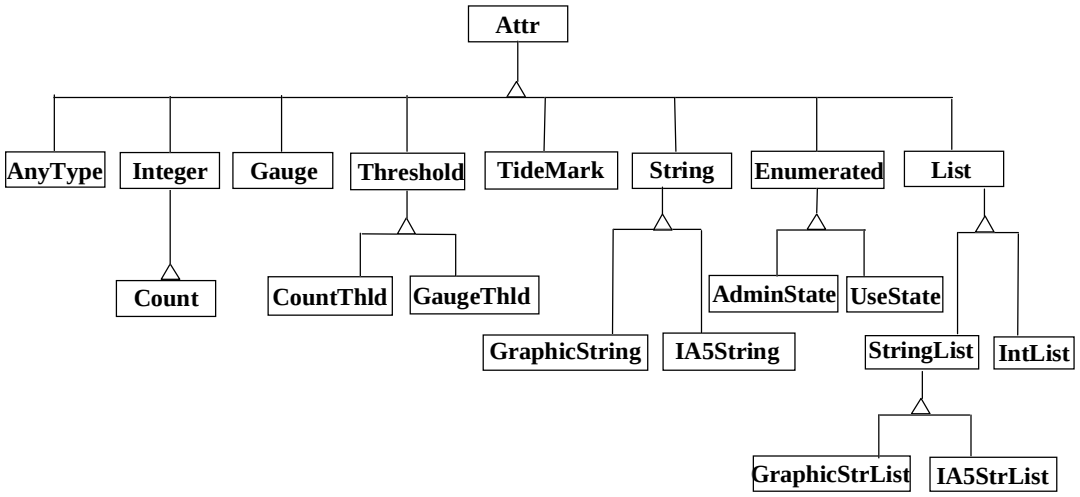


Figure 3-7 Example ASN.1 Class Hierarchy

In an object-oriented CMISE protocol stack, values of the "unknown" ANY type i.e. attribute, action, event and object-specific error values, can be passed through the API as Attr parameters. These can descend the stack in the case of requests and responses, carrying with them behaviour to produce the required intermediate representation at some point. In the case of indications and confirmations, the CMISE layer needs to know which ASN.1 type corresponds to a particular name so that it can construct the corresponding object e.g. *Count* for the *bytesSent* attribute. This

can only be achieved if the CMISE layer has access to *meta-data* for a particular GDMO/ASN.1 object model. The exact information required is the mapping of the relevant name to the corresponding ASN.1 type. This information can be produced by GDMO compilers and stored in some form of database that implements an information model *repository*. We will discuss issues on such a repository in section 3.5.6.

In management environments, there is a particular class of manager applications which are information model independent in the sense that they do not depend on the semantics of the particular model they access. An example of such an application is a MIB browser [Pav92a] which allows a human user to browse through the MIT across a management interface and set attribute values, perform actions and create/delete managed objects. Such an application is written once, in a generic fashion, and needs simply to be "updated" (i.e. linked with in software terms) with the particular ASN.1 syntaxes for a new GDMO information model [Pav92a]. Such an application cannot use the specific C++ classes that model particular syntaxes, e.g. Integer, GraphicStringList etc., since it is supposed to deal with any future types introduced by new GDMO models. This necessitates the introduction of a generic type, the *AnyType*, which is a generic specialisation of the Attr class (see Figure 3-7). This type implements its functionality by having access to meta-data, which include in this case the encoding, decoding, printing and parsing procedures produced by the ASN.1 compiler or supplied by the human user. This type is of paramount importance for generic applications and was conceived early in OSIMIS, together with the Attr class. It should be noted that the NMF NMF/C++ API [Chat97] includes such a facility.

Finally, the mapping of arbitrary ASN.1 types to names is modelled in ASN.1 via the ANY DEFINED BY construct. This is a powerful feature but can also be easily misused. It can be thought as the ASN.1 equivalent to the "void pointer" in C and C++. For example, the programmer may map the wrong type to a particular name which will be transmitted correctly across the network but will most probably result in an obscure error produced by the other end (or in a core dump in the case of not-so-bullet-proof software!). In order to harness the relevant power, the author designed the Attribute Value Assertion (AVA) class, whose salient features are depicted in the Code 3-2 caption.

```
class AVA
{
    // . . .
public:
    static Bool     createError ();

    char*           getName ();
    Attr*           getValue ();
    CMISErrors      getError ();
    CMISModifyOp    getModifyOp ();

    char*           print ();
    void            clear ();
    // . . .

    AVA (char*, void*, CMISModifyOp = noModifyOp);
    AVA (char*, char*, CMISModifyOp = noModifyOp);
    AVA (char*, Attr*, CMISModifyOp = noModifyOp);
    AVA (OID, PE, CMISErrors, CMISModifyOp);
    // . . .
    ~AVA ();
};
```

Code 3-2 The AVA Class

The AVA class encapsulates the relevant syntax object together with the corresponding name. It also encapsulates a modify operator, used in CMIS m-set requests, and an error value, used in CMIS results to denote some error condition e.g. attribute not set because of access, invalid value or other problem. When constructing a AVA instance, the consistency of the name and type are checked through access to the information model repository, so the programmer is protected in the case of an error. Both the Attr and AVA classes are used extensively in the OSIMIS high-level APIs.

The Code 3-3 caption shows some of the power and expressiveness of the object-oriented ASN.1 API through brief examples. The latter show the manipulation of the base ASN.1 type INTEGER through C++ classes. Initially, an *Integer* instance is constructed, first through the encapsulated data structure, i.e. the C++ *int* built-in type, and then through an equivalent string value. The encapsulated data structure can be accessed either through the generic Attr::getval() method or through the type-specific user-supplied Integer::getint() method. The next example shows the construction of a generic AnyType instance that realises an integer with the same value as before. The type can be specified either explicitly, e.g. “Integer”, or implicitly through the associated name, e.g. “bytesSent”. Note that the value may be printed without any knowledge of the encapsulated data structure. This form of manipulation is typical in generic manager applications. Finally an AVA instance is constructed and printed, first by using explicit type knowledge and then generically. Programming with explicit types has the advantage of additional type-specific methods which may be manually supplied, e.g. Integer::getint(). It is also more performant since there is no need to access meta-data at construction time, as it is the case with the AnyType class.

```
// Integer construction with the encapsulated data structure

int* val = new int;
*val = 10;
Integer* i = new Integer(val);

// Integer construction with string

Integer* j = new Integer("10");
cout << "j is " << j->getint() << endl; // getint() is user-added

// generic AnyType construction based on type knowledge (Integer)

AnyType* k = new AnyType("Integer", "10");
char* cp;
cout << "k is " << cp = k->print(); // generic print
delete cp;

// generic AnyType construction based on name knowledge (bytesSent)

AnyType* l = new AnyType("bytesSent", "10");
cout << "l is " << *(int*) l->getval() << endl;

// AVA construction based on name (bytesSent) and value (Integer)

AVA*     m = new AVA("bytesSent", i);
cout << "m name is " << m->getName();
cout << " and value is " << *(int*) m->getValue->getval() << endl;

cout << "m is " << cp = m->print() << endl; // prints name: value
delete cp;
```

Code 3-3 Example Use of the O-O ASN.1 API

We will finalise this section with a brief discussion of the realisation issues. Ideally, one would like to implement an ASN.1 compiler that produces directly the relevant C++ classes for the various types. Since users might also want to overwrite some of the produced methods, such a facility needs also to be supported. Unfortunately, most OSI upper-layer environments come with their own, typically procedural, ASN.1 compilers which follow their own conventions. One may try to encapsulate their output, which is what the author has done with the ISODE pepsy compiler, but there are limits as to how far one can go with this approach. For example, the C structures produced by a procedural compiler are still visible in the C++ API presented above and have to be manipulated by programmers. Removing all the dependencies on a procedural ASN.1 API means essentially implementing a new native object-oriented ASN.1 compiler and this can be *a lot* of work (ASN.1 is a pretty complex language). The "wrap-up" approach though is acceptable and can go a long way towards supporting object-oriented APIs for ASN.1 manipulation.

The OSIMIS O-O ASN.1 compiler wraps-up the output of the ISODE pepsy compiler. It performs first some rudimentary parsing of the ASN.1 input file, invokes the pepsy compiler and parses partially the output of by the latter. It knows the ASN.1-to-C data structure conventions used by pepsy and uses this and the parsing information to build a symbol table and to drive the generation of C++ classes that correspond to the ASN.1 types. User's code that overwrites

produced methods can also be incorporated. The process of producing C++ classes for ASN.1 types is shown in Figure 3-8. This wrap-up compiler is a modestly complex program, written using the GNU version of the UNIX awk tool [Kern84]. The latter provides a very flexible "interpreted C"-like facility. The author spent about a month for the relevant implementation. On the other hand, it took almost a year to realise the possibility for such a solution and work out the relevant details. The O-O ASN.1 compiler was implemented in the course of 1994.

The heart of the ASN.1 API is the Attr class whose polymorphic interface took quite some time to finalise. A first version was designed and implemented in 1990 with the early version of OSIMIS but it took a number of iterations to get it right. It is difficult to quantify such a task in terms of complexity and time, since it involves very delicate aspects of polymorphic object-oriented design. The same is true for all the high-level object-oriented OSIMIS APIs. Finally, the AVA class was conceived and added in 1993, while designing the RMIB high-level manager API; the latter will be described in section 3.5.

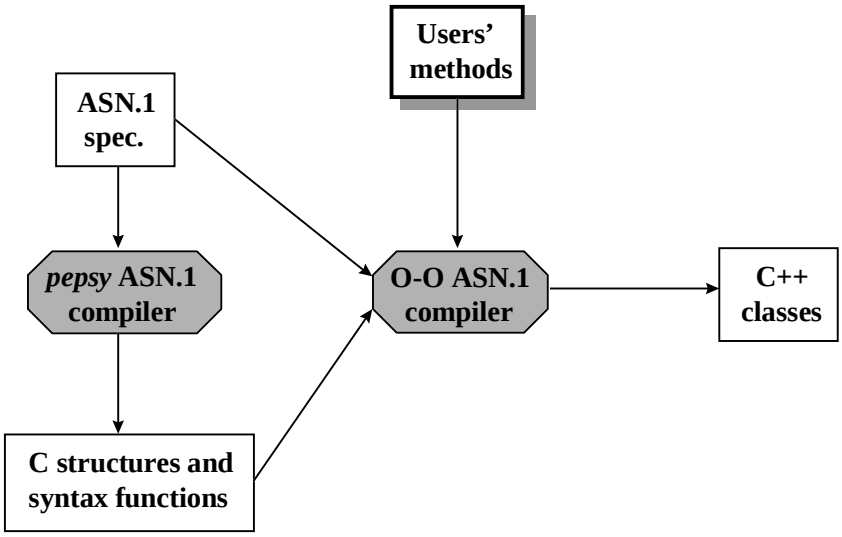


Figure 3-8 C++ Class Generation for ASN.1 Types

In summary, object-oriented ASN.1 manipulation is key for user-friendly upper layer APIs. The author recognised early the need for such a facility and designed and implemented first the relevant classes and later the O-O ASN.1 compiler. For a number of years, the only available compilers and APIs were procedural while there is very little research work discussing the issues behind object-oriented ASN.1 APIs. The need for such an API was recognised in the mid-90's by the NMF TMN/C++ team, with the author's approach being an input to that work. Their ASN.1/C++ solution [Chat97] bears a lot of similarities to the approach presented here, which was conceived, designed, realised and made publicly available much earlier.

3.5 Issues in Realising Object-Oriented Manager Infrastructures

3.5.1 Introduction

In section 3.3.2.3 we concentrated on issues related to the realisation of CMISE and discussed policies for the relevant API. The author chose to design a relatively "low-level", procedural CMISE API in accordance to the ISODE conventions, the latter being the supporting OSI upper layer development environment; this API is known as MSAP. Such an API can be cumbersome and difficult to use while it typically results in increased code size for management applications. As already explained, the author adopted such an approach for non-technical reasons and had already in mind higher-level, object-oriented APIs while designing the MSAP one. In fact, developers of OSIMIS-based TMN applications that use those object-oriented APIs are not aware of the MSAP API at all.

Because of the manager-agent duality, there exist two types of higher-level APIs: APIs in manager roles, which should provide access to managed objects in a user-friendly, high-level fashion, hiding CMIS/P access details but not sacrificing its expressive power. And APIs in agent roles, providing an environment for the realisation of managed objects which hides the underlying CMIS/P access aspects, allowing implementers to concentrate in the realisation of the associated behaviour. It should be noted that this separation is not specific to the manager-agent model and the OSI-SM/TMN information architecture but it is an inherent aspect of any distributed object-oriented environment that follows the client-server model. For example, the OMG CORBA [CORBA] mapping of the relevant abstract language to concrete O-O programming languages has also two distinct client and server facets.

In this section, we concentrate in high-level object-oriented infrastructures and associated APIs for applications in *manager* roles. There can be two types of such APIs: those that provide an object-oriented abstraction of an application in agent role; and those that provide object-oriented abstractions of its individual managed objects. In both cases, by object-oriented abstractions we mean objects in local address space which model either an agent application or an individual managed object, in a "proxy" fashion. One may remark that CMISE APIs model to some extent an agent application. This is true, but none of the existing CMISE APIs, i.e. the OSIMIS MSAP [Pav93b], the X/Open XMP [XMP] and the NMF CMIS/C++ [Chat97], provide the level of abstraction and functionality we will propose here.

3.5.2 Related Work

There is less research work on higher-level manager APIs, as opposed to research work in designing and developing agent infrastructures and APIs, which will be considered in section 3.6. In particular, there is very little research work on abstractions modelling remote agents, similar in scope and functionality to the proposal of the author detailed in section 3.5.3. It seems that relevant research work either concentrates in modelling raw CMIS functionality, e.g. the X/Open XMP [XMP], the NMF CMIS/C++ [Chat97], or addresses directly abstractions at the managed object level, e.g. the IBM Object-Oriented Interface (OOI) [Holb95], the NMF GDMO/C++ [Chat97] and even CORBA [CORBA][BenN94].

The IBM Object-Oriented Interface (OOI) [Holb95] presents an approach for modelling remote managed objects in a manager application through Proxy Managed Objects (PMOs) associated to a Proxy Agent object that models the remote agent. The latter provides an object-oriented view of CMIS, realised over XOM/XMP. It encapsulates the XMP "session" and provides access to XMP operations through its methods, the key advantage being the use of an O-O ASN.1 API which is also part of the OOI framework. The proxy agent provides methods for accessing multiple objects through scoping, filtering and may create a number of PMOs as a result of those operations. It also allows manager objects to request event reports and supports a relevant "event-queue" facility. In summary, the Proxy Agent provides facilities similar to some extent to those of the Remote MIB (RMIB) agent proposed by the author. It should be noted that the author's work [Pav94b] predated the relevant work by IBM and has been taken into account in the latter.

The PMOs model remote managed objects and the emphasis is on strong typing that results in a friendlier interface for novice application developers, providing compile-time checking. Specific PMO classes, produced through a GDMO compiler, include methods for accessing attributes and invoking actions. Both the attribute access methods (get, set) and the action methods take as parameters the precise C++ classes that model ASN.1 types according to the GDMO/ASN.1 specification. In addition, a weakly-typed generic PMO class is also provided which can be used in generic applications such as MIB browsers. This uses a generic ASN.1 type, similar to the AnyType presented in section 3.4. When weak-typing is used, run-time checking is supported through access to meta-data. IBM's approach was the main input to the NMF TMN/C++ API team for the object oriented manager API. As such, it has influenced strongly the latter which is described below.

The NMF GDMO/C++ API [Chat97] is a complete approach towards such an API. It is to an extent a superset of the IBM OOI one, incorporating also aspects from other proposals, including

the OSIMIS high-level manager APIs. A remote agent is modelled through an Agent Handle (AH) class, through which multiple managed objects can be addressed through scoping and filtering. A MO is modelled through a Managed Object Handle (MOH) class, which is specific to the particular GDMO class and is produced automatically by the GDMO compiler. A Managed Object Handle Factory (MOHF) contains a set of "machine objects", one for each of the GDMO classes the manager knows about. New MOHs can be created through the factory for a particular Agent Handle or as a result of linked replies from scoped operations. The whole framework is strongly-typed, providing compile-time error checking,. There is also a weakly-typed version which can be used in generic manager applications. Finally, the user may introduce specific MOH-derived classes which implement particular policies with respect to the relevant remote MO e.g. attribute value caching, periodic updating, etc.

Although CORBA will be introduced formally in Chapter 4 and despite the fact that its underlying protocol is not CMIP/Q_3, it is worth examining briefly its client or "manager" API. In CORBA [CORBA][BenN94], remote objects are accessed through object references which point to a local proxy object. The latter is similar to the NMF GDMO/C++ MOH, providing strongly-typed access to either general or to attribute access methods (get, set). Here, there is no notion of agents since these do not exist in the CORBA framework. A difference between CORBA proxy objects and MOHs is that every access to a proxy object results in an operation to the relevant master object, while this is not the case with a MOH. In addition, in CORBA it is not possible to implement access policies "inside" a proxy object through inheritance. Finally, weakly-typed access is possible through the Dynamic Invocation Interface (DII).

3.5.3 The Remote MIB Manager Infrastructure

The need for a higher-level manager API was identified early while designing and developing OSIMIS. Having developed first the Generic Managed System (GMS) agent support infrastructure which will be described in section 3.6, it became clear that a similar environment was required for manager applications. The initial approach for developing those was to use the native MSAP CMISE API. This necessitated low-level CMIS primitive manipulation and resulted in a lot of complex code for management applications.

The first relatively sophisticated TMN system that was developed using OSIMIS was the RACE NEMESYS service management system during 1991. This comprised a combined Q-Adapter / Mediation device for an ATM simulator, an Element Manager OS and a Service Manager OS, organised in a hierarchical fashion [Pav91b][Pav92b]. The two OSs were going to be designed

and developed by people who had little or no experience of network programming. It was necessary to provide them with high-level abstractions, so that the author investigated the relevant issues and designed the framework for two infrastructures:

- the Remote MIB (RMIB) agent level infrastructure; and
- the Shadow MIB (SMIB) managed object level infrastructure.

J. Cowan of UCL implemented the first embryonic version of the RMIB infrastructure in 1991. This version served as a proof of concept and was successfully used in the NEMESYS project, allowing relatively inexperienced people to develop (the manager parts of) TMN OSs. It was though skeletal and incomplete and, as such, it was not released with OSIMIS-3.0 [Pav93b]. A more complete approach to the design and implementation of the RMIB infrastructure took place during 1993 in the RACE ICM project. The author together with T. Tin of UCL revised the model and redesigned the relevant API. T. Tin subsequently implemented the RMIB infrastructure which was publicly released with OSIMIS-4.0 [Pav95b].

3.5.3.1 Design Issues and Objectives

One of the key objectives while designing such an infrastructure was to hide the intricacies of the underlying communication infrastructure as much as possible, without sacrificing any of the available expressive power. This implies a genuine object-oriented abstraction of the OSI management access service which would be programmer-friendly and easy to use, making possible the development of TMN applications by users with very little or no knowledge of network programming. We examine in detail below the sub-objectives accruing from this main objective.

As already explained, most CMISE APIs leave the implementer to deal with the low-level mechanics of management information access. Managed object class, attribute, action, notification and specific error names are typically passed across the API as Object Identifiers (OIDs), leaving to the user the responsibility to convert from and to user-friendly strings. Managed object names (i.e. distinguished names) are passed across as list-like data structures or objects, while the user typically deals with names as strings. The CMIS filter parameter is passed again as a complex data structure while the user would like a string-based, symbolic manipulation paradigm. In the case of non object-oriented CMISE APIs such as the OSIMIS MSAP, attribute, action, event and specific error values are passed across encoded in an intermediate representation. It is the responsibility of the API user to encode and decode native data structures to this representation.

In addition to the relatively low-level parameters to primitives, the CMIS service is by nature asynchronous and this is the way it is typically modelled in relevant APIs e.g. the OSIMIS MSAP and the NMF CMIS/C++. In this case, the application is responsible for assembling linked replies and identifying the empty PDU that denotes the termination of a particular series. While an asynchronous remote execution model is necessary in single-threaded execution environments, it requires state information to be kept by the application. A synchronous execution paradigm with RPC-like semantics is more natural to programmers, so a synchronous mode of CMIS operations should be supported. In this case though, a multi-threaded execution paradigm is necessary for increased performance. A discussion on the issues of synchronous vs. asynchronous execution paradigms for distributed management applications will be presented in section 3.7.

A key aspect of OSI-SM / TMN is their event-driven management approach. Particular events may be requested at a fine level of granularity by setting relevant filters in Event Forwarding Discriminator (EFD) objects as explained in Chapter 2. The manager application needs to explicitly manipulate EFDs while some "dispatching" mechanism is necessary to deliver an arriving event report to the right manager objects inside that application. This functionality could be undertaken by the supporting infrastructure. Finally, association management could be supported transparently by the infrastructure but the facility to explicitly initiate, terminate and abort associations should be also available to applications.

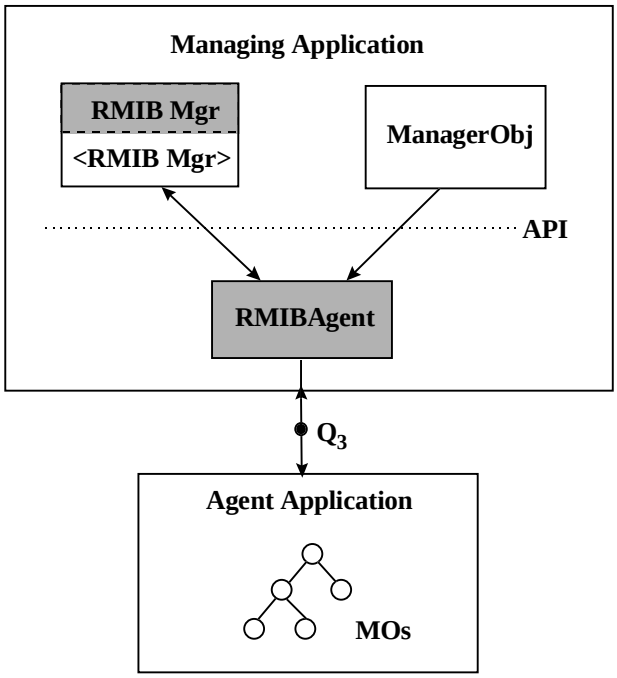
In summary, a high-level manager API has the following requirements as presented above:

- attribute, action, event and specific error names should be manipulated through user-friendly strings instead of object identifiers;
- attribute, action, event and specific error values should be manipulated through objects that correspond to the relevant ASN.1 type, rendering unnecessary any explicit encoding and decoding manipulation;
- object names and CMIS filters should be manipulated in a symbolic, string-based fashion;
- both asynchronous and synchronous remote operation paradigms should be supported;
- linked replies should be assembled in the synchronous mode; in the asynchronous mode, both an assembly of the whole series and a "n-by-n" callback facility should be supported;

- a high-level facility for the requesting and delivery of event reports should be supported, hiding the explicit manipulation of EFDs;
- association management should be transparent but also explicit control of associations should also be supported; and
- location transparency should be supported, in a similar fashion to CMISE APIs.

3.5.3.2 The RMIB Model

RMIB [Pav94b] is an object-oriented model that encapsulates the agent and associated MIB of a remote management interface. This is done through a class called *RMIBAgent* since it acts as an agent within the manager application for the remote system. It should be noted that the term "remote" implies logical as opposed to physical remoteness. For example, the co-operating manager and agent applications can be co-located at the same network node, interworking either through a full Q_3 interface or through a more lightweight interprocess communication i.e. a q_3 reference point in TMN terms.



<RMIBMgr> - specific class derived from RMIBMgr

Figure 3-9 Remote MIB Model and Interactions

Typically there exists one RMIBAgent instance within a manager application that corresponds to a particular remote agent, modelling a "binding" to that system. It is of course possible to instantiate more than one RMIBAgent for the same remote agent. The RMIBAgent class provides

methods that correspond to the manager role CMIS primitives i.e. m-get, m-set, m-action, m-create, m-delete and m-cancelGet, methods to request, terminate, suspend, resume the forwarding of event reports and methods for association management. The forwarding of event reports is inherently asynchronous while remote operations on managed objects are supported in both a synchronous and an asynchronous fashion. An abstract *RMIBManager* class provides the interface for the asynchronous "callbacks" or "upcalls" [Clark85], which should be specialised in derived classes. The relationship between the RMIBManager and RMIBAgent instances is many-to-many.

The RMIB model and relevant interactions is shown in Figure 3-9, in the form of co-operating object instances. In case a manager object does not need callbacks because it is not interested in event reports and uses only the synchronous operation facility, then it does not need to be of type RMIBManager. The shaded parts of the diagram indicate generic re-usable classes. Note that the RMIBAgent class encapsulates the CMISE functionality, which in the case of OSIMIS is provided by the MSAP procedural realisation. An OMT relationship of the relevant classes is also shown in Figure 3-11 in the next section, depicting both the RMIB and SMIB models.

Aspects of the RMIB model have been adopted in the IBM OOI [Holb95] and the NMF GDMO/C++ [Chat97] approaches. A difference is that these support both callback and event-queue facilities for asynchronous operations, while the RMIB supports only a callback model[4]. Another more important difference is that those models combine a RMIB and SMIB approach in *one* framework. For example, as a result of scoped operations on the equivalent remote agent object, a set of new "shadow" objects are created which can be then accessed then locally in order to retrieve the results. The OSIMIS RMIB approach has been deliberately kept separate from the more sophisticated SMIB approach, the latter being a clear superset.

The requirements presented above are all met with the above model and associated design. Distinguished names follow the ISODE string notation e.g. "`logId=1@logRecordId=5`". CMIS filters follow a special string notation [Pav93b] which the author devised together with S. Bhatti of UCL. The latter implemented the relevant conversion logic to and from the native complex data structures. An example filter is:

"`( (objectClass=log) & (!(logId=1))& (eventType=stateChange) )`".

A complete description of the string "language" for CMIS filters can be found in Appendix D.

[4] In OMG CORBA [CORBA], the callback and event-queue models are called the "push" and "pull" model respectively.

Attribute, action, event and specific error values are passed through the Attr and AVA classes of the O-O ASN.1 API. Both synchronous and asynchronous operations are supported. In the case of scoped asynchronous operations, the relevant RMIBManager may request the assembly of linked replies which will be passed to it in a single callback. It may also request separate callbacks for every linked reply or even for every group consisting of *n* linked replies as they arrive. In the asynchronous case, the specific RMIBManager is able to exercise the m-cancelGet facility.

Associations can be either explicitly manipulated or left to the infrastructure i.e. the RMIBAgent. In the latter case, the application may manipulate a time-out for "caching" an association. If no traffic to the remote real agent is encountered during that period, the association is terminated and re-established when the next operation to that agent is requested. Location transparency is supported through the OSI Directory [X750] in a similar fashion to CMISE as described in the section 3.3.2.3.

The RMIB model supports programming with local rather than global distinguished names. The RMIBAgent is identified through the remote agent's name e.g. the full directory name `c=UK@o=UCL@ou=CS@cn=NM-OS` or simply `NM-OS` when in the local domain. Subsequent operations on managed objects through the RMIBAgent should use local names e.g. `logId=1@logRecord=5`. This is in accordance with the TMN model in which managed object clusters are manipulated together, as explained in Chapter 2. A TMN OS is typically configured with the names of other OSs it needs to access instead of global names of individual managed objects. The RMIB model supports naturally this mode of operation.

The RMIBAgent supports also a high-level event reporting API. Event reports may be requested through a string filter, they can be subsequently suspended, resumed and finally terminated. The RMIBAgent needs to create and manipulate one or more EFDs with the event filtering requirements of the RMIBManager instances. with a. When an event report is received, the RMIBAgent needs to identify the relevant RMIBManager and "push" the event report to it through an upcall. The necessary "demultiplexing" has revealed an important limitation in CMIS: it is not possible to distinguish which EFD an event report originates from since the CMIS m-eventReport primitive does *not* contain the name of the relevant EFD that triggered it. This makes impossible to demultiplex the event reports arriving at a manager application.

The solution adopted in the RMIB design overcomes this limitation in the following fashion. A separate EFD is created for each RMIBManager, with a filter which is a union, i.e. an OR filter, of all the assertions a particular RMIBManager has requested. Each such EFD is created on a

different association, with an *empty* destination attribute. The OSIMIS agent realisation of an EFD conceives such a request to imply "*use the association in which you were created to report future events*". If the manager terminates this association without deleting the EFD, the EFD's operational state becomes "disabled". The RMIBAgent is thus able to demultiplex event reports and pass them to the relevant RMIBManager based on the association it receives them. This of course requires that the underlying CMISE API supports explicit association control. Though this approach works, there are two problems with it:

a) the proposed EFD behaviour in the agent is not standard, i.e. not prescribed in [X734], but was simply devised by G. Knight of UCL together with the author; and

b) the manager needs to keep the relevant association open continuously, which results in consuming network resources in connection-oriented networks and increases the memory size of the relevant applications.

It should be noted that despite the fact this approach is not standard, it was pioneered in OSIMIS and was subsequently adopted by many other commercial products. This means that interoperability between those products is possible. On the other hand, an implementation adhering strictly to [X734] should reject the creation of an EFD with no destination attribute value supplied. The problem can be properly solved if the name of the EFD that triggered the event report is added in the m-eventReport primitive as shown in the Code 3-4 caption. This of course requires a revision of the CMIS/P recommendations [X710][X711].

```
EventReportArgument ::= SEQUENCE {
    managedObjectClass      ObjectClass,
    managedObjectInstance   ObjectInstance,
    eventTime               [5] IMPLICIT GeneralizedTime OPTIONAL,
    eventType               EventTypeId,
    eventInfo               ANY DEFINED BY eventType OPTIONAL,
    efdObjectInstance       ObjectInstance -- added parameter
}
```

Code 3-4 Proposed Modification of the CMIS/P EventReport PDU

3.5.3.3 The RMIB API

A part of the API specification for the RMIBAgent and RMIBManager classes is shown in the Code 3-5 caption. A number of customised versions of the same method with different parameters are offered to suit different requirements. For example, the most powerful m-get method allows one to request many attributes from different objects through scoping and filtering in a synchronous or asynchronous fashion. Once though the name of an object is known, simpler methods may be used e.g. a synchronous method to retrieve a single attribute. Requests for

asynchronous operations manifest themselves through an argument that passes the "identity" of the relevant RMIBManager in order to be called back.

```
class RMIBAgent : public KS {
    friend class SMIBAgent
    // . . .

public:
    int bind (char* applName, char* host = NULL);
    int unbind ();

    int connect ();
    int disconnect ();

    // most powerful m-get with scope/filter (sync or async)

    int Get (char*objName, char* scope, char* flt, CMISSync sync,
             char* attrNames[], CMISObjectList*& resultList,
             RMIBManager* rmibMgr = NULL, int nByN = 0);

    // synchronous m-get for one object, one attribute only

    int Get (char* objName, char* attrName,
             Attr*& attrVal, AVA*& errInfo = NULLAVAREF);

    // synchronous m-get for one object only, many attributes

    int Get (char* objName, char* attrNames[],
             AVAArray*& attrs, AVA*& errInfo = NULLAVAREF);

    // synchronous m-action for one object only

    int Action (char* objName, char* actionType,
             Attr* actionInfo, Attr*& actionReply,
             AVA* errInfo = NULLAVAREF);

    // request and cancel event reports through string filter

    int receiveEvent (char* flt, RMIBManager* rmibMgr);
    int stopReceiveEvent (char* flt, RMIBManager* rmibMgr);

    // . . .
};

class RMIBManager {
public:
    virtual int EventReport (char* objClass, char* objName,
                 char* eventType, char* eventTime,
                 Attr* eventInfo, Attr*& eventReply,
                 RMIBAgent* rmibAgent);

    // . . .
};
```

Code 3-5 The RMIB O-O API

The Code 3-6 caption shows two usage examples which demonstrate the nature of the RMIB API, which is similar to that of recent object-oriented frameworks such as OMG CORBA. The GDMO/ASN.1 specification of the *uxObj* and *simpleStats* managed object classes used in the examples can be found in Appendix C. We will "walk through" the example and explain what is happening in order to demonstrate how the various features are used.

```
// bind to the application with name "SMA-athena"

RMIBAgent rmibAgent;
if (rmibAgent.bind("SMA-athena") != OK)
    error("cannot bind to SMA-athena!");

// find the name of uxObj instance through scoping/filtering

rmibAgent.Get(NULL, "1stLevel", "(objectClass=uxObj)",
                          NULL, NULL, resultList);
if (resultList->getError())
    error("problem with the uxObj instance at SMA-athena");
char* uxObjName = resultList->first()->getName();
delete resultList;

// request the value of the nUsers attribute

GaugeInt* nUsers; AVA* errInfo;
rmibAgent.Get(uxObjName, "nUsers", nUsers, errInfo);
delete uxObjName;

if (! errInfo) {
    cout << "nUsers is ", nUsers->getint() << endl;
    delete nUsers;
}

// bind to the application "STATSRV" (statistics server)

if (rmibAgent.bind("STATSRV") != OK)
    error("cannot bind to STATSRV!");

// perform a sq. root calculation on the object "statsId=null"

Real arg(4), *res;
rmibAgent.Action("statsId=null","calcSqrt", &arg, res, errInfo);
if (! errInfo) {
    cout << res->getreal() << endl;
    delete res;
}
```

Code 3-6 Example Use of the RMIB Infrastructure

The initial knowledge is that the uxObj class is realised by agents with the name SMA-<host> (SMA stands for System Management Agent). We would like to see how many users are currently logged in at the host "athena". This means the application name we are interested in is "SMA-athena". Note that this is not a violation of location transparency since we do not specify the location where that application runs. It might *not* run at host athena but communicate with through another protocol i.e. be a Q-Adapter in TMN terms. We first instantiate a RMIBAgent and bind through it to "SMA-athena". The RMIBAgent knows the local domain and will construct the directory name for the SMAP object of that application as explained in Chapter 2 e.g. `c=UK@o=UCL@ou=CS@cn=SMA-athena`. It will then contact the directory, retrieve the presentation address of the contained SMAE object and try to connect to that address. If any of those remote operations fails, NOTOK will be returned from the bind method and the program subsequently will exit through *error*. If an association is established, it will be "cached" for 60 seconds which is the default caching period.

Now we are bound to that system and would like to retrieve the *nUsers* attribute of the *uxObj* instance, but we do not know its name. We know though from its GDMO specification that it

should be in the first level of the MIT as it is "bound" to the class *system* [X721] which is always at the top of the MIT. We then perform then a m-get operation to the top MIT object whose LDN is "empty", request the first level subordinates, apply the filter *(objectClass=uxObj)* and request no attributes. This will result in two CMIP PDUs passed back to the RMIBAgent, one with the uxObj instance information and one being the empty terminator (note that scoping results in linked replies even if one object only is selected). This request took place in a synchronous fashion and we now know the name of the uxObj instance. We subsequently perform another synchronous m-get operation to the latter, request the nUsers attribute and print its value. We could have retrieved the nUsers attribute with the first operation which would have avoided the second one, but we are trying to demonstrate more aspects of the RMIB API.

We subsequently bind to another application, called STATSRV, which contains an instance of the simpleStats class. The RMIB agent terminates first the connection to SMA-athena which is still established and then goes through the same process of constructing the directory name of the application, retrieving its address and connecting to it. It should be noted that the connection to the directory server exists already as it has been "cached-in" by the underlying infrastructure in a similar fashion. In the second example we know in advance the name of the simpleStats instance which is simpleStatsId=null. The latter is a convention used for single-instance classes i.e. the value of naming attribute to be *"null"*. We subsequently perform the "calcSqrt" action with argument 4 and print the result, which should be 2.

The above example, though simple, demonstrates the power and simplicity of the RMIB infrastructure. It should be noted that the O-O ASN.1 API contributes significantly to the relevant power and simplicity. Finally, it should be stated that this is a "weakly-typed" environment, where type mismatch errors are discovered only at run-time. For example, if the argument to the calcSqrt action was defined as "GraphicString arg("4")", the program would compile happily but the action method call would return NOTOK when validating the consistency of the arguments and would print a diagnostic error message. If, the reply was specified of type GraphicString, the program would compile happily, perform the action fine but it would either crash when trying to print the type or it would simply print garbage.

In an agent-level infrastructure such as the RMIB it is *impossible* to support strong typing since the relevant methods operate still at the CMISE level and precise ASN.1 types are not known. This observation brings us naturally to the next level of management infrastructure that operates at the managed object level and can be strongly-typed, the Shadow MIB.

3.5.4 The Shadow MIB Managed Object Level Infrastructure

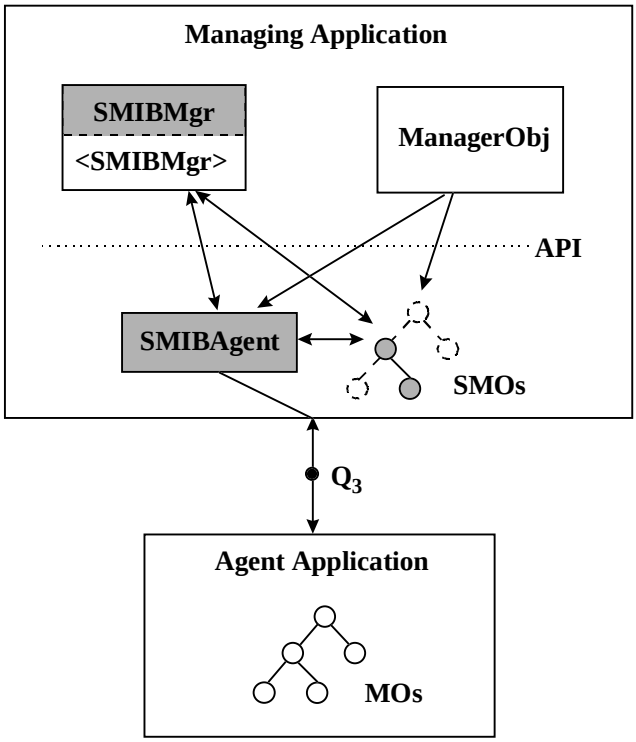
The idea for a facility that "caches" managed objects in manager applications was conceived early in the NEMESYS project, before the use of OSIMIS and TMN Q_3 protocols. At the time (1988-1989), Objective C [Cox86] was used which supports an object serialisation facility in a similar fashion to Java [Sun96]. The first NEMESYS management platform used Sun RPC [Sun88] to communicate management information in the form of serialised managed objects or simply serialised attributes, operation parameters and results. Based on this "management protocol", an application infrastructure was developed, known as the Management Unit Information Base (MUIB). This provided support for remote operations through local operations on "cache" objects. The reader may observe the similarity of this approach to the much more recent Java Remote Method Invocation (RMI). The main contributor of this idea and implementer of the relevant infrastructure was P.-E. Stern of GSI Erli, France.

When OSIMIS was introduced in the last phase of the NEMESYS project, the author thought of reproducing the MUIB functionality over true Q_3 interfaces in order to support novice application developers by hiding the details of Q_3. The concept was named Shahow MIB (SMIB) and a detailed specification was produced, following a weakly-typed approach. The concept was embraced with enthusiasm by IBM ENC, Heidelberg, who were a NEMESYS partner, and D. Jordaan provided an embryonic OSIMIS-based implementation of the initial SMIB specification as a proof of concept. This was not used in NEMESYS but IBM ENC took the concept further a few years later, resulting in the Object-Oriented Interface (OOI) specification [Holb95] which influenced the NMF GDMO/C++ API [Chat97]. The author independently explored the concept further during 1993, together with A. Carr of Cray Communications, UK, in the context of the RACE ICM project. The relevant issues were revisited and a new specification was produced and implemented in OSIMIS by A. Carr. It should be mentioned that the SMIB infrastructure was never released with OSIMIS since it was more of an experimental prototype rather than robust infrastructure for building TMN applications.

3.5.4.1 The SMIB Model

The key characteristic of the SMIB approach is that it supports a local cache of (parts of) the remote MIB within the manager application in the form of Shadow Managed Objects (SMOs) [Pav94b]. These are administered by a Shadow MIB Agent (*SMIBAgent*) which models the "binding" to that system, in a similar fashion to the RMIBAgent. In fact, the SMIBAgent contains a RMIBAgent instance which it makes available to its users. This means that the SMIB

framework is a clear superset of the RMIB one. The difference between this approach and the IBM OOI or the NMF GDMO/C++ is that the RMIB and SMIB infrastructures are distinct, so that the RMIB infrastructure can be used *without* the SMIB. In those frameworks, the relevant concepts are tightly-coupled and, as such, inseparable.



<SMIBMgr> - specific class derived from SMIBMgr

Figure 3-10 Shadow MIB Model and Interactions

The Shadow MIB model is shown in Figure 3-10, in the form of co-operating object instances. The API can be both synchronous and asynchronous, with the *SMIBManager* class providing the abstract callback interface. It should be emphasised that only the MOs the application is interested in are shadowed; these are shown in shaded form in Figure 3-10. SMOs can be created either explicitly, through a request to the SMIBAgent with their name, or implicitly as a result of a scoped request either to the SMIBAgent or to another SMO. The SMOs typically hold information regarding the name, class and attributes of the "master" MO in the agent. Note that the SMIBAgent encapsulates and uses an instance of RMIBAgent, which provides O-O access to the real remote agent. The RMIBAgent instance is not shown in Figure 3-10 but this relationship is shown in the following figure.

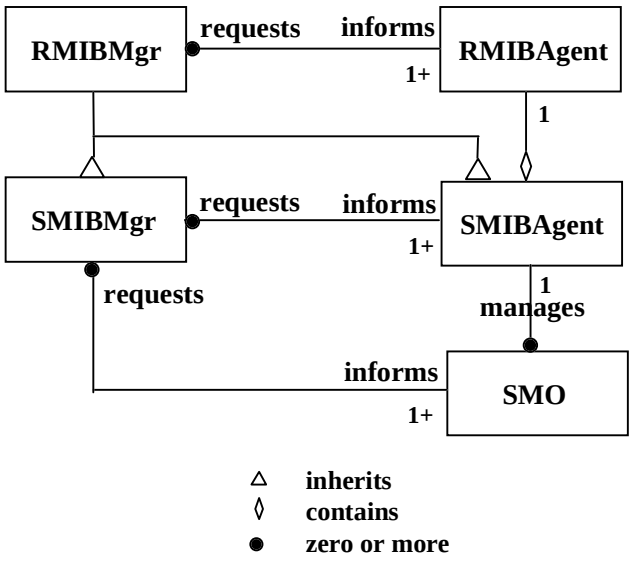


Figure 3-11 OMT Relationships of the RMIB and SMIB Classes

Figure 3-11 shows an OMT diagram of both the RMIB and SMIB models. Relationships other than inheritance and containment are explicitly named while instance relationships include their cardinality. The SMIBAgent contains one RMIBAgent and it is itself a RMIBManager in order to receive events and asynchronous results. The SMIBManager derives from the RMIBManager so that it can receive events and asynchronous results both from the SMIBAgent and the contained RMIBAgent (the SMIB model is a clear superset of the RMIB one). The SMIBAgent manages a number of SMOs which are members of the relevant group. Finally, the manager objects *request* information from the agent and shadow objects and are *informed* of events and asynchronous results.

Manager objects access typically the SMOs either in a local fashion, in order to retrieve values of cached attributes, or in a fashion that triggers a remote operation to the associated master MO (m-get, m-set, m-action). The creation and deletion of SMOs can also take place in two modes: a local mode, which does not affect the master MO, and a remote or real mode, which results in the creation and deletion of the master MO through m-create and m-delete. Operations with scope and filter can be performed either to the SMIBAgent, using explicitly the base object's name, or to an SMO, which in this case assumes the role of the base object. In the same fashion, event reports may be requested either from the SMIBAgent, when a general filter assertion is specified that spans different object classes and instances, or from a particular SMO, requesting events either from the individual instance or from all the instances of that class.

There are two different approaches for the design of an SMIB framework supporting the model described above:

- a weakly-typed approach, in which a single SMO class is used to model any managed object class in the remote agent; and
- a strongly-typed approach, in which specific SMO classes are produced for the corresponding MO classes through a GDMO compiler.

The weakly-typed approach is suitable for generic applications such as MIB browsers and its main advantage over the RMIB is that it provides a cache for the MO's attributes, alleviating the task of the application developer. Individual attributes can be requested to be updated at regular intervals, according to some predetermined schedule or through attributeValueChange and stateChange event reports if the MOC supports those.

The strongly-typed approach adds the advantage that attribute access methods and actions are strongly-typed. In addition, since the classes are produced through a GDMO compiler, the user may add methods for adaptive polling or other behaviour according to the application's management policy and the semantics of the particular object class.

3.5.4.2 The SMIB API

The Code 3-7 caption shows an example of using the SMIB infrastructure. The first part demonstrates the weakly-typed, generic approach which is very similar to the RMIB one. The second part demonstrates the strongly-typed, information model specific approach, supported through a GDMO compiler. The example is similar to the first one used in the RMIB case, i.e. retrieve the nUsers attribute from the uxObj instance, but we now also perform an "echo" action to that instance (the full class specification can be found in Appendix C). Incidentally, the echo action demonstrates how to say "hello world" in an OSI-SM/TMN distributed fashion!

After the SMIBAgent is instantiated and bound to the SMA-athena application, we request the creation of a SMO with name "`uxObjId=null`". In the general case, the SMIBAgent will transparently send a m-get request to the master object, retrieving its objectClass, allomorphs and packages attributes i.e. its *top* class [X721] part, in order to verify that the SMO is valid and populate it with the necessary "base" information. The second boolean argument in our call though indicates that we would also like to retrieve at the same time all its attributes, which means that the SMIBAgent will request all the attributes in this case. Having succeeded, we retrieve the nUsers attribute locally, which demonstrates the use of the shadow object as an attribute cache. We subsequently perform the echo action. You may note that the latter is extremely similar to the equivalent RMIB method in the Code 3-6 caption. In fact, there is no real advantage in the SMIB approach over the RMIB one regarding actions, attribute sets and object

creations / deletions. The key advantage is attribute value caching and relevant generic update policies which can be encapsulated in the SMO class realisation.

```
// weakly-typed, generic SMIB approach

SMIBAgent smibAgent;
smibAgent.bind("SMA-athena");

AVA* errInfo;
GraphicString arg("hello world"), *res;
Gauge* nUsers;

SMO* smo = smibAgent.getSMOHandle("uxObjId=null", True);

if (smo) {
    attr = (Gauge*) smo -> getAttr("nUsers", False);
    smo -> Action("echo", &arg, res, errInfo);
    // . . .
}

// strongly-typed, information model specific SMIB approach

SMIBAgentUx smibAgentUx;
smibAgentUx.bind("SMA-athena");

uxObjSMO* uxObjSmo = smibAgentStats.uxObjFactory("uxObjId=null");

if (uxObjSmo) {
    nUsers = smo -> get_nUsers(False);
    uxObjSmo -> echo(arg, res, errInfo);
    // . . .
}
```

Code 3-7 Example Use of the SMIB Infrastructure

The strongly-typed approach requires a GDMO compiler that will produce specific SMO-derived classes, e.g. uxObjSMO, and also a specific SMIBAgent for the particular information model. The latter is necessary in order to include strongly-typed methods for getting access to the relevant specific shadow objects, as demonstrated in the example. There will typically exist one specific SMIBAgent class for every information model the manager "knows" about e.g. SMIBAgentG774 for the SDH element model [G774]. Returning to the example, we bind to that system and request the creation of a shadow object in the same fashion as before. The key difference is that the relevant method (uxObjFactory) is now strongly-typed. The same is true for the subsequent attribute retrieval (get_nUsers) and the echo method call. For example, if the echo argument or result was declared of type Real by mistake, the program would not compile. This approach has significant advantages for program development by novice users.

3.5.4.3 Manager Mapping of GDMO to O-O Programming Languages

The SMIB approach requires essentially a "manager" mapping of the GDMO O-O abstract language to concrete O-O programming languages. We have implicitly proposed aspects of such

a mapping while discussing the SMIB model and explaining its features by example. We elaborate on this mapping and present an explicit proposal below:

- Managed object classes map to C++ classes. In the weakly typed approach, *every* GDMO class is modelled by the SMO class. In the strongly-typed approach, GDMO classes map to corresponding C++ classes with the *SMO* suffix, following exactly the same inheritance relationships as in GDMO. The *topSMO* class derives from SMO.

- Packages are not explicitly present in shadow managed objects. Their attributes, actions and notifications map onto methods of the containing managed object class as proposed below. These methods are produced for both mandatory and conditional packages. If a conditional package is not present in the MO instance, the associated SMO methods to that package should return an error.

- Attributes map to relevant access methods according to the access rights, i.e. *get*, *set*, *setToDefault*, and also *add* and *remove* for settable multi-valued attributes. Access could be either local or remote. Methods should be provided to allow the remote get and set of more than one attributes at a time i.e. resulting in one CMIS m-get or m-set request. In the strongly-typed approach, specific methods should be generated for each attribute with the exact ASN.1 type e.g. “Gauge* nUsers_get()”, “int wiseSaying_set(GraphicString*), etc.

- Actions map to relevant access methods, with an input argument modelling the GDMO “action information” and an output argument modelling the GDMO “action reply”. In the strongly typed approach, specific methods should be generated for each action with the exact ASN.1 types e.g. “int echo(GraphicString*, GraphicString*&)”.

- In the weakly-typed approach, the SMIBManager provides a generic *EventReport* method, inherited from the RMIBManager as in the Code 3-5 caption. This method is used to “push” a notification to the RMIBManager. In the strongly-typed approach, notifications should be mapped to generated methods of a specific SMIB manager class for that information model. The notification methods should take event information and reply parameters with specific ASN.1 types.

- Parameters for attributes, actions and notifications indicate essentially MOC-specific errors. These should be mapped onto error parameters, with type-value information for the relevant methods (OSIMIS maps those to AVA error arguments).

- Name bindings do not map explicitly to shadow managed objects. Every SMO knows its name and, as such, its position in the MIT. SMOs should provide facilities for getting the handle of the superior SMO or the handles of subordinate ones through scoping and filtering.

The proposed mapping highlights a "failing" of GDMO as an abstract O-O specification language. Arbitrary methods of distributed managed objects are modelled in GDMO through actions. These take only one input argument, the action "information", and return one output argument, the action "reply". In the case of actions with more than one input or output arguments, they have be artificially combined into *one* argument using typically the ASN.1 SEQUENCE type. This results in a non-natural mapping of the arguments to C++ method signatures. As an example, the simpleStats class supports a calcMeanStdDev action which calculates the mean and standard deviation given a list of real numbers (see Appendix C). The action reply in this case should comprise two distinct output arguments, the *mean* of type REAL and the *stdDev*, again of type REAL. This is not possible in GDMO, so one is forced to define a new output argument of type MeanStdDev, which is of type SEQUENCE and comprises two elements of type REAL for the mean and standard deviation. The "contrived" ASN.1 type and the resulting action signature in C++ are shown in the Code 3-8 caption, together with the ideal method signature for this type of action.

```
-- ASN.1 definition for "calcMeanStdDev" action reply

MeanStdDev ::= SEQUENCE {
    mean        REAL,
    stdDev      REAL
}

// resulting action signature in C++

int calcMeanStdDev(RealList* info, MeanStdDev*& reply);


// ideal action signature for the "calcMeanStdDev" action

int calcMeanStdDev(RealList* rList, Real*& mean, Real*& stdDev);
```

Code 3-8 The C++ Mapping "Problem" with GDMO Actions

The solution is to modify the GDMO Action template to allow for more than one information and reply types. In this case, the action and reply information at the CMIS/P level will be a SET OF ActionInfo and a SET OF ActionReply respectively. This implies a modification of CMIS/P in addition to GDMO. The Code 3-9 caption depicts both the grammar for the new template and an example for the calcMeanStdDev of the simpleStats class. The proposed syntax would result in the more natural C++ signature for that action as explained above.

```
<action-label> ACTION
    [MODE CONFIRMED ;
    ]
    [PARAMETERS         <parameter-label>
                      [,<parameter-label>]* ;
    ]
    [WITH INFORMATION SYNTAX
                        [<argument-label>] <type-reference>
                      [,[<argument-label>] <type-reference>]* ;
    ]
    [WITH REPLY SYNTAX
                        [<result-label>] <type-reference>
                      [,[<result-label>] <type-reference>]* ;
    ]
REGISTERED AS object-identifier ;


calcMeanStdDev ACTION
    MODE CONFIRMED;
    WITH INFORMATION SYNTAX
        numbers UCL-ASN1Module.RealList;  -- SET OF REAL
    WITH REPLY SYNTAX
        mean    UCL-ASN1Module.Real,      -- REAL
        stdDev  UCL-ASN1Module.Real;      -- REAL
REGISTERED AS { uclAction 702 };
```

Code 3-9 Proposed Modification of the GDMO Action Template

A weakly-typed SMIB approach was prototyped during 1994 in the ICM project, validating the design and GDMO to C++ mapping presented above. The reason the weakly-typed approach was chosen was practical: the strongly-typed approach needs a GDMO compiler with an SMIB-specific back-end for code generation. At that time (early 1994), the OSIMIS GDMO compiler was not entirely stable. The SMIB prototype was incomplete, so it was never released with OSIMIS. On the other hand, this implementation validated the relevant concept and model.

3.5.5 The Tcl-RMIB Scripting Manager Infrastructure

The Tool Command Language (Tcl) [Oust94] emerged in 1994 as a general-purpose scripting language that interfaces well with C/C++ and can be extended with commands implemented in those languages. The key advantage of Tcl is its interpreted nature which accelerates the development process and may also support code mobility. Tcl is a weakly typed language based on strings and it is relatively easy to use compared to compiled programming languages such as C/C++. On the other hand, it is about an order of magnitude slower compared to those.

One of the main reasons behind the popularity of Tcl is its extension for the MIT X Window System and MS Windows, known as Tk [Oust94]. This is a graphical toolkit that extends the core Tcl facilities with additional commands for constructing GUIs. The Tk toolkit exists in the form of a collection of display "widgets", providing a user-friendly way to compose graphical primitives. The combination of Tcl and Tk presents a suitable environment for the rapid construction of GUI-based applications. A Tcl extension of the compiled higher-level manager infrastructures could support the rapid construction of TMN WS-OS applications.

We have chosen to provide RMIB extensions to Tcl since RMIB is the primary high-level manager infrastructure. Providing those Tcl extensions was relatively straightforward since the RMIB infrastructure supports already string parameter passing for distinguished names, filters, and attribute, action, notification and specific error values. A more difficult aspect is the mapping of the object-oriented RMIB model to the procedural Tcl one. Given the fact Tcl allows to reference and invoke procedures in a similar fashion to the C language "function pointer" facility, it is possible to emulate object-oriented style of programming in Tcl. [Meyer88] discusses how object-oriented features can be supported in procedural languages that support a facility equivalent to "function pointers".

The resulting language is called Tcl-RMIB [Tin95][Pav96d] and allows interactions with remote agents using the same abstractions as the compiled C++ RMIB. Tcl-RMIB provides a number of extension commands to Tcl in which the underlying RMIB C++ objects are manipulated by their interpreted counterparts. Commands are provided for the user control (creation, deletion, etc.) of the agent and manager objects locally in Tcl and for performing management operations through those objects. Both synchronous and asynchronous modes of operation are supported. The management commands are shown in Table 3-5. Their syntax owes much to that of the generic command line manager programs which also influenced the string-based CMIP that was described in section 3.3.2.4.

Management Command	Description
m_bind *agentId* ?-a *agentName*? ?-h *hostName*?	binds to a remote agent
m_unbind *agentId*	unbinds from a remote agent
m_get *agentId* ?-c *class*? ?-n *name*? ?-s *scope* ?*sync*?? ?-f *filter_expr*? ?-a *attribute* ...? ?-m *managerId ?-o?*?	allows the M-Get invocation; may return an invoke identifier for call-back correlation
m_set *agentId* ?-c *class*? ?-n *name*? ?-s *scope* ?*sync*?? ?-f *filterExpr*? ?-w|d|a|r *attrType*?=*attrValue*? ...? ?-m *managerId*?	allows the M-Set invocation; may return an invoke identifier for call-back correlation
m_action *agentId* ?-c *class*? ?-n *name*? ?-s *scope* ?*sync*?? ?-f *filterExpr*? ?-a *actionType*?=*actionValue*?? ?-m *managerId*?	allows the M-Action invocation; may return an invoke identifier for call-back correlation
m_delete *agentId* ?-c *class*? ?-n *name*? ?-s *scope* ?*sync*?? ?-f *filterExpr*? ?-m *managerId*?	allows the M-Delete invocation; may return an invoke identifier if call-back correlation
m_create *agentId* ?-c *class*? ?-n *name* | -s *superiorName*? ?-r *referenceName*? ?-a *attrType=attrValue* ...? ?-m *managerId*?	allows the M-Create invocation; may return an invoke identifier for call-back correlation
m_notify *agentId managerId* ??-c *class*? ?-n *name*? ?-e *eventType*? | ?-f *filterExpr*? | ?-a?? ?-s?	allows the call-back registration of a manager to receive event reports and also request termination of notification; event reporting may be cancelled (using -s).

Table 3-5 Tcl-RMIB Management Commands

The Tcl-RMIB approach has semantics of a weakly-typed dynamic invocation interface, in a similar fashion to the RMIB one. It mirrors the functionality of the compiled RMIB approach and provides an analogous interpreted interface. The full CMIS expressive power is available i.e. scoping, filtering, linked replies and fine-grain event reporting based on filtering. Control over management associations is provided while it may be also left to the underlying infrastructure. A generic list structure that applies to all the requests is used for replies and errors as described in detail in [Tin95].

While the Tcl-RMIB approach is agent-oriented in a similar fashion to the RMIB one, higher-level managed object-oriented interpreted approaches are possible e.g. a Tcl-SMIB. It should be mentioned that while Tcl was thought to be the definitive interpreted scripting language 2-3 years ago, it has been recently overshadowed by the emergence of Java [Sun96]. The latter has all the

advantages of Tcl/Tk, including facilities for rapid GUI construction. In addition, it is object-oriented, it compiles into intermediate "byte-code" representation which results in better performance and will be eventually supported by Java-capable hardware. In summary, Java is the language to be used for TMN WS-OSs in the future. A Java-RMIB infrastructure is perfectly feasible to provide since Java can interface to C++ (though not as easily as Tcl). In addition, the fact that Java is object-oriented and has syntax similar to C++ will result in exactly one-to-one mapping between the Java-RMIB and the RMIB features while equivalent methods will have similar syntax.

The Tcl-RMIB infrastructure was designed together by T. Tin and the author while it was implemented by T. Tin in early 1995. It was subsequently used successfully in the ICM project for developing WS-OS applications. It was also publicly released with OSIMIS-4.0 [Pav95b], so it was also used by the wider community. It demonstrates that the concepts and facilities of the high-level manager infrastructure are general enough to allow mappings to programming languages other than C++. In addition, it suggests that interpreted high-level manager support in languages with built-in GUI capabilities is an important ingredient of a TMN distributed software platform.

3.5.6 The Management Information Repository

While discussing both CMISE and higher-level manager infrastructures and APIs, we referred to necessary knowledge of the GDMO information model. This is required in order to be able to map user-friendly string names of attributes, actions, events and specific error names to the respective object identifier and the corresponding syntax. This information should be made available to such infrastructures in a data-driven fashion so that they are able to cope with new information models without changes in their code. As such, it needs to be stored in a form of database that can be accessed at run-time, hence the term *management information repository*.

This information is typically produced by compiling a GDMO information model and has to be added to the information repository if it is not already there. This process can be automated so that every time an information model is compiled, the repository is automatically updated. A manager infrastructure, such as a high-level CMISE implementation or the RMIB, typically reads this information when starting up and builds-up a core memory image of the relevant mappings. Access to this information is provided either through the name or the associated OID, which serves as the "key". A typical realisation of this information is through a hash table in order to support fast access to the relevant data.

With this information in place, an attribute name such as the "wiseSaying" of the uxObj class can be mapped onto the 2.37.1.1.7.503 OID and the GraphicString corresponding syntax. This allows, for example, to check the consistency of an attribute value assertion that the user constructs in order to set the value of this attribute. In addition, it allows the infrastructure to map the attribute name to the corresponding OID. In the case of indications, it allows the infrastructure to map the OID to the user friendly name and to construct the correct C++ type with the value i.e. GraphicString in this case.

This is the minimum information required for supporting infrastructures such as the RMIB and the weakly-typed SMIB. Additional information about the GDMO objects may also be part of the repository, supporting further facilities. For example, it might be desirable to perform additional checking in a manager application, before a request is forwarded to the agent. If, say, a manager tries to perform a calcSqrt action on a uxObj instance which the latter does not support, it should be possible to detect this locally and generate an error. This can only be achieved if the repository contains additional information about managed object classes e.g. the packages, attributes, actions, notifications and specific errors they support. This information is typically referred to as *meta-data* or *meta-information* since it describes the information model itself.

The key benefit of having access to this type of information in manager applications is that requests can be validated locally, before sending them to the agent. An additional benefit has to do with generic applications such as MIB browsers. In those applications, it is possible to present this meta-information to the user in order to support better a particular request. For example, when creating a new object instance, the user may be presented with a list of those attributes that may be initialised so that s/he may supply initial values. If this information is not available, the human users need to have the GDMO model "in their head", which is certainly not possible.

This meta-information can be produced through the GDMO compiler, stored in the repository and read by manager applications at start-time in order to build an image in core memory. Incidentally, the same type of information is required in agent applications. The core memory representation of this type of information can be a tree of meta-class objects that model accurately the GDMO inheritance relationships, together with a container object that provides access to those through the class name or OID.

```
# name              OID
uxObj               uclManagedObjectClass.50
uxObj-system        uclNameBinding.50
uxObjPackage        uclPackage.50

# attr name         OID                 attr type
uxObjId             uclAttributeID.501  SimpleNameType
sysTime             uclAttributeID.502  UTCTime
wiseSaying          uclAttributeID.503  GraphicString
nUsers              uclAttributeID.504  ObservedValue

# action name       OID                 info type        reply type
echo                uclAction.502       GraphicString    GraphicString
```

Code 3-10 Structure of the OSIMIS Management Information Repository

The approach followed in OSIMIS was that of the minimum possible meta-information i.e. the table mapping names to OIDs and ASN.1 types. This means that it is not possible to detect "mismatch" errors locally, within manager applications. Requests are always sent to the agent that will detect the error. This is acceptable, since such errors are typically development type errors hat will be sorted out before the application is deployed in a real system. The real drawback is that generic OSIMIS applications such as the MIB browser [Pav92a] do not have access to class information that could be used to provide a friendlier interface as describe above.

Finally, the Code 3-10 caption shows the OSIMIS structure of the management information repository using as an example the uxObj class which is formally specified in Appendix C. There are *three* types of "database records": those for classes, name bindings and packages which do *not* have an associated syntax; those for attributes that have *one* associated syntax; and those for actions and notifications that may have *two* associated syntaxes, one for the information and one for the reply. Note that the OID prefixes are also part of the database so that OIDs are fully resolved. Note also that the objectCreation, objectDeletion and attributeValueChange notifications, which the uxObj class supports, are absent. This is because these are generic ones, introduced by the Object Management SMF [X730] and specified in [X721]. As such, they are already in the X.721 part of the repository. This means that the repository is constructed in a modular fashion.

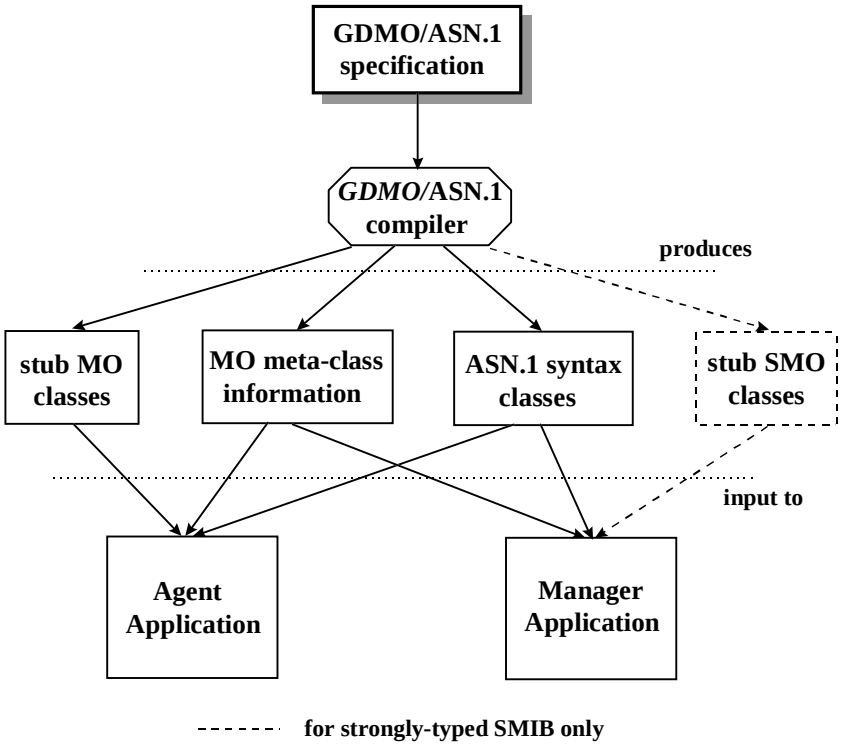


Figure 3-12 Information Produced Through the GDMO/ASN.1 Model

We will close this section with a note on realisation aspects and procedures. While these have been pioneered in OSIMIS, they are also followed by most commercial TMN platforms. Every time an agent application with a new management information model is implemented, the following steps are followed. The GDMO/ASN.1 model is compiled first in order to produce the "stub" C++ managed object classes to support the realisation of the agent part, as it will be described in section 3.6. The compilation of the ASN.1 part will result in C++ classes for the new ASN.1 types this information model introduces, produced through the O-O ASN.1 compiler. The managed object classes and associated syntax classes need to be kept separate, since the syntax classes are used by both agent and manager applications. If the relevant TMN platform supports a strongly-typed SMIB infrastructure, shadow object classes will be also produced. Finally, the meta-class information will be produced for the management information repository. The ASN.1 classes and the management information repository are used by both agent and manager applications, as shown in Figure 3-12. Note that the information model which is "built-in" in both the agent and manager logic constitutes the shared management knowledge which together with the supporting Q_3 protocol stack achieves interoperability, as discussed in Chapter 2.

High-level manager infrastructures will need to be made "aware" of the new ASN.1 syntaxes. In software terms, this means that the relevant library should be linked with the application and that

the RMIB code should be "told" somehow of the new syntaxes. In OSIMIS the latter involves changing *one* line of code in a header file in order to include another header file produced by the O-O ASN.1 compiler. This is not an issue for specific manager applications since these will be implemented "from scratch" with information model specific logic and behaviour. Generic manager applications though, such as an MIB browser [Pav92a], will need to be slightly modified and re-linked with a new syntax library every time an information model with new syntaxes is encountered.

While this is acceptable in most cases, there may exist situations in which it is not desirable to stop a generic manager application, re-link it with new syntaxes and restart it. Commercial TMN platforms support facilities so that new ASN.1 syntaxes can be incorporated "on the fly". This is done through additional meta-data produced by the ASN.1 compiler for every syntax which "guide" the application how to *print* and *parse* relevant instances. In this case, whenever new names and types are encountered, the generic application needs to read again the information repository in order to "learn" how to manipulate those types. This type of behaviour is possible but introduces additional complexity. The author thought of accommodating this feature when designing the OSIMIS ASN.1 API but decided against it because of its complexity. The solution adopted in OSIMIS has an impact on generic applications which cannot deal with new information models and syntaxes in a fully dynamic fashion. On the other hand, this is acceptable in most real-life situations.

3.5.7 Summary

In this major section we looked at issues behind realising high-level manager infrastructures and APIs that hide the complexity of the underlying CMIS service without sacrificing any of its expressive power. We identified two types of abstractions and relevant APIs:

- weakly-typed APIs that model a remote agent application; the relevant OSIMIS infrastructure is known as the Remote MIB (RMIB); and
- both weakly-typed and strongly-typed APIs that model individual managed objects in a shadow or proxy fashion; the relevant OSIMIS infrastructure is known as the Shadow MIB (SMIB).

Both these infrastructures have a lot of commonality with emerging distributed object frameworks such as CORBA [CORBA] and DCOM [DCOM], providing an easy to use APIs and supporting access and location transparencies. Some of the relevant features were demonstrated through examples in the Code 3-6 and Code 3-7 captions. A few lines of code are enough to perform

operations on remote managed objects, in a similar fashion to emerging distributed object frameworks. In addition, the CMIS scoping, filtering and fine-grain event-reporting power is available through the same simple APIs. This is the major contribution in this section.

Having examined aspects of the RMIB and SMIB infrastructures in detail, we conclude here that the right approach for the SMIB is the strongly-typed one, which has serious advantages over the RMIB approach. Strong-typing can be combined with user-included behaviour in the relevant shadow classes, alleviating the task of the management application developer. The weakly-typed SMIB approach is syntactically very similar to the RMIB one as demonstrated through the examples. In addition, the generic "caching" behaviour of that approach is only of limited usefulness, since it can not exploit the semantics of the management information.

We have also shown how to map those infrastructures to interpreted scripting languages with "integrated" GUI development support such as Tcl/Tk. The latter though has been rendered "obsolete" by the emergence of Java, which is now the prime candidate for TMN workstation development. Since Java is an object-oriented language, it would be natural to map to it the RMIB and SMIB models presented in this section.

In summary, we demonstrated in this section how to provide an object-oriented platform infrastructure for the manager part of the OSI-SM/TMN manager-agent model. As a result of the relevant research towards providing such infrastructures, we identified weak aspects in the CMIS/P and GDMO specifications for which we proposed relevant solutions. The RMIB and SMIB models and APIs have been input to the NMF TMN/C++ effort [Chat97]. This proposes a industrial solution to the problem of TMN high-level manager APIs which has a lot of commonality with the models presented in this section.

3.6 Issues in Realising Object-Oriented Agent Infrastructures

3.6.1 Introduction

In the previous section we concentrated on issues related to the realisation of object-oriented infrastructures for applications in manager roles. The key issue was to provide object-oriented abstractions of whole agent applications or managed objects, giving the illusion that these objects were available in the local address space. The result was user-friendliness and easy programmability that hides underlying protocol aspects; this is an important attribute of distributed object frameworks as identified in section 3.2.2.

A similar infrastructure is required for applications in agent roles. In this case, the behavioural aspects of managed objects should be shielded from CMIS/P access aspects as much as possible, allowing designers and implementors to concentrate in the intelligence of applications rather than been concerned with the underlying access details and complexity. A key issue in such an infrastructure is the mapping of the GDMO abstract language to a concrete object-oriented programming language such as C++.

Other important issues in realising agent infrastructures are the following: support for CMIS access aspects such as name resolution, scoping, filtering and linked replies; support for the allomorphic behaviour of object instances; support for different models in maintaining the consistency of managed object attributes and associated resources; support for managed object persistency; and support for the systems management functions and in particular for event reporting and logging.

The realisation of object-oriented agent infrastructures was an issue that attracted significant attention from the research community. The author pursued early research in this direction which resulted design and implementation of the OSIMIS Generic Managed System (GMS) [Pav91a]. Related research work is presented in the next sub-section, while the following sub-sections discuss the relevant issues and present a concrete proposal for a generic agent infrastructure.

3.6.2 Related Work

The author's research work in realising object-oriented agent infrastructures preceded other research work in this area. The fundamental principles behind agent infrastructures were presented in [Pav91a] and were validated by the early implementation of the OSIMIS Generic Managed System (GMS). This work was enhanced and refined over the years and has been subsequently presented in more detail in [Pav93a], [Pav95a] and [Pav96b]. Related research work in this area is described in chronological order below.

[Nakai91] proposes a GDMO MIB editor tool which acts also as GDMO/ASN.1 compiler, producing a *data dictionary* for managed object classes. This is stored in a relational database and it is parsed into core memory in order to validate the consistency of operations to managed objects. It forms part of the agent process, which also contains a "protocol processor" and the managed objects.

[Nakak91] proposes an agent infrastructure which contains the following modules: the protocol processing module, the MIT module, the MIB access module and the managed object module. The interesting aspect of this work is the separation of the MIT representation from the managed objects themselves; this approach which has been later adopted by a number of products. A GDMO/ASN.1 compiler called MINT is used to parse specifications and produce stub class definitions. The programming language used is *superC*, a proprietary object-oriented extension of the C language.

[Newn92] describes an agent object-oriented infrastructure in C++ which is built over the XOM/XMP C-based API. This resembles a lot to the OSIMIS managed object stubs. The realisation of the top class provides an API to the "object manager", which performs the CMIS/P functions. It also defines the get, set and action methods in polymorphic fashion, i.e. as C++ virtual methods, so that derived classes can redefine them and add behaviour. The proposed design was validated through a prototype.

[Doss93] describes the design of an agent prototype which is driven by a GDMO/ASN.1 compiler. The internal decomposition of the agent functionality is presented only briefly but the mapping of GDMO to C++ is presented in some detail. C++ classes model the GDMO classes and packages while C++ classes model the ASN.1 attribute, action and notification types. Class specialisation is supported through "class doubling" i.e. behaviour is not added to stub classes produced by the GDMO compiler but a new class needs to be derived from the produced one. The

approach is similar to the CORBA Interface Definition Language (IDL) to C++ mapping. Aspects of this work were incorporated in Alcatel's ALMAP TMN platform.

[Deri95] discusses in detail the mapping of the various GDMO templates to C++ classes. It also discusses how filtering assertions can be supported by a generic attribute class, similar to the Attr class presented in section 3.4. The approach reflects the authors' experience from developing various commercial and public domain OSI-SM infrastructures. It is quite similar to the OSIMIS GMS apart from the fact that conditional packages are modelled by separate C++ classes. The acknowledgements state that "*the importance of the OSIMIS platform is acknowledged, since it has greatly contributed to the diffusion of OSI-SM and has introduced a number of concepts now adopted by many commercial implementations*".

[Flau95] discusses the object-oriented aspects of DEC's TeMIP TMN platform. A particularly interesting aspect is that the management applications themselves can be distributed, based on a proprietary "object broker" model which supports a dynamic invocation interface and offers "intra-application" location transparency. The whole framework is data-driven through dictionaries so that new functionality can be added on the fly.

[Feri96] describes IBM's Netview TMN platform and its support environment for hybrid agent/manager applications e.g. TMN OSs. Its key feature is the MIBcomposer front-end GUI. This serves as GDMO/ASN.1 editor and compiler but can be also used to associate behaviour to managed object classes. The agent environment is modular and the functionality of new classes can be introduced without having to shutdown the agent for a "cold start".

[Chat97] describes the NMF GDMO/C++ API in agent role. The MIT is kept separate from the managed objects so that parts of an agent application may be distributed. Behaviour is added through separate classes which are derived from the GDMO produced ones. This approach is similar to the CORBA IDL to C++ mapping. The whole framework bears a lot of similarities to the OSIMIS GMS, which was input to the relevant standardisation work.

3.6.3 The Overall Architecture

The key aspect behind an object-oriented agent infrastructure is to provide an environment which supports the development and deployment of managed objects that form an agent "cluster". Management application developers should be able to concentrate in the provision of the associated behaviour, without being concerned about the CMIS access details and underlying protocol aspects. This implies that the APIs for such an infrastructure should be defined at the managed object level, in a symmetrical fashion to the Shadow MIB infrastructure. The managed objects in this case are the *master* objects, implementing the information aspects dictated by a TMN Q_3 or X interface.

Given the object-oriented nature of GDMO as information specification language, it is most natural to map the abstract managed objects onto concrete C++ object instances. The exact mapping of the various GDMO features to a language like C++ is a difficult task because of the object model differences; this mapping will be discussed in detail in the next section.

A managed object should be able to respond to the CMIS get, set, action and delete primitives, should know its name, should keep handles to its superior and first level subordinate objects in the MIT and should be able to evaluate filters according to the Management Information Model (MIM) [X720]. We have thus identified a core component of an agent application, the *management information tree*.

An object instance contains information pertinent to that instance i.e. its attributes and other instance variables that relate to its behaviour and state. Management requests passed to it need to be validated e.g. verify that the relevant attributes or action type are supported by the instance, convert the raw *any* values to specific types according to the associated ASN.1 syntaxes, etc. Keeping this *meta-information* on a per instance basis means that it would be "n-plicated" for *n* instances of the same class. Since the information is related to the class rather than the instance, it should be kept in objects which model the managed object classes. We have thus identified another agent component, the *meta-class* objects. Given the fact that classes are related through inheritance, meta-class objects should be organised in a hierarchy that models the GDMO inheritance tree. These class objects can also serve as "factories" for the relevant MO instances.

Access to managed object instances should be provided through a component that supports the CMIS/P functionality i.e. performs CMIS/P PDU processing, resolves distinguished names to MO handles, evaluates scoping and filtering, provides access control, takes care of atomic transactions and forwards replies and event reports to manager applications. We will call this

component the CMIS or MIB agent. This can be modelled by a single C++ object instance that encapsulates the Q_3 protocol stack. The CMIS or MIBAgent is in fact similar to the RMIBAgent in manager applications. The difference is there is only a single instance per agent application.

A final consideration is event reporting and logging. EFDs and logs will be obviously part of the MIT but the notification processing function, as specified in [X734][X735], should be a separate object which receives notifications and evaluates them. It may subsequently instruct an EFD to forward the event report through the MIB agent or instruct a log to create a new log record. The Notification Processing (NP) processing function or object can be thought as part of the MIT component. It is "invisible" though by the MIBAgent which accesses only managed objects.

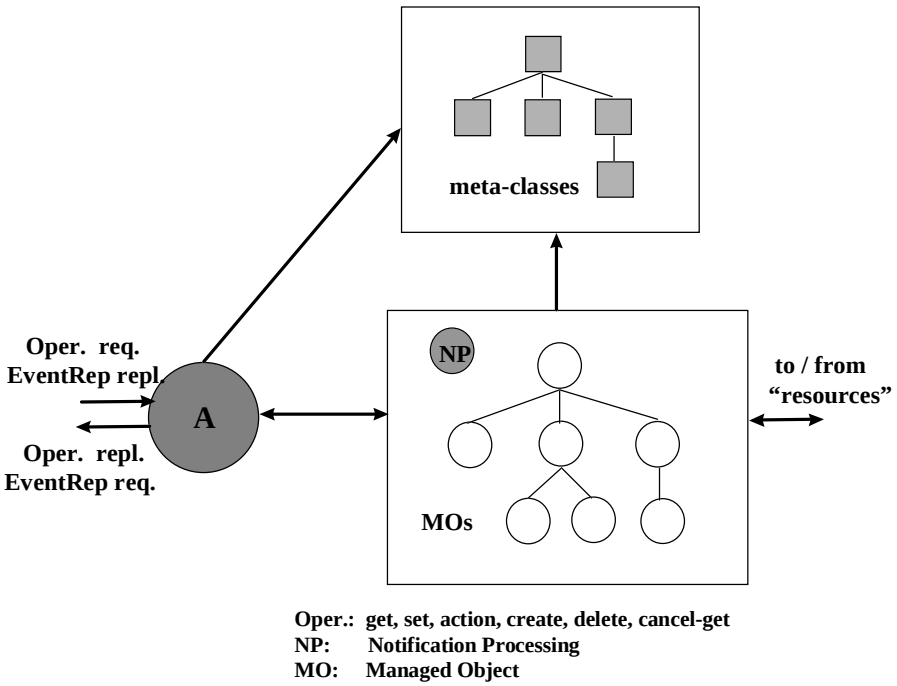


Figure 3-13 Object-Oriented OSI-SM Agent Decomposition

The initial decomposition of the agent application is shown in Figure 3-13. A management request always arrives to the agent object. In the case of a m-create, the agent resolves the superior and reference names to MO handles, locates the relevant meta-class object and requests for the object instance to be created. In the case of any other request, the agent resolves the base object name to a MO handle, evaluates scoping and filtering, checks the access control rights, performs the operations and returns the replies. A notification emitted by a MO is passed to the notification processing function which may instruct an EFD to forward it through the agent. If the notification is confirmed, the agent will pass eventually the confirmation back to the EFD.

An important design decision concerns the MIT representation of a MO and the actual MO itself. In the architecture presented above, these are mapped to the same C++ object instance. Another approach would be to keep them separate so that the managed objects could be in different operating system processes, using a proprietary distribution mechanism. In this case, the "core agent", i.e. the agent object, MIT representation and class dictionary could be in one operating system process while the managed objects and associated meta-class objects could be distributed in different processes. This approach has been adopted by a number of commercial products, with distribution supported through proprietary lightweight object brokers. OSIMIS and a number of other products follow a more "monolithic" approach, in which an agent application maps always to a single operating system process.

There are advantages and disadvantages with both approaches. The obvious advantages of the distributed approach is scalability and modularity: very large agent applications can be physically distributed. In addition, new functionality may be added to an agent in terms of new or modified classes by simply "detaching" and "attaching" new processes without having to shutdown the agent. This is an important requirement in telecommunications environments: a "cold start" of the management part of a network element may be undesirable. The key drawback of this approach is it complicates the agent architecture, necessitates the design of an object broker and results in slower response times. This approach goes to some extent beyond the OSI-SM model and introduces an "intra-application" distributed object framework, with Q_3/X interfaces retained for "inter-application" interoperability.

The author considered distributing the OSIMIS agent infrastructure but this meant essentially designing and implementing an object broker in addition to OSI-SM. The whole concept behind OSIMIS was to prove the feasibility, implementability, economy and performance of OSI-SM and not to provide a more general framework for distributed objects. Now that CORBA platforms exist, it would be relatively easy to distribute an agent application and this is a direction that a number of TMN platform vendors are taking. On the other hand, if CORBA is to be used within an application, it might be also used across applications and replace OSI-SM completely. This possibility is examined in detail in Chapter 4.

3.6.4 A GDMO to C++ Mapping for Managed Objects

In section 3.5.4 we discussed the mapping of GDMO to *shadow* managed objects in manager applications. Here we consider the mapping to *master* managed objects in agent applications.

A managed object is part of the MIT, so it should keep handles to the superior and first level subordinate objects. In addition, it should know its relative name i.e. the name of its distinguished attribute and value. In distributed agent environments, the MIT is a tree of "minimal" objects that contain only this information together with a reference to the actual managed object. This reference is according to a broker mechanism and can be used to invoke operations to the object. In centralised agent environments, the managed object is part of the MIT and responds directly to management operations. OSIMIS takes the approach of a centralised agent. The functionality described above is provided by the *MO* class, which is the root of the C++ inheritance hierarchy for managed objects.

```
class MO
{
    // . . .

public:
    MO*         Resolve    (DName* name);
    int         CheckClass (OIDentifier* cmisClass,
                            Bool& allomorphic);

    MO*[]       Scope (CMISScope* cmisScope);
    Bool        Filter (CMISFilter* cmisFilter);

    CMISErrors  Get (OIDentifier* objClass,
                     int nattrs, OIDentifier*[] attrIds,
                     int& nres, CMISGetAttr[] resAttrs,
                     AVA*& errInfo);
    CMISErrors  Set (Bool confirmed, OIDentifier* objClass,
                     int nattrs, CMISSetAttr[] attrs,
                     int& nres, CMISSetAttr[] resAttrs,
                     AVA*& errInfo);
    CMISErrors  Action (Bool confirmed, OIDentifier* objClass,
                     CMISParam* actionInfo, CMISParam *actionResult,
                     AVA*& errInfo);
    CMISErrors  Delete (AVA*& errInfo);

    // . . .
};
```

Code 3-11 The MIB Agent to Managed Object Interface

Since an object knows its relative name and has access to its first level subordinates, it can provide a *find* or *resolve* method which maps a distinguished name to an object handle through a recursive descent algorithm. In the same fashion, it is easy to provide a method that evaluates the CMIS scope parameter. The CMIS filter can also be evaluated if this class has access to all the attributes of derived classes. The key methods of the MO class which provide an interface to the MIB agent object are shown in the Code 3-11 caption.

The GDMO *top* class [X721] maps to a top C++ class which derives from MO. All subsequent GDMO derived classes should derive from top and mirror the GDMO inheritance in C++. There exist two key issues with inheritance in GDMO and inheritance in C++:

a) how GDMO multiple inheritance will be handled in C++; and

b) how behavioural aspects of GDMO managed object classes will be implemented in the corresponding C++ classes.

The immediate answer to the first question is to use C++ multiple inheritance to model the GDMO multiple inheritance. The problem with this approach is that it will not work with languages that do not support multiple inheritance such as Smalltalk [Gold83] and Java [Sun96]. Even in C++, it is necessary to eliminate duplicate attributes which should only appear once in a MO instance . In addition, method resolution is required when the same method appears more than once in parallel branches of the inheritance tree. The OSIMIS approach is to collapse the GDMO multiple inheritance into single inheritance and implement this in the standard C++ fashion. This is also an approach followed by many commercial products and there exist various algorithms for collapsing multiple to single inheritance. This can be done either automatically, by a GDMO compiler with a built-in algorithm, semi-automatically by the human user guiding the compiler, or manually. OSIMIS uses the third approach i.e. the user has to collapse manually GDMO multiple to single inheritance.

Adding behaviour to a class can be handled either by adding methods to the relevant C++ class or by “class doubling”, as in [Doss93], [Chat97] and [CORBA]. In the latter case, behaviour is added through a derived class e.g. *discrImpl* for the *discr* class. The problem with this approach is that multiple inheritance becomes necessary for classes in levels deeper than two in the inheritance hierarchy. For example, the *eventForwDiscrImpl* class should inherit both from the *eventForwDiscr* class and from the *discrImpl* class. The OSIMIS approach is to include methods with behaviour rather than use class-doubling. This is done by files with the suffixes *.inc.h* and *.inc.cc* which should be present while compiling the GDMO specifications. This approach works with any object-oriented programming language that supports only single inheritance.

The next important issue concerns packages. An initial thought would be to map those to separate C++ classes, so that they become reusable entities. The problem is that while conditional packages are re-usable when considered as units of specification, their behaviour is in most cases dependent on the class in which they are contained. As such, the case for re-usability is weak while the complexity of a MO instance is increased. OSIMIS models the functionality of mandatory and conditional packages through the C++ class that models the GDMO class. This

means that packages do *not* map to separate C++ classes. An instance keeps state information regarding which conditional packages are "active" so that it can respond correctly to management requests.

Attributes map to C++ classes that model the corresponding ASN.1 types. When an object instance is created, the attributes of the mandatory and active conditional packages form an array. Such arrays exist for all the classes of the inheritance branch, e.g. for *top*, *discr* and *eventForwDiscr*, and their handles are passed to the MO part at construction time. Since the latter has access to all the attributes of the instance, it can evaluate CMIS filters and respond to "get all" operations.

Polymorphic get, set, action, delete and create methods are defined by the MO class and may be redefined by derived classes to add behaviour. The get and set methods operate on a per attribute basis and are called for every attribute in the m-get and m-set request. They inform the object instance to either "refresh" those attributes or to do something to the associated resource as a result of a set request. The actual get and set operations to the attributes are performed transparently by the MO class. The action method instructs the behavioural part of an object instance to perform the action to the associated resource. The delete and create methods trigger associated behaviour, if any. Finally, the "trigger notification" method is called by the behavioural part of an MO to emit a notification.

```
class MO
{
    // . . .

protected:
    virtual int get (int attrId, int classId, AVA*& errorInfo,
                     Bool checkOnly = False, int asyncInvokeId = -1);
    virtual int set (int attrId, int classId, ModifyOp mode,
                     Attr* setVal, Attr*& resVal, AVA*& errorInfo,
                     Bool checkOnly = False, int asyncInvokeId = -1);
    virtual int action (int attrId, int classId,
                     Attr* info, Attr*& reply, AVA*& errorInfo,
                     Bool checkOnly = False, int asyncInvokeId = -1);
    // . . .
public:
    virtual int ddelete (DeleteType dtype, AVA*& errInfo,
                         Bool checkOnly = False,
                         int asyncInvokeId = -1);
    virtual int create  (CreateType ctype, AVA*& errInfo,
                         int asyncInvokeId = -1);


    int        triggerNotification (int notifId, int classId,
                                    Attr* notificationInfo);
    // . . .
};
```

Code 3-12 The Polymorphic Managed Object Methods

The polymorphic MO methods are shown in the Code 3-12 caption. A dynamic, weakly-typed approach has been followed, in a similar fashion to the RMIB model. If the GMS was to be redesigned now, the author might have followed a static, strongly-typed approach, similar to the NMF GDMO/C++ [Chat97] and CORBA [CORBA]. On the other hand, a static approach is much more complex and results in the generation of more code. The reader is reminded here that the advantage of the strongly-typed approach is compile-time as opposed to run-time checking. An important aspect of the polymorphic API is that these methods can be asynchronous as well as synchronous. The asynchronous option is typically used in single-threaded environments when the relevant method involves remote access of a subordinate information model.

Finally, aspects pertinent to a managed object class are modelled by a meta-class object. The general meta-class is called *MOClassInfo* and keeps information about the class, name bindings, packages, attributes, actions, notifications and the respective syntaxes and OIDs. The MO part of an object instance is told about the actual class by its leaf-most derived class and keeps a handle to the relevant class object. This means it has access to all the meta-class information so that it validate and handle a particular management request. Figure 3-14 shows the layout of a uxObj instance and its access to meta-class object instances.

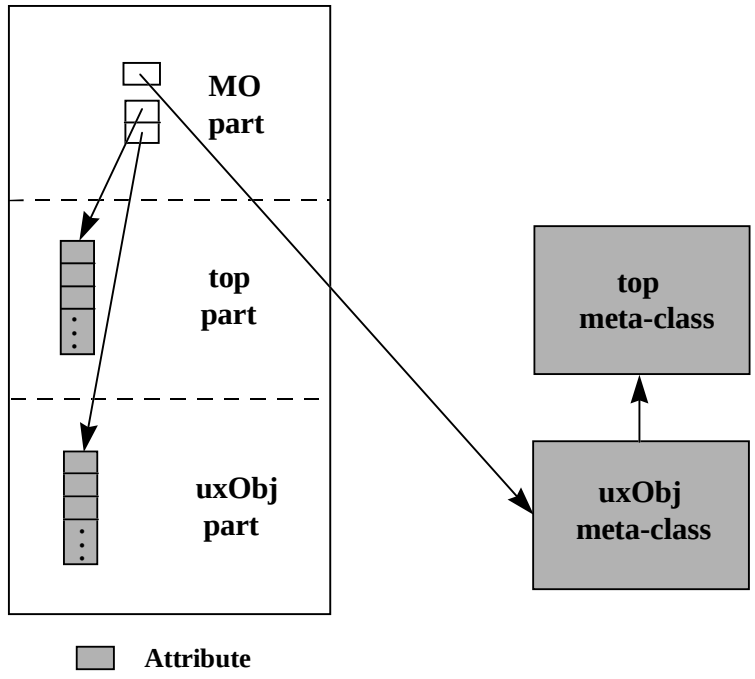


Figure 3-14 The Internal Layout of a uxObj Instance and Associated Meta-Class Objects

In summary, the rules for mapping GDMO to C++ presented above are the following:

- Managed object classes map to C++ classes. In the case of GDMO multiple inheritance, this is collapsed first to GDMO single inheritance and subsequently mapped to C++ single inheritance. The root of the C++ inheritance hierarchy is the MO class.
- Packages are implicitly present in managed object instances through their attributes, actions and notifications. The attributes of conditional packages are only instantiated if the package is present. An object instance keeps state information about the active conditional packages so that it can respond correctly to requests.
- Attributes map to C++ instances that model the relevant ASN.1 type. Every class keeps an array of its attributes and passes its handle to the MO part, which has access to all the attributes of that object instance. The get and set polymorphic MO methods are called to refresh an attribute value or to perform an operation to the associated resource.
- Actions map to the polymorphic action method which is called as a result of an action request. The action information and reply map to C++ instances that model the associated ASN.1 type.
- Notifications are supported by the triggerNotification method of the MO class which is called when a notification is emitted. The notification information maps to a C++ instance that models the associated ASN.1 type.
- Parameters to attributes and actions indicate MOC-specific errors and are mapped onto C++ AVA instances in the relevant polymorphic methods.
- Name bindings map to special C++ instances kept by the meta-class object. Every object knows its name and the active name binding through the nameBinding attribute of the top class.
- Finally, any other information pertinent to the class is mapped to instance variables of the meta-class object.

This approach has been validated by the implementation of the OSIMIS GMS which has been used to realise various applications in agent roles as described in Appendix A. Most of the abstractions presented above have been adopted by a number of commercial products. The presented APIs hide the underlying protocol aspects as much as possible and leave an implementor to concentrate in the behaviour of the managed objects, which is provided by the redefinition of the polymorphic methods.

As an example, the implementation of the uxObj class presented in Appendix B involves the following. The *get* method should be redefined to refresh the *sysTime* and *nUsers* attributes when these are requested by name or when all the attributes are requested; and the *action* method should be redefined to respond to the *echo* action. The set method does not need to be redefined for the *wiseSaying* attribute since there is no associated resource. The agent and the MO part perform transparently all the checking, decoding, encoding and behaviour triggering. The *objectCreation*, *objectDeletion* and *attributeValueChange* notifications are also emitted transparently, without any additional code by the implementor. The GDMO compiler produces the object skeleton in *uxObj.h* and *uxObj.cc* files while the behaviour is included through *uxObj.inc.h* and *uxObj.inc.cc* files supplied by the implementor. The uxObj implementation involves only a few lines of code.

3.6.5 The Relationship of Managed Objects and Associated Resources

A managed object always represents a real resource. This may be a fine-grain resource at the lowest level of the TMN hierarchy, i.e. in network elements, or a more abstract resource in higher TMN layers. Network element resources can be thought as tightly-coupled with the associated managed objects since they are co-located in the same network node e.g. an ATM switch. In this case, access time between objects and resources is typically small. Resources in higher TMN layers can be also tightly coupled with the associated managed object, e.g. a customer record in the service management layer. Most typically though, a resource in a higher TMN layer is loosely coupled with the managed object in the sense that it maps onto lower layer resources in a recursive manner. In this case, an operation to the associated managed object may result in operations to subordinate managed objects as described in Chapter 2.

An important issue is how the attributes of the managed object will present a consistent and up-to-date view of the resource in the case of m-get operations. There are three major approaches for maintaining the MO and associated resource consistency:

a) "*access upon request*" - the resource is accessed only as a result of the m-get request;

b) "*cache ahead periodically*" - the resource is accessed periodically and the relevant attributes are refreshed; and

c) "*update through events*" - in this case the resource sends changes to the managed object asynchronously and this updates its attributes.

The first approach has the advantage of reduced management traffic in the case of loosely coupled resources but at the cost of increased response time. In the case of tightly-coupled

resources within network elements, it is the most sensible approach. The second approach has the disadvantages of increased network traffic and potentially reduced information timeliness. Its only advantage is fast response time since the managed object is able to respond immediately to the m-get request. The third approach generates less traffic than the second one and retains the advantage of fast response. On the other hand, it incurs more management traffic than the first one. When using b) and c) above, the polymorphic get method does not need to be redefined since the attribute values will be up-to-date upon the reception of a get request.

There can be variations and combinations of those approaches. OSIMIS supports all three methods of operation: a) by redefining the polymorphic get method, b) through support for periodic polling (it will be described in section 3.7.2) and c) through events from subordinate systems.

3.6.6 Realisation of the "Difficult" Agent Aspects

In this section we look at those features of OSI-SM which are considered difficult to implement and explain how they are supported by the underlying agent infrastructure.

The MIT is a n-ary tree which can be represented internally as a binary tree. The manipulation of binary trees has been addressed in detail in the literature. The author used data structures and algorithms described in [Aho83]. An issue related to the MIT representation is CMIS scoping. This is almost trivial to provide: a recursive method is used which performs a pre-order search down to the required level and adds object handles to an array.

Another aspect that was initially considered difficult to implement is filtering. Attributes are mapped to C++ Attr instances which can evaluate filter assertions as explained in section 3.4. A CMIS filter is a tree data structure in which all the leaves are filter items (see Appendix D). A recursive algorithm can be used to scan the filter expression and evaluate the filter items linked by boolean operators. The attribute name in the filter item is first verified through the meta-class information and the relevant assertion is then evaluated. The filter method is supported by the MO class which has access to the meta-class information and the attributes of the object instance. Before filtering is applied, the object instance is requested to "refresh" the relevant attributes.

Resolution of distinguished names is also supported by the MO class. Since every MO knows its relative name and keeps handles to superior and subordinate objects in the MIT, a recursive method can perform a breadth-first search and compare relative names. This means that the associated computing cost is generally a linear function of the breadth and depth of the MIT for a particular object. Another approach which could result in a flat response time is a hash-table

approach. In this case the choice of the hash algorithm becomes crucial in order to result in a relatively even distribution and avoid collisions which cause inefficiencies.

One of the most difficult OSI-SM aspects is atomicity which may be requested through the CMIS synchronisation parameter. OSIMIS implements atomicity through an internal two-phase commit approach. Initially all the selected managed objects are asked by the agent if they can perform the requested operation. This is done by calling the relevant polymorphic method with the *checkOnly* flag set to *True* (see Code 3-12). If at least one object does not accept, the operation is rejected. Otherwise, the objects are requested to commit the operation. This approach provides a rudimentary atomicity facility. Distributed transaction processing [DTP] may be used in conjunction with CMIS to "bracket" a number of operations to different agents in an atomic transaction.

Another difficult aspect related to the realisation of the GDMO model is allomorphism. This is supported in the following fashion. The MO part initially checks the asserted class and makes sure that the attributes or specific action are supported by it (or by its parent classes). The behavioural code of a leaf most class subsequently checks the class asserted in the operation. If this is one of the parent classes, it calls the equivalent method of the parent class. In this way, the behaviour of the right class is always invoked even if an attribute or action has been redefined in a derived class. This implies that support for allomorphism should be explicitly provided by the behavioural parts of managed objects.

Finally, managed objects need to be persistent so that they can survive re-starts of the agent. Another reason for object persistence is that it may be impossible to hold all the managed objects in core memory in the case of agents with a very large amount of objects. Object persistence can be provided by storing the state of an object through its attribute values in secondary storage. The fact that attributes in OSIMIS have a well-defined string representation can be used to serialise a MO and "save" it on disk. A database with random access is necessary to "revive" a particular MO. The GNU version of the UNIX *dbm* database was used in OSIMIS for object persistency.

3.6.7 Systems Management Functions

The OSI Systems Management Functions (SMFs) [SMF] have been introduced in detail in Chapter 2. In this section we consider briefly their agent realisation aspects.

The most important SMFs are event reporting [X734] and logging [X735]. Their rationale and model is explained in [Laba91a]. When a managed object emits a notification, this is passed to the notification processing (NP) function. The latter could be modelled by a separate object as it

was depicted in Figure 3-13 but in the case of OSIMIS it is part of the MO class. A notification is triggered by the behavioural part of a managed object through the *triggerNotification* MO method. A "potential event report" or "potential log record" is subsequently created. This. is a specific log record object according to the notification type e.g. an *attributeValueChangeRecord*. This object is not part of the MIT but serves solely the purpose of evaluating the EFD and log filters. The NP runs subsequently through the EFDs and logs and evaluates their filter on the potential event report. If the filter evaluates to true, it instructs the relevant EFD to send the event report or instructs the relevant log to create a copy of the log record. The event report is sent through the agent. The EFD may need to retransmit confirmed event reports until a confirmation is received through the agent.

The power of the OSI-SM event reporting model stems mainly from the fact that EFDs and logs may contain sophisticated filters which can be applied to the potential event report objects. This allows assertions on the class and name of the emitting object, on the event type and time and on information specific to the event e.g. the attribute name, its previous value and its new value for an *attributeValueChange* event.

Other important SMFs are object [X730] and state [X731] management. The object lifecycle notifications, i.e. *objectCreation*, *objectDeletion*, are automatically supported when an object is created and deleted. In addition, the *attributeValueChange* and *stateChange* notifications are also supported automatically when the attribute changes as a result of a CMIS m-set request. The relationship management SMF [X732] specifies relationships through "pointer" attributes that contain the distinguished name of another object. These can be resolved to the pointed object handles through the find or resolve MO method.

The only other SMF which has an impact on the agent infrastructure is access control [X741]. The agent object will contain an Access Decision Function (ADF) object which will check access control rights and will grant or deny access for a management operation. The MIB agent is seeded with hooks which will invoke the ADF functionality and check access rights.

The rest of the SMFs consist simply of support managed objects which are implemented and linked with agent infrastructures so that they can be instantiated by manager applications. OSIMIS supports the SMFs mentioned above and also the metric monitoring objects [X739], the summarisation objects [X739] and a combination and extension of the previous two known as the intelligent monitoring objects [Pav96c].

3.6.8 Summary

In this major section we looked at issues related to the realisation of object-oriented agent infrastructures. The target was to provide an object-oriented decomposition of an agent infrastructure, propose a concrete mapping of GDMO to object-oriented programming languages and provide an environment for the realisation of managed objects which shields implementors from the underlying access service and protocol complexity.

We proposed an agent architecture known as the Generic Managed System which separates service and protocol processing from the managed objects through an MIB agent object. We also proposed a GDMO mapping to C++ which uses only single inheritance and can be provided in languages such as Smalltalk and Java. The managed object class specifications are compiled through a GDMO/ASN.1 compiler which produces stub MO classes. These can be augmented with behaviour by redefining the polymorphic methods of the generic MO class. The attributes, action and notification information are modelled though C++ instances according to the O-O ASN.1 principles presented in section 3.4. The resulting environment shields implementors from protocol details and enables them to concentrate in the object behaviour and associated application intelligence. The whole approach is very similar to CORBA but was designed, specified and implemented much before the CORBA IDL to C++ mapping.

We also discussed issues related to the maintenance of consistency between managed object attributes and associated resources. Three different schemes were identified which have different properties in terms of incurred management traffic and information timeliness. We finally discussed the “difficult” implementation aspects of OSI-SM and explained that they are in fact easy to provide through object-oriented design principles. These included the MIT representation, scoping, filtering, name resolution, atomicity, allomorpism and persistence. We also explained how event reporting, logging and the rest of the OSI SMFs can be supported.

In summary, this section demonstrated the feasibility and implementability of the agent aspects of the OSI-SM model. The author designed and implemented the majority of the agent infrastructure. He would like though to thank other members of the UCL team for their contributions and in particular: G. Knight for various discussions and direction and also for conceiving and implementing the uxObj example; J. Cowan for implementing the GDMO compiler in an ingenious fashion so that its back-end can be easily customised; S. Bhatti for designing and implementing the log control SMF and the managed object persistency; and K. McCarthy for trying to “break” the agent infrastructure in every conceivable fashion while implementing the generic CMIS/P to SNMP gateway.

3.7 Issues On Synchronous vs. Asynchronous Remote Execution Models

In the previous sections of this chapter we have demonstrated how to map the OSI-SM/TMN model onto O-O programming environments in the form of a distributed management platform. The realisation model for both the agent and manager infrastructure included both synchronous and asynchronous API facilities. In this section, we investigate the relevant issues behind the use of synchronous and asynchronous options. We also present the solution adopted in OSIMIS for co-ordinating the activity of a management application.

3.7.1 Remote Procedure Call and Message Passing Paradigms

The synchronous remote execution model was first presented in [Birr84] through the notion of Remote Procedure Calls (RPCs). A remote operation is modelled through a procedure in the calling entity, with semantics similar to those of local procedures, which takes as input and output arguments those of the remote operation. When this procedure is called, it triggers the actual remote operation to another entity, possibly across the network, and returns with the result or error. While a remote procedure call looks exactly like a local procedure call from a programmatic point of view, it has very different performance characteristics. A message needs to descend and ascend the local and remote protocol stack, including the relevant encoding and decoding overhead. In addition, the end-to-end network latency for the request and response adds to the overall delay. In general, the difference in performance between local and remote method calls is at least two orders of magnitude (10^2).

The asynchronous remote execution model predated the RPC model and is often referred to as “message passing”. A remote operation is modelled through two different operations in programmatic terms: a “send” procedure, which takes as arguments only the input parameters for the operation; and a “receive” procedure”, either in a callback [Clark85] or in a message queue fashion. In the case of the callback or “push” model, the “send” procedure also passes as an argument (a pointer to) the “receive” procedure, which is eventually invoked by the infrastructure with arguments the output parameters for the operation. In the case of the event queue or “pull” model, the “receive” procedure needs to be invoked by the caller. The receive procedure is typically generic, i.e. the same for all the different send procedures, and fills-in a data structure which is the “union” of all the possible output parameters for the remote operations.

The performance cost of calling the send or receive procedures is exactly that of descending or ascending the protocol stack. This means that the relevant cost is only a fraction of the total cost of a remote procedure call. Exact figures of this difference in the case of CMIS/P will be presented in the next section. According to the models described above, the MSAP API is an asynchronous one using the event queue model while the RMIB/SMIB and GMS APIs are both synchronous and asynchronous, using the callback model.

The main advantage of the synchronous execution model is that it is natural to the programmer: a remote method call appears exactly like a local method call, so the logical flow of a program that uses remote operations is not different to a program that invokes no remote operations at all. The RMIB and SMIB examples in the Code 3-6 and Code 3-7 captions used the relevant API in a synchronous fashion. In comparison, the asynchronous execution model appears less natural for the programmer: every time a remote method is invoked, the logical flow of the program is "interrupted" i.e. the method that contains the "send" method invocation typically keeps some state and returns. If, say, the callback model is used for the result, the logical flow of the program will continue again when the callback method is invoked with the result. In summary, the asynchronous execution model necessitates to keep some state information while it also splits the logical flow of control in two parts: one before the remote method is called and one after the callback method is called with the result, the two parts being in different program methods.

While from the above discussion it appears that a synchronous execution model is more natural, and thus desirable, it also has a relevant limitation which requires special support. Synchronous method calls block the performing application until they return (by application in this case we mean a single operating system process). Since the cost of remote method call is much higher than that of a local method call, the overall performance of the application will degrade since it will stay idle for relatively long periods of time, while there may something else that needs attention. For example, a TMN OS may be performing a number of sequential synchronous operations to subordinate OSs while a peer or superior OS has sent a request to it which is being "queued up".

In the above discussion we have assumed a single-threaded execution paradigm i.e. the application is a "heavyweight" operating system process that does not contain "lightweight" processes or threads internally. The answer thus to the above performance problem is to use multi-threading. The latter though introduces the need for *concurrency* control: different threads inside the same program may try to access the same data, so some form of locking is required. This in fact requires state information while it also introduces the possibility of deadlocks. In fact,

it is difficult to harness the power of multi-threading in complex programs such as a TMN OSs, where many things make take place in parallel, and make sure they will never deadlock.

An additional issue with multi-threading is the fact it is not yet supported by every operating system in a native fashion, i.e. through the operating system kernel. For example, it is supported in the Sun Solaris version of UNIX and the MS Windows NT but it is not yet supported in the increasingly popular Linux version of UNIX. In fact, "native" threads have only been available during the last 2-3 years. In the past there had been user-space thread packages which did not have proper support for system facilities. They made programs difficult to debug because debuggers were "unaware" of them. In addition, they added their own performance overhead. The author had some "interesting" experiences with the ANSA [ANSA89a] user-space threads package before he decided to switch into single-threaded execution mode to solve those problems. It should be noted though that nowadays multi-threading has become much easier to use with native operating system support and languages like Java [Sun96] which provide built-in support.

When OSIMIS was first designed (circa 1989-90), it was necessary to support asynchronous APIs for applications that were conscious about performance, in addition to synchronous ones. It is interesting to observe that the NMF TMN/C++ family of APIs, which represents a serious industrial approach towards TMN platform APIs, provides *both* synchronous and asynchronous APIs, in a similar fashion to OSIMIS. It is also interesting the fact that OMG has in its plans a specification an asynchronous API facility for CORBA version 3. The author has always expressed the necessity for asynchronous APIs, in addition to synchronous ones, and this seems to be in line with the current industry approach to distributed systems. Finally, [Crow93] points out problems of the synchronous remote execution paradigm when used without multi-threading.

OSIMIS was designed to operate initially in a single-threaded execution mode, leaving the introduction of multi-threading for a later stage. In fact, the latter never happened because the author got too busy with other things, but at least two commercial TMN platform vendors whose products are based on OSIMIS have introduced multi-threading. Given the fact that OSIMIS is single-threaded, asynchronous remote method execution is necessary for increased performance in complex applications. Asynchronicity requires state management, so it is necessary to support the application programmer through relevant platform facilities. As such, an object-oriented application *co-ordination* mechanism was designed and developed. This makes possible to organise a complex program that receives external input and generates output in a single-threaded fashion. The principles of this important mechanism are described in the rest of this section.

3.7.2 The OSIMIS Coordination Mechanism

The idea of a co-ordination mechanism with a central facility in each application that becomes the focal point of all external input was contributed by G. Knight of UCL in late 1989 i.e. in the very early days of OSIMIS. The author researched into the relevant issues and designed and developed this mechanism in an object-oriented fashion, using inheritance and polymorphism. A particular objective in this design was that it should be able to coexist with similar mechanisms of other systems. The only other system in mind at the time was the X Windows system, which was going to be used for TMN WS-OSs. The relevant design has been general enough though, so it has been possible to extend the mechanism later and add support for the Tk widget set [Oust94] and more recently for the Orbix CORBA platform.

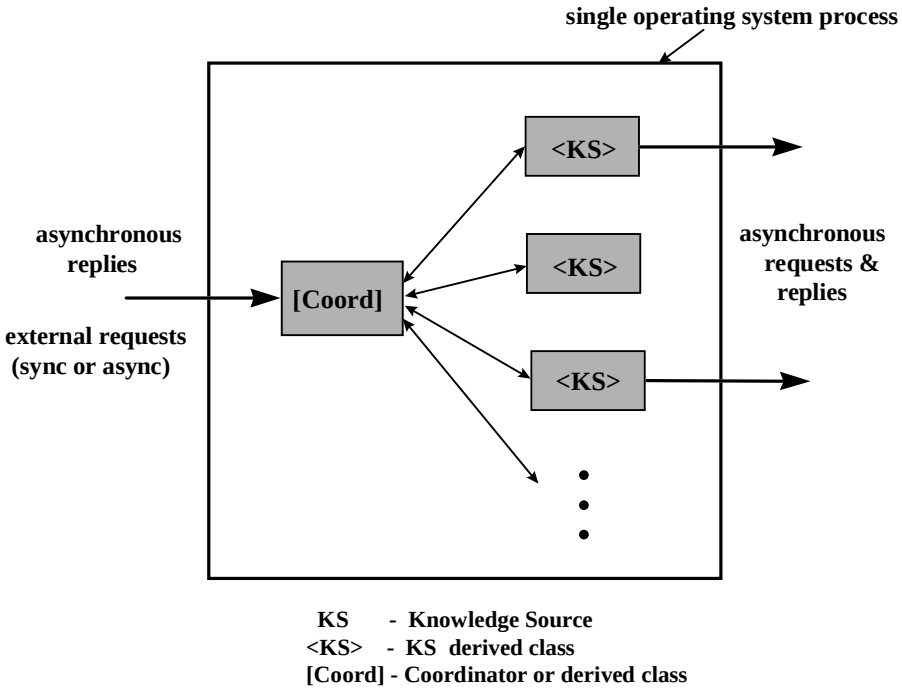


Figure 3-15 The OSIMIS Co-ordination Model

The OSIMIS co-ordination mechanism allows to organise a management application which is realised as a single operating system process in a fully event-driven fashion. All external input is serialised and taken to completion on a "first-come-first-served" basis. In every application, there exists a single instance of class *Coodinator* or of a derived class. This implements the central listening loop of the application, in fact the last line of the main program of any OSIMIS application is always "`coordinator.listen();`". Any other object in the system that expects external input should derive from the abstract class *KS* which stands for "Knowledge Source", its name having the roots in Artificial Intelligence (AI) Blackboard Systems. KSs register

"communication endpoints" with the Coordinator and are informed when there is relevant input. In addition, the Coordinator handles timer alarms and wakes-up periodically KSs that have requested this facility. This is because most operating systems do not "stack" timer alarms, so a central facility for taking care of them is necessary.

This model is depicted in Figure 3-15. The various KSs in the system should perform only asynchronous operations, with the replies passed back to them through the Coordinator. Every time there is external input or a timer alarm expires, the Coordinator informs the relevant KS who starts some activity. During the period of this activity, any other external input or timer event will have to be queued up. This means it is up to the objects involved in a particular sequential thread to relinquish control i.e. this is *not* pre-emptive round-robin scheduling. As a consequence, time-consuming activities should be ideally delegated to other processes so that the system remains responsive. For example, if as a result of a number of alarms, a TMN OS needs to evaluate a large rule base which is expected to take time, the rule-based system should be ideally implemented in another process that communicates with the OS through some Inter-Process Communication (IPC) mechanism, e.g. shared memory. Of course, nothing prevents any part of the application from performing time consuming activities, invoking remote operations in a synchronous fashion etc. This attitude though is not recommended since it will affect the application's performance and responsiveness.

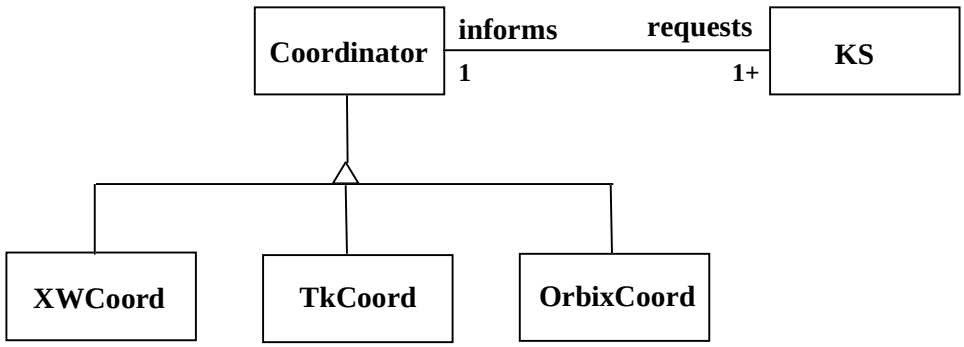


Figure 3-16 OMT Relationships of the Co-ordination Classes

In a TMN OS realised using OSIMIS, the only KSs that expect external input are the MIBAgent and the various RMIBAgents. There may be of course many other KS objects that require to be "awaken up" periodically. This mechanism has been tried and tested with complex TMN applications and has proved remarkably successful. In fact, this mechanism could also provide the basis for introducing multi-threading: the central listening loop will be realised in a central thread, while new threads will be spawned for every new external input or timer alarm. While this

is trivial to introduce, the key issue becomes then concurrency control and this involves a lot of work. One commercial product which is based on OSIMIS has introduced multithreading exactly in this fashion. In this case, pre-emption is taken care by the operating system. The advantage is that spawned activities need not be "conscious" anymore about their duration.

Figure 3-16 shows an OMT diagram of the relevant classes. There exists always one instance of Coordinator or of any derived class in the application while there exist one or more KSs. These request to be informed of external input or to be informed at regular time intervals and the Coordinator subsequently informs them. The Coordinator derived classes allow OSIMIS applications to coexist with other systems. For example, OSIMIS-based TMN WS-OSs may use X Windows or the Tk widget set. In addition, OSIMIS applications may contain CORBA objects, as it will be explained in Chapter 4.

```
class KS {
protected:
    // request wake-ups, register commEndpoints with Coord

    int scheduleWakeUps (long period, char* token = NULL);
    int cancelWakeUps   (char* token = NULL);

    int startListen (int commEndpoint);
    int stopListen  (int commEndpoint);
    // . . .

public:
    // callbacks for wake-ups, external events, process shutdown

    virtual int wakeUp (char* token);
    virtual int readCommEndpoint (int commEndpoint);
    virtual int shutdown ();
};
```

Code 3-13 The O-O Coordination API - the KS Class

The Code 3-13 caption above shows the object-oriented co-ordination API realised by the KS class. This provides methods to register external communication endpoints and to request periodic "wake-ups". Derived classes should implement the behaviour of the callback methods. Note that the Coordinator class is not visible to KS-derived classes but is hidden behind this O-O API.

The author designed and implemented the OSIMIS co-ordination mechanism, which is the "heart" of OSIMIS-based applications, early in the design of OSIMIS. It does not involve a lot of software in terms of lines of software but it took some time to get the object-oriented design right. In addition, the implementation of the Coordinator class and the relevant derived classes contain very complicated and relatively low-level software. Through object-orientation though this complexity is encapsulated and reused. The OSIMIS co-ordination mechanism was quite unusual at the time and still stands out for its object-oriented design and extensibility. It is now very similar though to the event model adopted for GUI implementation in many systems.

3.8 Performance Analysis and Evaluation

Object-oriented technology is generally thought to be expensive in terms of both processing and memory overhead, in comparison to non object-oriented systems. In addition, OSI technology is also considered expensive and non-performant. Since the proposed TMN framework is realised using object-oriented technology over OSI protocols, it is interesting to evaluate the relevant performance overheads and attribute them to the various parts of the system. This is necessary in order to validate the proposed framework from a performance point of view. While we have already demonstrated its flexibility, simplicity, user-friendliness and support for rapid application development in the previous sections, the significance of those would be undermined if the proposed environment is expensive in terms of required resources, resulting in poor performance for the relevant applications.

In this section, we measure the performance of the proposed framework in terms of program size, response times and amount of communicated information for a number of carefully chosen "benchmark" operations. In the relevant measurements, we try to separate the protocol overheads from those of the OSI-SM application framework. This is particularly important since the latter may be mapped onto other environments and protocols, e.g. over OMG CORBA, as proposed in Chapter 4. The overhead of the protocol stack and the application framework are also analysed further. We measure the overhead of every layer of the protocol stack while we also measure particular aspects of the proposed application framework.

3.8.1 The Environment and Methodology Used in the Experiments

The environment for the experiments was the following. The applications ran on two different UNIX workstations, connected to a lightly-loaded Ethernet local area network. The round trip delay was 1.5-2 msecs measured with the UNIX *ping* program. The applications operated the full Q_3 stack over TCP/IP, using the RFC 1006 method to emulate TP0 [Q811] - the Q3 protocol stack was described in detail in section 3.3.1. TCP/IP ran in the UNIX kernel while RFC1006 and the rest of the upper layer stack were linked with the applications and, as such, ran in user space. The OSI upper layer stack, including ACSE, ROSE and DASE, and the QUIPU Directory Service Agent (DSA) [QUIPU] were provided by ISODE-8.0 [ISODE]. CMISE and the rest of the TMN application framework were provided by OSIMIS-4.0 [Pav95b], which is the environment resulted from the research work described in this thesis.

Two different pairs of UNIX workstations were used in the experiments:

a) a Sun SPARC 5 and a Sun SPARC 20, running Sun's Solaris 2.5 version of UNIX; in this environment, the Sun C++ and C compilers version 4.1 were used; and

b) a Dell Latitude XP 486-75 laptop PC and a Viglen Genie 486-100 desktop PC, running the Linux 1.12 version of UNIX; in this environment, the GNU C++ and C compilers version 2.6.3 were used.

The first pair represents typical UNIX workstations used in work environments while the second pair represents fairly old technology PC's. The reason for the second pair is that they are in fact the author's home desktop and laptop PC's. The second pair offered the possibility to conduct tests in a fairly isolated environment i.e. a home Ethernet LAN. All the experiments were conducted in such a way as to make sure that the programs were kept in core memory, so that no paging took place which could affect the relevant performance figures.

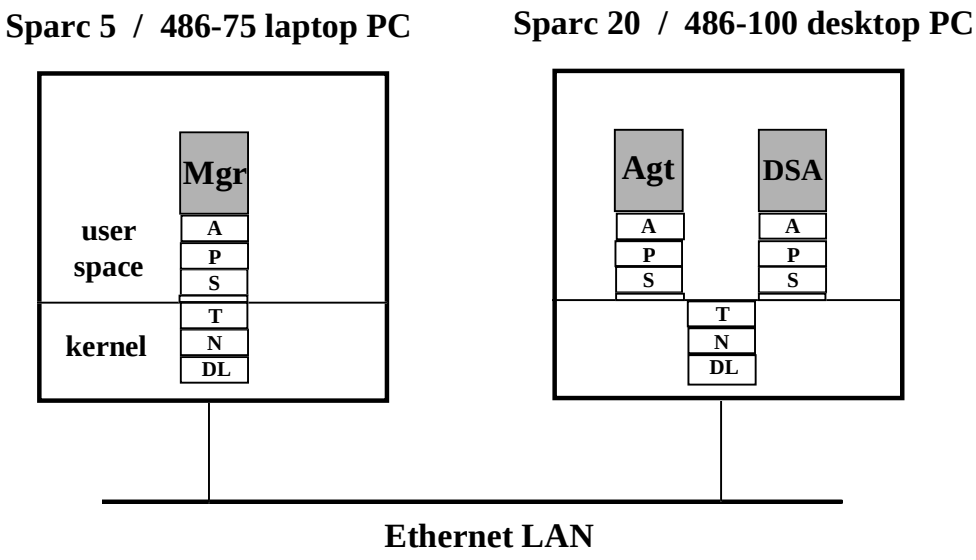


Figure 3-17 The Experiment Environment

All the experiments involved interactions between simple manager and agent applications. Since the computers in each of the two pairs were not the same, the agent application was run typically in the more powerful computer (i.e. the Sparc 20 and the 486-100 desktop). The experiments were also performed in the opposite direction i.e. with the agent running in the less powerful computer, without experiencing any significant differences in the pattern of the results. When a third application was involved, i.e. the DSA for location transparency, this was run at the same computer with the agent, so that the manager had to access it across the network. The environment for the experiments is depicted in Figure 3-17, showing the kernel / user space

separation inside every workstation and the protocol stack supporting the application. Note that the transport layer runs partly in the kernel (the TCP part) and partly in user space (the TP0 emulation through RFC1006).

The methodology for the measurements was the following. Program size was measured using the UNIX *top* utility program, with measurements taken on both the Solaris and Linux systems. The resulting differences were only minor while the program sizes reported are those experienced on Solaris. Packet sizes were measured using the popular *tcpdump* program, developed at the Lawrence Berkeley Laboratory. The sizes reported reflect the TCP payload as introduced by higher layers i.e. TCP, IP and Ethernet headers are not included. This is desirable since the same upper layer stack can operate over different lower layers, as it was described in section 3.3.1. Response times were measured using the UNIX *gettimeofday* system call and comparing measurements taken on the same computer in order to avoid the need for synchronised clocks. Timestamps were taken at different points of a sequential execution thread within a program and were kept in core memory, to be printed out only after the last significant timestamp had been taken. This was in order to reduce the overhead and discrepancy introduced by the measurement mechanism itself. It is not though possible to minimise totally its impact, in fact the author observed behaviour similar to the principle of Heisenberg in atomic physics: when trying to observe minuscule things, the measurements themselves had a significant impact in the observed results. Every experiment was conducted a number of times. Unusual results were discarded while the mean was derived from the rest. The results were fairly uniform and the standard deviation small.

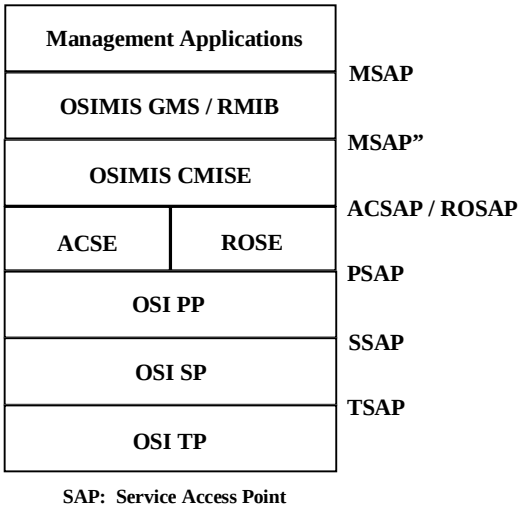


Figure 3-18 Service Access Points for the Performance Measurements

Special client and server applications were developed to echo data at every level of the protocol stack, including the TSAP, SSAP, PSAP, ACSAP/ROSAP and MSAP access points as shown in Figure 3-18. Two different notions of MSAP were used. The first was a "contrived" one, directly over the CMISE service without any agent and manager infrastructure, aiming to measure only the overhead of the OSIMIS CMISE layer over ACSE and ROSE; we will term this MSAP". The second is a "proper" MSAP, with the agent and managed objects in place through the GMS and the manager sending the request and receiving the response through the RMIB high-level API.

Response times in the TSAP and SSAP access points were measured using an "echo" operation with a byte string. Response times in PSAP were measured using an "echo" operation for an instance of the ASN.1 GraphicString type. Response times in ROSAP were measured by defining an *echo* remote operation with a GraphicString argument and result type. Finally, response times in MSAP" and MSAP were measured using the *echo* action of the uxObj class specified in Appendix C. The relevant CMIS request was the following:

```
m-action(objClass=uxObj, objName={uxObjId=null},
         actionType=echo, actionInfo=hello),
```

3.8.2 Program Size

We start first by examining the size of applications. Table 3-6 shows the size of client and server programs at the various SAPs identified above while Figure 3-19 depicts the same data in a graph form. A MSAP agent application with the object management [X730], event reporting [X734] and logging [X735] SMF capabilities and one instance of the system [X721] and uxObj classes amounts to 1340 Kb at run time. A MSAP manager implementing the echo action mentioned above amounts to 1060 Kb at runtime. Two similar but trivial MSAP" programs that contain no agent, manager or O-O ASN.1 infrastructure amount to 800 and 780 Kb or 59.7% and 73.5% of the overall size respectively as shown in Table 3-6. The equivalent ACSAP / ROSAP programs are 43.2% and 50.9% of the overall size respectively. This means that the overhead of CMISE together with the O-O TMN application framework amounts to 56.8% of the agent size and to 40.1% of the manager size. This can be explained as follows: CMISE is a fairly complex protocol compared to the rest of the underlying protocol stack. In fact, it is four times bigger than that of ACSE / ROSE according to those figures. The rest of the TMN application framework is pretty complex as well and it is implemented in C++ instead of C, which increases the relevant overhead. In summary, the overhead of the overall management infrastructure is roughly as much as that of ACSE/ROSE together with the rest of the underlying OSI protocol stack.

In TMN environments, OSs are hybrid manager-agent applications. If we link together the agent and manager described above so that echo actions can be performed in a peer-to-peer fashion, the run-time size becomes 1400 Kb. This is slightly bigger than the size of the plain agent, the difference (60 Kb) being the overhead of the RMIB infrastructure. This modest increase in size is easily explained since the two applications contain already a lot of common infrastructure i.e. the O-O ASN.1 support, the DMI syntaxes [X721] and the O-O co-ordination mechanism. In summary:

- the smallest OSIMIS-based TMN OS with object management, event reporting, logging and one "useful" managed object class with one relevant object instance amounts to 1400 Kb at run time.

Service Access Point	**Server Size (Kb - *%*)**	**Client Size (Kb - *%*)**
TSAP	390 - *29.1*	350 - *33*
SSAP	470 - *35*	420 - *39.6*
PSAP	530 - *39.5*	480 - *45.2*
ACSAP / ROSAP	580 - *43.2*	540 - *50.9*
MSAP"	800 - *59.7*	780 - *73.5*
MSAP	1340 - *100*	1060 - *100*

Table 3-6 Application Sizes at the Various Service Access Points

Another interesting aspect of the data in Table 3-6 is that the overhead of the RFC 1006 transport protocol appears to be around 30% of the overall client or server size. It is not the actual RFC 1006 code that causes this overhead, in fact this should be smaller than the session and presentation protocols which are around 80 Kb and 60 Kb respectively. This "initial fat" amounts to the overhead of the ISODE environment and, in particular, is due to the fact that the transport code has been written in a way to operate over a number of underlying protocols and environments. Even so, this overhead is pretty high (around 300 Kb) and could probably be reduced in a more carefully engineered protocol stack.

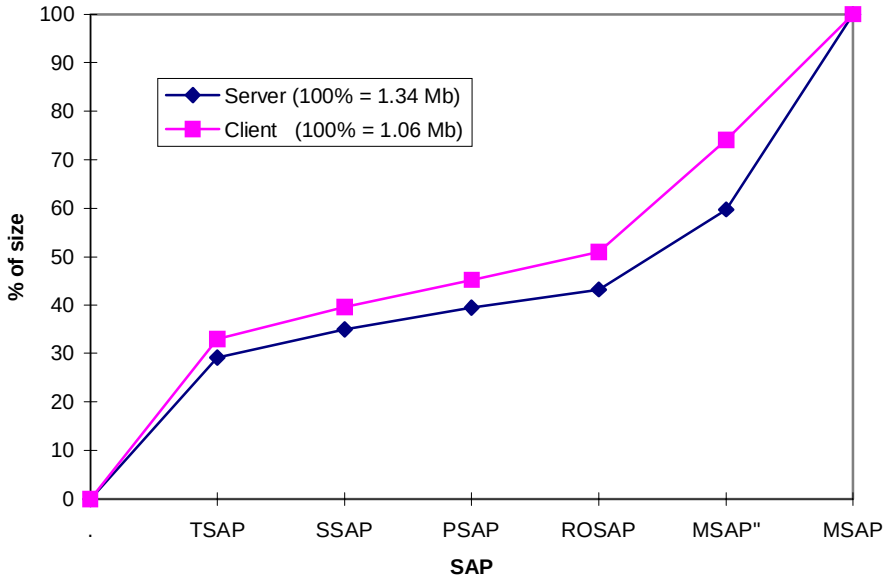


Figure 3-19 Application Sizes at the Various Service Access Points

We will finalise the discussion on the protocol stack overhead by examining the overhead of the CMOT instead of the full Q_3 stack. The CMOT protocol stack [Besa89] uses a Lightweight Presentation Protocol (LPP) [Rose88] that operates directly on top of the TCP. This is supposed to save mainly the session protocol overhead, given the fact that no sophisticated session layer services are needed for request-response protocols which are based on ROSE. Having rebuilt the ACSAP / ROSAP and MSAP programs over the LPP, it resulted in a reduction of about 150 Kb at runtime. This is only 10.7% of the size of the smallest TMN OS and it would be even smaller for OSs with real intelligence. As such, the CMOT memory savings are relatively insignificant.

Having examined the run-time memory overhead of the "base" management infrastructure, we will examine now the overhead of adding new classes, the overhead of object instances and the overhead of keeping management associations open. The overhead of linking in the uxObj class with the relevant behaviour amounts to 18 Kb at run time. The same overhead for the simpleStats class is 12 Kb. These two are example classes that were used for the purpose of demonstrating OSIMIS features. The performance monitoring classes *scanner*, *monitorMetric* and *movingAverageMeanMonitor* [X739] provide useful functionality for performance management and contain fairly complex behaviour. Linking those in amounts to another 76 Kb at run-time i.e. roughly 25 Kb for each of them. In summary, the minimal overhead of a new class with no behaviour seems to be about 12 Kb while the overhead for classes with significant behaviour can vary.

In order to evaluate the overhead of object instances, we implemented a simple manager that requests the creation of N instances of a particular object class. We used the uxObj class of Appendix C, with the values for the attributes depicted in the Code 3-14 caption. The data overhead of this particular instance is 420 bytes at runtime.

```
objectClass       2.37.1.1.3.50  (uxObj)
nameBinding       2.37.1.1.6.50  (uxObj-system)
uxObjId           <n>            (number in the 1-n range)
sysTime           19971223xxxxx  (time in UTC string form)
wiseSaying        "hello world"
nUsers            <m>            (small number e.g. 5)
```

Code 3-14 The uxObj Instance Used for the Memory Measurements

An agent containing 1 uxObj instance amounts to 1.34 Mb, 10^3 instances increase its size to 1.76 Mb, 10^4 instances to 5.54 Mb, $5*10^4$ instances to 22.34 Mb and 10^5 instances to 43.34 Mb. The agent with 10^5 or 100000 instances was running on a Sparc 5 with 64 Mb of RAM, which ensured there was no performance degradation when accessing those objects since the whole process image was kept in core memory. An agent with half a million or even one million MOs is possible through a computer with 256 and 512 Mb of RAM respectively, assuming all the MOs need to be kept in core memory. In general though, "inactive" MOs can be put on secondary storage through the persistency service, reducing the requirements for the required core memory.

Another important aspect worth measuring is the memory overhead of open associations. One of the arguments behind SNMP is its connectionless nature, which allows centralised management of thousands of network nodes. While this is not exactly a requirement in the TMN because of its distributed hierarchical nature, it is worth investigating if connection-oriented management scales. Experiments involved establishing multiple associations between a simple manager and agent. The memory overhead of an existing association was found to be 9.2 Kb for the initiator and 7.2 Kb for the responder. The asymmetry of the observed overhead has to do with the way the ISODE infrastructure handles associations since it was also observed by repeating the experiment at the ACSAP point. Assuming that the manager initiates those associations, the overhead is 920 Kb or roughly 1 Mb for every 100 associations. This does not pose a difficult problem, since a manager managing 1000 devices would be 10 Mb bigger at runtime. A particular issue though is that UNIX systems allow a only a maximum number of file descriptors open per process at a time, typically 64 or 256. Dealing with more descriptors necessitates to re-build the UNIX kernel.

Description	Size (Kb)
Minimal TMN application size	1400
Overhead of a "typical" MO class with no behaviour	10-15
Overhead of the uxObj class	18
Overhead of the uxObj instance as in Code 3-14	0.42
Overhead of an association (initiator)	9.2
Small DSA size	1500

Table 3-7 Summary of Q_3 Memory Overheads

Table 3-7 shows a summary of the application size overheads. In practice, fairly complex TMN applications like the ones implemented in the ICM project [Gri95][Gri96a] had a typical size between 3 and 5 Mb at runtime. As another benchmark, an agent implementing the OSI version of the Internet SNMP MIB-II [Laba91b] in a non-proxy fashion amounts to 3.2 Mb at runtime. The size of the equivalent SNMP agent is 1 Mb i.e. one third of the OSI-SM agent size. While it still does not make sense to manage devices such as video-recorders, set-top boxes, modems and repeaters with full Q_3 agents, the application sizes described above pose hardly a problem for typical telecommunication devices e.g. switches, multiplexors, exchanges, intelligent peripherals, etc. In fact, the size of a TMN OS with useful intelligence can be actually smaller than the size of a word-processing program on a desktop or laptop PC.

We will finally examine the application size of the QUIPU Directory Service Agent (DSA), which is used for dynamic address resolution and location transparency. The typical size of a DSA which serves the needs of the management environment, i.e. *not* a general purpose DSA with tens of thousands of directory objects, amounts to around 1.5 Mb at run time. This is reasonable and is in fact very similar to the minimal size of a TMN OS.

3.8.3 Response Times

In this section we examine the response times of management operations. Despite the fact that the TMN does not operate with the same stringent real-time requirements as the control plane, it still needs to react relatively fast to evolving network conditions. As such, it is important that the supporting infrastructure exhibits good performance characteristics.

SAP	**Sparc 5 - Sparc 20 (msecs - %)**		**486/75 - 486/100 (msecs - %)**	
	Connection Establishment	**Connection Release**	**Connection Establishment**	**Connection Release**
TSAP	10 - *25.6*	0.9 - *13.8*	19.1 - *18.7*	2 - *16.7*
SSAP	15.5 - *39.7*	3 - *46.2*	33.1 - *32.4*	6.25 - *52.1*
PSAP	19 - *48.7*	3.5 - *53.8*	51 - *50.0*	7.4 - *61.5*
ACSAP	22 - *56.4*	5 - 76.9	64.2 - *62.9*	9.5 - *79.2*
MSAP”	37.7 - *96.7*	6.25 - *96.1*	100 - *98.0*	11.6 - *96.7*
MSAP	39 - *100*	6.5 - *100*	102 - *100*	12 - *100*

Table 3-8 Response Times for Association Establishment and Release

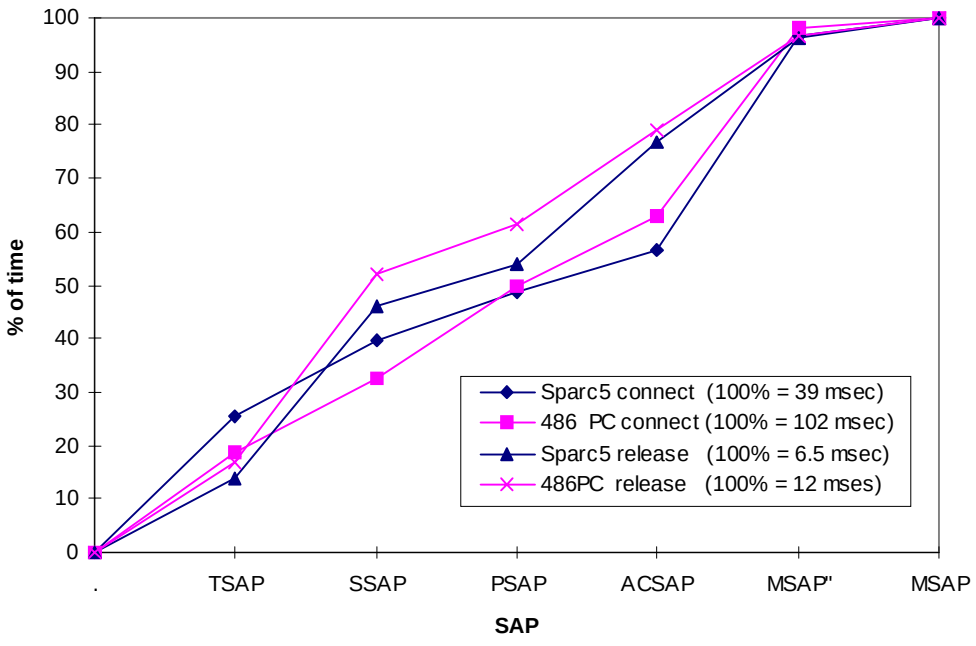


Figure 3-20 Response Times for Association Establishment and Release

We will examine first the performance aspects of establishing and releasing management associations. Table 3-8 shows the response times for association establishment and release at the various access points defined before. Both the absolute and normalised response times for the two pairs of computers and relevant environments used for the measurements are shown. Figure 3-20 depicts the same data in a graph form.

In general, establishing an association is expensive because of the various negotiations that need to take place in the session, presentation and application layer. It should be noted that while a transport connection requires only one request-response exchange, session / presentation connections and application associations require two request-response exchanges. Connection establishment times increase almost linearly until the ACSAP point. Establishing an ACSE association takes around 60% of the time of establishing a full Q_3 association with the information specified by CMIP [X711]. The overhead imposed by CMISE is almost 35% of the overall association time and this can be attributed mostly to the overhead of encoding and decoding the relevant negotiation information. The overhead of the proposed TMN application framework, which is reflected by the difference between the MSAP and MSAP" access points, is minimal i.e. around 3-4% of the overall association establishment time.

Association release is much faster in absolute figures i.e. around 15% of the association establishment time. In general, it is much easier to tear something down than it is to build it in the first place! It should be also noted that transport connections are torn down with a single packet exchange while session / presentation connections and applications associations require three packets: request, response and final confirmation packet sent by the initiator. Apart from the fact that closing down the association is faster, Figure 3-20 shows also that the overhead is more or less equally shared at each layer. This is logical since CMIP does not require to pass any information when closing down an association. Finally, the overhead of the proposed TMN application framework is again very small i.e. again between 3-4% of the overall association release time.

Association establishment and release to the DSA for location transparency purposes exhibits very similar response times to those presented above. This makes sense since the protocol stack is exactly the same until ACSE while the Directory Access Service (DAS) [X511] specifies initial association information similar to the one passed across for CMISE associations.

We will examine now the response times for management operations. While the performance of association control is important, it is reminded here that many operations are typically "multiplexed" onto the same association, with associations "cached" and released after a period

of inactivity. Given this mode of operation, response time for management operations are more critical. We will use the echo operation described in section 3.8.1 as a benchmark management operation. Table 3-9 shows the response times in the various access points echoing the "hello" string. Figure 3-21 shows the same information in a graph form.

SAP	**Sparc 5 - Sparc 20 (msecs - *%*)**	**486/75 - 486/100 (msecs - *%*)**
TSAP	2.2 - *18.3*	4.7 - *17.4*
SSAP	3.2 - *26.7*	6.2 - *23.0*
PSAP	4.2 - *35.0*	9.3 - *34.4*
ROSAP	5.5 - *45.8*	13 - *48.1*
MSAP"	10.7 - *89.2*	25 - *92.5*
MSAP	12 - *100*	27 - *100*

Table 3-9 Response Times for an Echo Operation

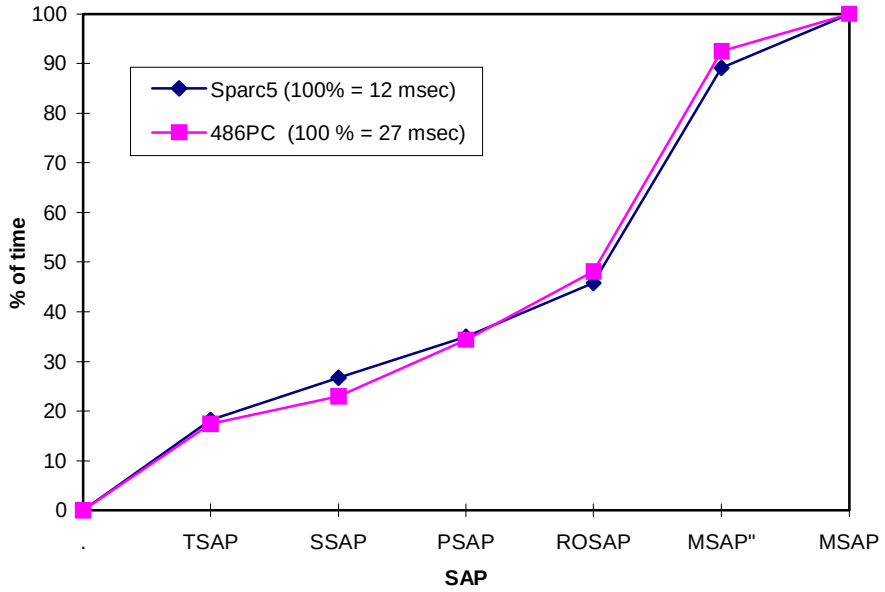


Figure 3-21 Response Times for an Echo Operation

The pattern experienced is similar to that for association establishment. The overhead of the echo operation at the ROSAP point is roughly half of that at the MSAP point. The overhead of the CMISE layer, as reflected by the difference between the MSAP" and ROSAP access points, is around 45% of the overall overhead. This can be explained by the fact that CMIP PDUs are

fairly complex ASN.1 structures and a significant overhead is involved in encoding and decoding them.

In general, ASN.1 encoding and decoding in ISODE is dealt with is in two stages, involving substantial memory manipulations in both. While this approach is general and flexible, as discussed previously in this chapter, it seems to incur a substantial performance overhead. [Huit92] claims that ASN.1 performance can be in fact higher than other similar mechanisms such as Sun's eXternal Data Representation (XDR) [Sun87], but their approach is a *one pass* encoding / decoding together and the lightweight encoding rules are used instead of the BER [X209].

The overhead of the proposed TMN application framework is reflected by the difference between the MSAP and MSAP access points, i.e. the overhead of the GMS generic agent and RMIB generic manager infrastructure, and amounts roughly to 10% of the overall overhead. This is a particularly important result, since the same application framework can be mapped over other mechanisms. In Chapter 4, we will discuss such a mapping on OMG CORBA.

While not shown in Table 3-9, using the CMOT instead of the full Q_3 stack results in MSAP response times that are around 15% smaller. In general, both the response time and memory advantages of CMOT are not significant, while Q_3 compliance and interoperability is lost. In summary, it is really not beneficial to use the CMOT stack.

The above experiments at the MSAP" and MSAP access points were conducted with the manager application operating in an asynchronous fashion. This means the request was sent through the asynchronous RMIB API and execution control was passed to the coordination mechanism. The latter passed control back to the RMIB infrastructure when the confirmation packet arrived, which in turn passed the result to the performing managing object.

An interesting finding is that exactly the same request in a synchronous fashion resulted in 7-8% faster response times. While this is initially surprising, it reveals that the cost of the coodination mechanism which was described in section 3.7.2 is not negligible. In addition, there is more context switching. Despite the fact that operations are slightly more expensive when performed in an asynchronous fashion, the performing application may be doing something useful while waiting for the confirmation packet. Of course, this can be also the case with synchronous operations performed in a multi-threaded execution environment.

We will now examine where exactly the overall end-to-end overhead is attributed to. We seeded both the manager and agent applications with instrumentation that took timestamps at various

stages of the operation in progress. Having analysed those, the results are very interesting. The procedure of sending the request down the protocol stack in the manager amounts to around 25% (3 msecs) of the overall response time. When the confirmation arrives, it also takes around 25% of the overall response time for the message to ascend the protocol stack.. The same figures for the indication and response messages at the agent amount to 15% of the overall time (1.8 msecs). This means that 25+25+15+15=80% of the overall time (9.6 msecs) is spent for the CMIS messages to descend and ascend the protocol stack. If we add to that the overhead of the asynchronicity and coordination mechanism (1 msec) and the network latency (0.5 msec), we are close to 90% of the overall time (11 msecs) which is the overhead at the MSAP" point. The rest 10% of the overhead (1 msec) is due to the RMIB and GMS infrastructure. The absolute figures mentioned above are for the Sparc 5 - Sparc 20 pair of computers.

What is particularly interesting with these figures is the asymmetry of the overhead of sending and receiving messages in both the agent and manager. Since the difference is not negligible, i.e. 25% vs. 15% of the overall time respectively (or 3 vs. 1.8 msecs), a number of additional measurements were conducted which verified this behaviour. The author cannot find a suitable explanation since the CMIP action request and reply packets are not that dissimilar. Detailed profiling is necessary to see what exactly causes this discrepancy but this was outside the scope of this study.

We will examine now the performance impact of a number of other parameters. We will consider first the size of the data in the management operation. The response times for the echo operation with a string of length 5 ("hello") was shown in Table 3-9. Increasing gradually the size of the echoed string results in an almost linear increase of the response time. The additional overhead is 0.21 msecs or 0.0175% of the overall response time per 100 bytes. The response time increase for amounts of data seems reasonable.

We will consider now the response times using CMIS scoping. Performing an operation on more than one objects through scoping results in a 25% increase of the response time per additional object. This means that 5 scoped operations take the same time as 2 non-scoped ones and this 5:2 ratio holds for other sizes. The 25% overhead per additional object is smaller than the expected overhead according to the above figures for descending and ascending the protocol stack: it should be at least 25%+15%=40% for every additional response. The fact that additional results are sent and received "back-to-back" has positive effects in the overall latency, resulting in smaller times for processing those packets. This behaviour was verified by additional measurements.

We will now investigate the overhead of name resolution, scoping and filtering inside the agent. Name resolution is implemented through a MIT recursive descent and comparison of relative names. As such, it should be a function of the breadth and depth of the MIT, i.e. *f(b,d)*. In order to measure this, we created 5000 uxObj instances and performed various experiments. The additional overhead for searching through 100 MOs until the specified one is found is 1.5%. This means a 15% overhead for searching through 1000 MOs. As discussed in section 3.6, a hash table approach would result in a more or less "flat" delay for locating the object anywhere in the MIT. Even with the current approach though, the overhead is fairly small i.e. an additional 1.5% for searching through 100 objects.

The cost of performing the scope operation within the agent is negligible. This operation simply assembles the right MO pointers by traversing the MIT, starting at the base MO. This is not surprising since assembling pointers through a recursive tree descent algorithm is something that modern processors do very quickly. The same is true for evaluating a filter on a managed object. We tried a number of filters on the uxObj MO and the evaluation time was negligible compared to the end-to-end response time. These filters were evaluated without "refreshing" the attribute values first, so that only the overhead of filtering was measured.

Finally, the cost of accessing the directory for location transparency reasons is very similar to the cost of performing a read or action operation on a managed object. The cost of establishing an association to the DSA, retrieving the presentation address of a TMN application, establishing an association to the latter and performing a management operation takes in total 40 + 10 + 40 + 10 = 100 msecs using the Sparc 5 / Sparc 20 setup. It should be noted though that this sequence of operations takes place only the first time. The presentation address of the target application is "cached" in the performing application while the same is true for the association to that application. The second operation will take in this case around 10 msecs.

3.8.4 Packet Sizes

We finally consider the sizes of management packets. These include the payload measured over the TCP, as explained in section 3.8.1.

Table 3-10 and Table 3-11 show the sizes of the packets exchanged for establishing and releasing connections at the various service access points. As already mentioned, from the session layer and above connection establishment requires 4 packet exchanges (2+2) while connection release requires 3 packet exchanges (2+1). It should be also mentioned that none of those packets are "piggy-backed" on the TCP packets for connection establishment and release, so we should count

another two packet exchanges with no data. In summary, upper layer connection establishment requires 6 while connection release requires 5 overall packet exchanges.

SAP	Data sent (bytes / packet)	Data Received (bytes / packet)	Total (bytes)
TSAP	21	17	38
SSAP	18 29	14 31	92
PSAP	18 40	14 42	114
ACSAP	18 132	14 130	294
MSAP	18 140	14 138	310

Table 3-10 Packet Sizes for Connection Establishment

SAP	Data sent (bytes / packet)	Data Received (bytes / packet)	Total (bytes)
TSAP	11		11
SSAP	12 11	9	32
PSAP	12 11	9	32
ACSAP	28 11	25	64
MSAP	28 11	25	64

Table 3-11 Packet Sizes for Connection Release

A graph depiction of the total data in Table 3-10 and Table 3-11 is shown in Figure 3-22. ACSE adds another 180 bytes on top of the presentation layer connection setup, which requires the exchange of 114 bytes. CMISE adds another 16 bytes of initial information over ACSE. A substantial amount of traffic is required to setup a management association: 6 packets and 310

bytes in total. Connection release requires the exchange of 5 packets but the amount of overall data exchanged is much smaller, 64 bytes in total. In summary, management association establishment and release require a significant number of packet exchanges. In addition, association establishment incurs also a significant traffic overhead.

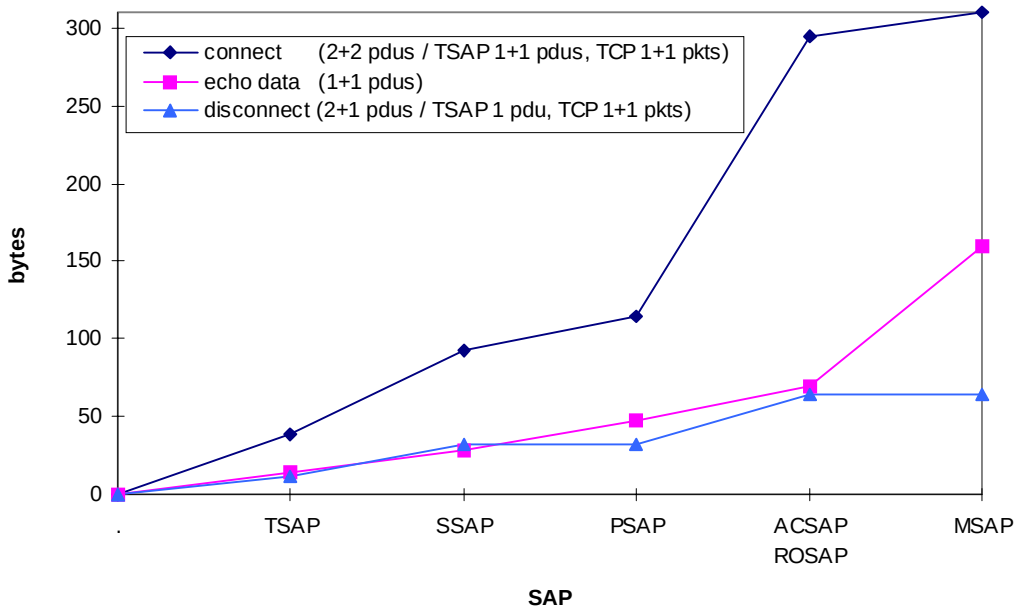


Figure 3-22 Connection Establishment, Release and Echo Operation Packet Sizes

SAP	Data sent (bytes)	Data Received (bytes)	Total (bytes)
TSAP	7	7	14
SSAP	14	14	28
PSAP	24	24	48
ROSAP	34	36	70
MSAP	72	88	160

Table 3-12 Packet Sizes for the Echo Operation (zero length string)

Table 3-12 shows the packet sizes at the various SAPs for the echo operation, echoing an empty string. Figure 3-22 shows also the overall data exchanged in graph form, together with the same data for connection establishment and release. The echo operation requires a request and a response packet in all the SAPs. In this case, the CMISE layer (difference between the MSAP and ROSAP access points) adds a substantial amount of data because of the parameters in the relevant PDUs. It should be noted that both the action request and response PDUs include the

name of the addressed object which is “uxObjId=null”. The overhead of that particular distinguished name component is exactly 16 bytes, including 4 bytes for the “null” string. In general, the request and response packet sizes will be larger for objects located “deeper” in the MIT.

Retrieving the wiseSaying attribute with value “hello world” of the uxObj instance incurs a request packet of 67 bytes and a response packet of 100 bytes. Retrieving all the attributes of the uxObj instance as depicted in the Code 3-14 caption but with the relative name “uxObjId=null” results in a request packet of 62 bytes and a response packet of 142 bytes.

3.8.5 Summary

In this section we looked at the performance aspects of the proposed object-oriented development environment in terms of application size, response times and packet sizes.

In subsection 3.8.2 we looked at TMN application size. A summary of the relevant overheads were presented in the Table 3-6. The minimal TMN application is about 1.4 Mb at run-time while example applications with fairly complex behaviour and a number of object were 3-5 Mb at run-time. The impact of additional object instances depends on the number and size of relevant attributes and other private instance variables. A typical overhead of an instance is smaller that 1 Kb, which means that applications with tens of thousands of managed objects are feasible. While the minimal application size, which effectively represents the overhead of the infrastructure, is not small, it is hardly a challenge for today’s computers and typical memory sizes. In fact, such applications run happily on the author’s laptop computer with 8 Mb of RAM and an Intel 486 processor.

In subsection 3.8.3 we looked at response times and presented results for two types of fairly modest computers. Association establishment is around four times more expensive than performing a simple management operation and this highlights the fact that associations should be typically “cached” and released only after a period of inactivity. Absolute times for association establishment are around 40 msecs for the relatively powerful computers and around 100 msecs for fairly old technology PCs. Absolute times for simple management operations are around 10 and 40 msecs respectively. While these absolute numbers are only meaningful when compared with similar operations in other frameworks, they confirm the subjective feeling of fast operation for the overall framework, which is also acknowledged by other researchers. For example, [Deri97] compares the performance of a number of management environments, including

OSIMIS, and characterises it as "performant" while it characterises typical commercial TMN platforms as "partially performant, requiring powerful hosts and a lot of memory".

What is more important is that the detailed measurements at various service access points and within management applications reveal that the main performance overhead is due to the protocol stack rather than the proposed TMN application framework. This is particularly important since the protocol stack can be replaced with other, more lightweight stacks. Alternative mappings for CMIP were discussed in section 3.3.2.4 while in the next chapter we will present a totally different mapping over OMG CORBA and will evaluate its performance.

Finally, in subsection 3.8.4 we looked at packet sizes. Association establishment seems to incur a significant amount of traffic, involving 6 overall packet exchanges. On the other hand, the packet sizes of management operations seem relatively reasonable, considering in particular the fact that OSI protocols and the BER are used. In subjective terms, the author was expecting much higher figures before conducting the experiments. It should be finally mentioned that packet sizes are pertinent to the *specification* of the OSI-SM/TMN framework while application sizes and response times are pertinent both to the specification and the proposed software *realisation* framework, including the ISODE OSI upper layer environment.

It should be finally mentioned that the performance evaluation in this section was not exhaustive. A detailed performance analysis of an OSI-SM platform and possibly its comparison to emerging distributed object frameworks such as OMG CORBA could be the subject of a whole thesis. The relevant study had as target to show the salient performance characteristics of the proposed OSI-SM / TMN software framework. Despite its limited scope, this work is significant since it shows that the proposed framework is performant, as detailed above. In addition, it is the very first of this type i.e. no relevant work can be found in the literature.

3.9 Validation

In the previous sections of this chapter we presented various aspects related to the object-oriented realisation of the TMN framework. Throughout those sections we explained how the proposed solutions contribute towards the goals of this thesis, validating the relevant assertions made in the beginning. In this section, we summarise the already presented validation aspects and validate further the proposed framework against additional criteria.

3.9.1 The Proposed Environment as an Object-Oriented Distributed Framework

We will examine first if the OSI-SM / TMN architectural framework together with the proposed realisation framework satisfy the key properties of object-oriented distribution frameworks, as identified in section 3.2.2. We re-iterate through those properties below and examine if and how these are satisfied.

- *An abstract object-oriented specification language that supports inheritance and polymorphism*. GDMO satisfies those requirements with the only drawback that actions and notifications do not map in a satisfactory fashion to abstract remote method calls. The modifications proposed in section 3.5.4.3 rectify this drawback. Of course it is not claimed here that GDMO is a general purpose O-O specification language in the same fashion as CORBA IDL, we comment further on this at the end of this section.

- *Mappings of that abstract language to multiple object-oriented and procedural/modular programming languages*. We have shown mappings of GDMO to C++ in this section that are general enough to be mapped onto other O-O programming languages e.g. Smalltalk, Java. We have also shown a manager mapping to Tcl/Tk which is a procedural scripting language.

- *User-friendly APIs that hide communication and protocol details*. Such APIs were presented through the ASN.1, RMIB/SMIB and GMS O-O concepts and infrastructures. It should be noted that powerful CMIS access aspects (scoping, filtering) and fine-grain event reporting are still available in a natural, “harness and hide” fashion.

- *Dynamic access facilities that obviate the need for static (i.e. pre-compiled) knowledge of object specifications in client applications*. Both the RMIB and weakly-typed SMIB APIs support this requirement. ASN.1 manipulation is static in OSIMIS, i.e. generic applications need to be re-linked with new types, but we have explained how this can be avoided through a data-driven approach in section 3.5.6.

- *Good performance and scalability so that distribution is encouraged and exploited.* The performance was shown to be good from a number of perspectives in the previous section. In addition, both the average application and object sizes are reasonable. Though scalability was not explicitly addressed from a global system perspective, the performance and size figures do not reveal any major deficiencies.

- *Openness in terms of both standard APIs and communication protocols*. The O-O APIs presented in this section could support "horizontal" openness and portability if they were standardised. The NMF TMN/C++ [Chat97] family of APIs, which has re-used many of the concepts proposed here, will be the relevant industrial standard. "Vertical", on-the-wire openness is provided through the Q_3 protocol stack [Q3].

- *Distribution transparencies, in particular access and location.* Access transparency is supported through the Q3 protocol stack, including ASN.1 [ASN1] and BER [BER] for data representation. Finally, location transparency is provided through the OSI directory [X750] mechanism discussed in Chapter 2 and the realisation model proposed in the previous sections of this chapter.

In conclusion, these properties are largely satisfied. The one that is only partly satisfied concerns the *standard* mapping to multiple programming languages. The NMF TMN/C++ API covers only the mapping to C++. Standard mappings to other object-oriented and procedural programming languages are not currently addressed. In comparison, OMG CORBA [CORBA] offers mappings to C, C++, Smalltalk and Java. The approach in this chapter though has shown that "horizontal", O-O distributed system-like APIs are feasible for harnessing the OSI-SM/TMN power and complexity. Standard APIs may be specified for a number of programming languages in the future.

It should be also noted that the OSI-SM/TMN model is *not* a general distributed systems model but targets distributed management systems. It is a composite model in which managed objects are clustered together in "ensembles", handled by applications in agent roles as it was explained in Chapter 2. Despite the fact we have used distributed systems examples to demonstrate the distributed system-like nature of the proposed realisation model, e.g. those involving the simpleStats class, these are degenerate cases in which the ensemble becomes essentially a single object. On the other hand, the fact that such cases can be accommodated reinforces the validity of the assertion that the OSI-SM / TMN framework can be realised in a distributed system like fashion.

3.9.2 Object-Oriented Support for TMN Operations Systems

In Figure 2-22 of Chapter 2 we presented a functional decomposition of the OSF which is the fundamental TMN building block. Having presented the object-oriented realisation framework in the previous sections, we present in Figure 3-23 a concrete engineering object-oriented decomposition of a TMN Operations System, as supported by the proposed OSI-SM / TMN realisation framework. This is a *generic* decomposition which shows how the various facilities of the proposed infrastructure are used.

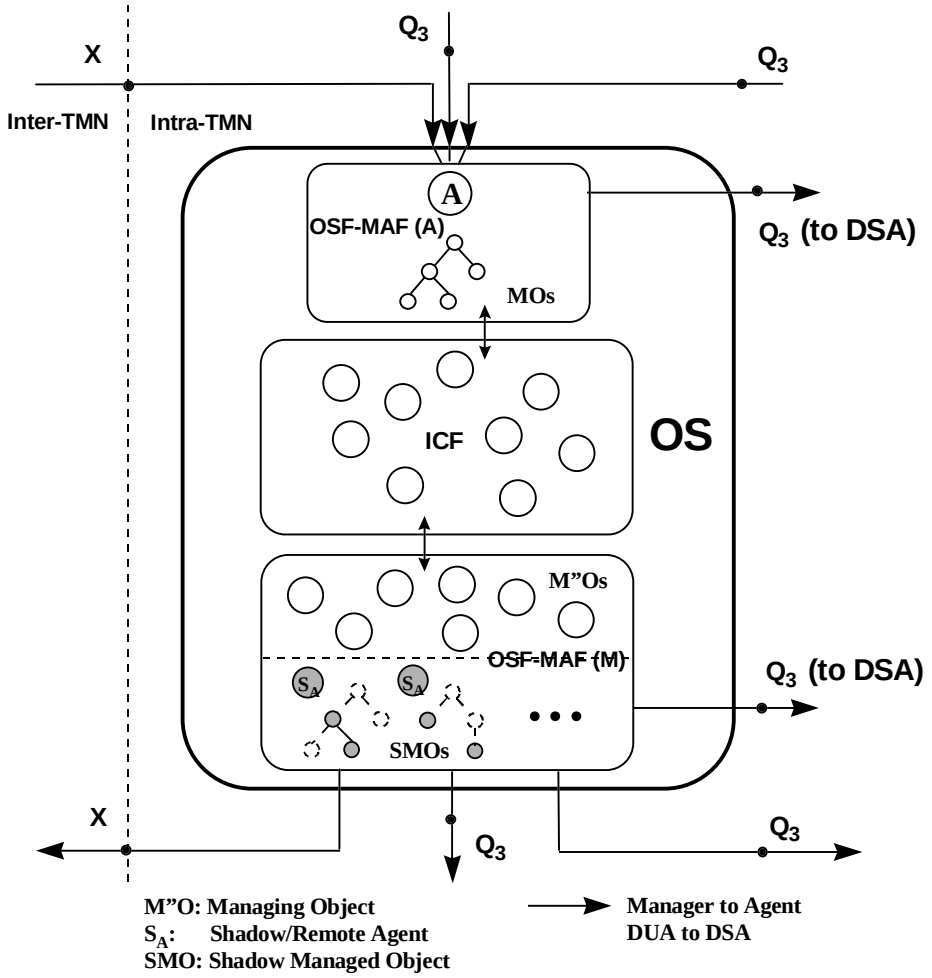


Figure 3-23 Object-Oriented Decomposition of a TMN OS

The Management Application Function in agent role (MAF-A) is realised through the generic agent infrastructure, the Generic Management System (GMS). The managed object classes available across the Q_3 and X interface are compiled through the GDMO compiler and enhanced with behaviour, as was described in section 3.6. Access to them is provided through the agent object instance, which encapsulates the interoperable protocol stack and provides the various CMIS services, including scoping, filtering, synchronisation and event reporting. The

authentication, stream integrity and confidentiality services are provided by the protocol stack, either by the presentation layer [GULS] or by CMISE/ROSE [Bhat96]. The access control service is provided by an Access Decision Function (ADF) object instance which is also encapsulated by the agent object [Pav96b]. The MAF-A contains pre-compiled knowledge of the managed object classes of the Q_3 and X ensembles, including also event reporting, logging and any other SMFs required by the relevant specifications e.g. metric monitoring [X739], summarisation [X738], performance management [Q822], fault management [Q821], etc. Any differentiation between the X and Q_3 interfaces offered to peer and superior OSs is provided through access control [X741].

The Management Application Function in manager role (MAF-A) is realised through the generic manager infrastructure, the Remote MIB (RMIB) and/or Shadow MIB (SMIB). The managed object classes of the Q_3 and X information model in superior and peer systems need to be compiled in order to produce the information necessary for the relevant repository. In addition, if a static, strongly-typed SMIB model is followed, specific SMO classes are produced and possibly enhanced with behaviour as it was described in section 3.5.4. Typically one Shadow/Remote agent and associated SMOs exist for every accessed TMN, as shown in Figure 3-23. Various managing objects implement the OS managing intelligence. A part of the application's management intelligence is realised by various objects implementing the Information Conversion Function (ICF). These interact both with managed and managing objects while their functionality is typically triggered by operations to managed objects "downwards" or through received event reports and results of access to subordinate or peer systems "upwards".

Directory access is required in order to update the directory about the location and capabilities of the OS and in order to discover the location and capabilities of other TMN. It is reminded here that directory access facilities are integrated with the CMISE API, as described in section 3.3.2.3. These are used by the MIB agent, which updates the directory about the location, capabilities and supported managed objects classes and instances for this OS. They are also used by the various RMIB and subsequently SMIB agents (a SMIB agent always contains a RMIB agent) which discover the location and capabilities of other applications.

Finally, such an application can be organised either in a multi-threaded fashion or in a single-threaded fashion as described in section 3.7. In the latter case, manager access to peer or subordinate systems should be asynchronous so that the application can still be active while waiting for a result or results.

3.9.3 Support for Peer-to-Peer Interactions

Another interesting point related to the validation of the proposed framework is natural support for peer-to-peer interactions. As already mentioned in section 2.4.5 of Chapter 2, the inherent asymmetry of the manager-agent model can be a limitation if the relevant engineering concepts separate completely the manager and agent aspects. This is certainly not the case with the proposed realisation model. Though Figure 3-23 presents a hierarchical internal decomposition with the MAF-A, ICF and MAF-M separation, this need not be so if particular peer-to-peer exchanges require it. For example, a managed object and managing object can be realised by the same object instance, acting in both roles. In this case, an operation to that instance may trigger an operation in the opposite direction, with the managed object becoming instantly a managing object. In fact, such operations may also pass the global distinguished name of the invoking object as an action parameter so that the accessed object can perform an operation in the opposite direction. This bears an exact analogy to the passing of object references in distributed system frameworks such as OMG CORBA, as discussed in Chapter 4.

3.9.4 Implementability in Terms of the Overall Required Software

Since one of the objectives of this thesis is to show the feasibility and implementability of the OSI-SM/TMN framework, it is interesting to investigate how much software in terms of lines of code is necessary to provide this type of flexible object-oriented environment. We will assume that there exists an upper layer OSI stack, including ACSE, ROSE, DASE implementations and a procedural ASN.1 compiler, which is what most OSI infrastructures provide.

Table 3-13 shows the size of the generic parts of the infrastructure in lines of code. This is in C++ unless it is indicated otherwise. The grand total is 55000 lines which seems reasonable for the relevant functionality and demonstrates the feasibility of the approach. Of course, turning this prototype this into production software would require a substantial amount of additional development. This has been done though by more than one commercial vendors of TMN infrastructure, validating further the relevant concepts.

It should be noted that the Table 3-13 includes only the absolutely necessary TMN platform facilities. As such, it does not include the generic MIB browser [Pav92a] (5000 lines), the generic CMIS/P to SNMP adapter [McCar95] (12000 lines together with the SNMP SMI to GDMO compiler) and the lightweight, secret key based authentication, integrity and confidentiality services [Bhat96]. It should be finally mentioned that the author developed around 65% of the amount of software mentioned in the Table 3-13.

Component	**Lines of code[5] and description**
CMISE	10000 (in C, includes also higher-level support functions)
X.500 access for X.750	3000 (in C)
O-O ASN.1 API / compiler and X.721 attributes	10000 (1000 the compiler in gawk, 2000 the generic ASN.1 classes, 8000 the rest)
GDMO compiler	10000 (8000 the compiler, 2000 the code generating scripts)
GMS Agent infrastructure	10000 (6000 the "kernel", 1500 event reporting, 1500 logging, 1000 access control)
RMIB Manager infrastructure	3000
Tcl-RMIB Manager infrastructure	3000
Generic command-line manager programs	3500 (these are required as debugging tools)
Coordination support	1500
Generic library classes	1000 (string, list, array)
Total	55000

Table 3-13 Amount of Software in the OSIMIS TMN Platform

3.9.5 Further Validation

Finally, the true validation and verification of the proposed framework has been achieved through its use to develop and deploy experimental TMN systems and through its use for additional research by the author and many other researchers. Appendix A provides a short description of the known research work that has been based on the proposed object-oriented TMN platform.

[5] In C++, unless mentioned otherwise.

3.10 Summary

3.10.1 Overview of this Chapter

In this chapter we presented first an introduction to object-oriented design and development principles, followed by the identification of desired properties of object-oriented distributed frameworks. These were used later to measure against them the proposed object-oriented TMN platform. The same properties will be also used in Chapter 4 to measure against them ODP-influenced distributed object technologies.

We subsequently looked at issues behind the realisation of the Q_3 protocol stack. We discussed aspects of realising ROSE and CMISE over the presentation service and presented possible design policies for a CMIS API. The latter is completely hidden behind higher level object-oriented infrastructures but is the fundamental building block for those. We also presented two lightweight mappings for CMISE that operate directly over the transport service and use string encodings for attribute, action and notification values.

We then discussed aspects behind object-oriented ASN.1 manipulation and presented a design that uses polymorphic principles. ASN.1 types map to C++ classes which are automatically produced by an O-O ASN.1 compiler and may be further customised by the implementor. Additional classes provide support for the ASN.1 ANY and ANY DEFINED BY constructs. These generic classes are used extensively in the higher-level managed and agent infrastructures and APIs.

We then discussed extensively aspects behind the realisation of object-oriented manager infrastructures and proposed two models, the Remote MIB and Shadow MIB. These hide completely underlying protocol details but provide access to powerful CMIS aspects such as multiple object access through scoping and filtering and fine-grain event reporting. A mapping of the RMIB infrastructure to the scripting Tcl language was also presented. This may be used for applications with GUIs, i.e. TMN WS-OSs, because of the associated Tk graphical toolkit. We also discussed aspects of the management information repository which represents the necessary "shared knowledge" between agent and manager applications that support a Q_3 interface.

A discussion on the aspects behind the realisation of object-oriented agent infrastructures followed. We presented a generic agent architecture known as the Generic Managed System which separates service and protocol processing from the managed objects. We also proposed a mapping of the GDMO abstract language to C++ in a fashion that behaviour is added through the

redefinition of polymorphic methods, shielding the implementor from unnecessary details. Stub managed objects are produced through a GDMO/ASN.1 compiler and are augmented with behaviour. We also discussed how the "difficult" aspects of OSI-SM can be implemented using O-O principles that harness and hide the relevant complexity.

We then discussed briefly issues behind synchronous vs. asynchronous remote execution models. Synchronous models are more user-friendly but require support for multithreaded execution and concurrency control. Asynchronous models are less user-friendly but do not necessarily require support for multithreading. In either of the two models, a coordination mechanism is needed to "dispatch" activities within a complex application implemented as a single operating system process. An object-oriented coordination infrastructure was presented which shields application objects from the underlying complexity.

We then presented a performance analysis and evaluation of the proposed object-oriented environment, addressing program size, response times and packet sizes. It was shown that the overheads of TMN applications are reasonable and much smaller than widely believed, though this is a relatively subjective judgement. In Chapter 4 we will perform similar measurements for CORBA. These will put the measurements for the OSI-SM based framework into perspective.

Finally, a validation of the proposed framework was presented by examining it against the properties of object-oriented distributed frameworks identified in the beginning. We also explained how the proposed O-O infrastructure can support the various aspects of a TMN OS and considered the amount of software required to build such an infrastructure. Since the ultimate validation of the proposed environment was achieved through its use for further research, design and development, relevant efforts are presented in Appendix A.

3.10.2 Research Contribution

The key research contributions in this chapter are the following:

- The identification and discussion of the general issues in realising transaction-based OSI upper layer protocols that rely on ROSE and ASN.1 in section 3.3.2.

- The detailed presentation of various issues and alternative approaches in realising CMISE APIs that include association control and location transparency features in section 3.3.2.3.

  ⇒ The validation of the procedural, asynchronous, "lowest common denominator" CMISE API approach through the author's design and implementation of the OSIMIS CMISE. This has been used world-wide and influenced both commercial products and subsequent API standards.

- The presentation of issues behind alternative lightweight mappings for the CMIP protocol and the specification of two approaches of a "string-based" CMIS/P that can operate directly over various reliable transport mechanisms in section 3.3.2.4.

- The presentation of the issues behind object-oriented, polymorphic ASN.1 manipulation in OSI upper layer infrastructures and the specification of the generic classes realising the relevant API in section 3.4.

  ⇒ The validation of the O-O ASN.1 approach and API through the author's design and implementation of the relevant generic classes and of an O-O ASN.1 compiler that generates specific classes for ASN.1 types. The O-O ASN.1 is used in the other OSIMIS high-level O-O APIs.

- The presentation of the issues and the design behind object-oriented, polymorphic manager infrastructures and the identification of two major approaches in section 3.5:

  * An agent-based approach of dynamic, weakly-typed nature which was termed the Remote MIB (RMIB).

  * A managed object-based approach which can be both dynamic / weakly-typed or static / strongly-typed, termed the Shadow MIB (SMIB). This includes the mapping of GDMO to O-O programming languages from a client or manager viewpoint.

- The specification of the relevant classes and APIs and the demonstration through examples of their flexibility, user-friendliness and economy in terms of lines of code required for distributed operations, resulting in a rapid system development cycle.
- The specification of a string-based RMIB API and its integration in the Tcl/Tk scripting language.

⇒ The validation of both approaches through the design and implementation of the relevant infrastructures. The RMIB and Tcl-RMIB have been part of OSIMIS and have been widely used for prototype TMN system development. Both the RMIB and SMIB infrastructures have influenced products and subsequent API standards.

- The identification of a logical "bug" in the CMIS/P m-event-report primitive which makes difficult to demultiplex event reports and pass them to the right managing object within a manager application. This was revealed through the validation of the manager aspects of the OSI-SM framework through the RMIB / SMIB infrastructures. The subsequent proposal of a CMIS/P modification that overcomes this limitation was presented in section 3.5.3.2.
- The identification of an inefficiency in the GDMO Action template which results in non-natural mappings to object-oriented programming languages. This was revealed through the validation of the framework through the RMIB / SMIB infrastructures. The subsequent proposal of a GDMO modification that overcomes this limitation in section 3.5.4.3.
- The presentation of the issues and the design behind object-oriented polymorphic agent infrastructures in section 3.6:
  - The presentation of an object-oriented architecture which separates service and protocol aspects from the managed objects - the Generic Managed System (GMS).
  - The presentation of a GDMO to C++ mapping which uses only single inheritance and can be also provided in languages such as Smalltalk and Java.
  - The presentation of a polymorphic managed object class API which can be used to add behaviour by redefining the relevant polymorphic methods.
  - The presentation of three different approaches for maintaining consistency between managed objects and associated resources.

- The discussion how the “difficult” OSI-SM aspects can be provided, including name resolution, scoping, filtering, atomicity, allomorphism and persistence.
- The discussion on how event reporting, logging and the rest of the OSI SMFs can be supported in agent environments.

⇒ The validation of all those concepts through the design and implementation of the GMS infrastructure. This has been widely used for prototype TMN system development and influenced commercial products and subsequent API standards.

- The demonstration that the OSIMIS O-O manager and managed object APIs bear a lot of similarities to those of distributed object frameworks such as OMG CORBA which were developed later. This shows that the complexity and power of the OSI-SM/TMN model can be harnessed behind O-O platform APIs that support abstractions similar to those of emerging distributed object frameworks.

- The demonstration that the perceived limitations of the OSI-SM manager-agent separation can be overcome through O-O realisation infrastructures that allow objects to take managed or managing roles at any time. This supports the natural realisation of peer-to-peer interactions for TMN OSs.

- The demonstration that the proposed object-oriented TMN realisation framework has modest requirements in terms of memory resources and it has good performance characteristics, even with modest computing infrastructures. In addition, the generated packet traffic from the supporting OSI protocols is reasonable. These conclusions were drawn through a detailed performance analysis and evaluation in section 3.8 which is the first of this type i.e. there is no similar work in the literature.

- The demonstration through the whole approach that the TMN is not a pile of complex and difficult to implement recommendations. On the contrary, the relevant O-O methodology and specification aspects lend themselves naturally to object-oriented realisation through the “harness and hide” principles presented in this chapter. The result is a powerful, flexible, performant and easy-to-use distributed environment.

- Finally, a more general contribution to telecommunication problem solving which moves away from the protocol-based “bits-and-bytes” approach of the past towards general-purpose, easy-to-use distributed computing environments that satisfy the relevant requirements of both interoperability and software openness.