Active Objects in TMN

A. Vassila, G. Pavlou, G. Knight
Department of Computer Science, University College London,
Gower Street, London WC1E 6BT, UK
Tel: +44-171 419 3679, +44-171 380 7366, +44-171 380 7215
Fax: +44-171 387 1397
e-mail: n.vassila, g.pavlou, g.knight @cs.ucl.ac.uk

Abstract

Telecommunications Management Network (TMN) systems use the object-oriented information modelling techniques and communication facilities provided by OSI Systems Management (SM). TMN interfaces are specified in terms of rather passive Managed Objects (MOs) with prespecified behaviour. In this paper, we propose the concept of Active Managed Objects (AMOs) as the means to specify and express arbitrary management functions (including those specified in [1]) through a suitable scripting language. AMOs may be delegated to a TMN application in agent role and function close to other managed objects they access. Such a facility increases the intelligence and autonomy of TMN applications and enables the expression of management functions with arbitrary intelligence. Also, since it uses the normal TMN mechanisms for information modelling and access (GDMO, CMIS/P), it could be potentially standardised. In this paper, we describe the AMO concept, examine tha information model and scripting language aspects and present our implementation experience.

Keywords

TMN, OSI Management, Active Managed Objects, Management by Delegation

1 INTRODUCTION

This paper describes work at UCL[*] which is aimed at increasing the intelligence and autonomy of components of TMN[2] systems. TMN systems normally make use of the services and protocols defined within OSI management[3]. The OSI management information model[4] is based on an object-oriented database paradigm with rather passive 'Managed Objects' (MO) encapsulating management information on a managed system. MOs are accessed by

[*] This work is partially funded by the ACTS project MISA (AC080).

applications in a managing role with the access being mediated by an ‘agent’ function at the managed system, which provides database-like functions enabling target MOs to be selected in a flexible and data-dependent way.

Our work is motivated by the following considerations:

- There are some exceptions to the general passive nature of MOs, for example the *discriminator*[5] objects which take an active role in the handling of asynchronous notifications, and the *monitor metric*[6] and *summarisation*[7] objects which calculate various statistics based on values obtained from other local MOs. We have found these more ‘active’ MOs to be powerful tools in management - in particular, they enable much management activity to remain local to the managed system which would otherwise require interaction with a remote managing application. However, their ‘managing intelligence’ is of static nature, parameterised only through MO attributes and actions. We have sought, therefore, to generalise the active MO concept by providing a framework for a MO whose precise behaviour is specified in a program or script downloaded at runtime.
- Moving management intelligence close to the resources being managed should eventually improve the overall performance of the management system as it reduces network traffic, it enables autonomy of managed systems and at the same time improves fault tolerance of the whole system as constant communication between managing and managed entities is no longer substantial.
- The OSI management model has little to say about the managing function beyond specifying the operations it is allowed to invoke. However, the TMN architecture involves a hierarchy of applications in a manager role in which high level ones control the behaviour of their subordinates. We have developed a system in which the behaviour of a system in a manager role is specified in a runtime script. It is convenient to represent a subordinate managing system and its script as a MO which can be manipulated by a high level managing system.

An answer to the above considerations is to have a script executed by an interpreter that performs management operations. This approach has also been followed by other research groups: In Columbia University the Management by Delegation paradigm[8] has been defined, and a system implementing it has been developed. In addition, in INRS Telecommunications Canada[9], another delegation framework is being developed for application management, as well as for systems administration. [10] also describes a prototype implementation of an ‘intelligent agent’ system for application management using CORBA. Other work includes the `scotty` implementation package[11]. But, what we have identified, is that the execution of the script should be controlled by a higher level (probably remote) management entity. This entity should be able to control every aspect of the script. Given the TMN context for the work this control must be through the normal TMN mechanisms. This means that the invocation of a script must be represented as a MO; the higher level managing system then interfaces with the script by manipulating the MO’s attributes, actions and notifications. We call such MOs ‘Active Managed Objects’[12] (AMO). A similar approach has been followed by ISO and has been documented in [13].It has to be noted that our primary interest is in prototyping the idea of AMOs and assess their usability and effectiveness, rather than presenting a fully implemented system.

In Section 2 of this paper we clarify the AMO concept and in Section 3 we present a detailed system design for an AMO. This design is, in fact, independent of the scripting language used -

however, the capabilities of this language are clearly important. In Section 4 we discuss language issues in general, while in Section 5 we give a detailed description of the OSI definition of the AMO. A pilot implementation of an AMO has already begun using the OSIMIS TMN platform; in Section 6 we describe this work.

2 THE AMO CONCEPT

Figure 1 shows the main AMO system components. The ‘Managed System’ can be any OSI managed system, complete with the usual collection of MOs representing the resources being managed. In addition to these MOs, there is an AMO representing a script. From the point of view of OSI management there is nothing special about an AMO; for example, it presents a conventional interface to the management agent, which can invoke the normal range of operations (get, set etc.) upon it. The ‘Remote Managing System’ in Figure 1 can manipulate the AMO by sending CMIP request PDUs to the managed system.

In Figure 1 the Remote Managing System is shown as a stand alone component. However, it may itself be implemented as a script and AMO in another managed system which is itself controlled by a higher level managing system. The interpreter in Figure 1 interprets a language which includes functions which perform management operations. We have investigated with two kinds of functions: those which provide access to local MOs and those which provide access to MOs on remote managed systems.

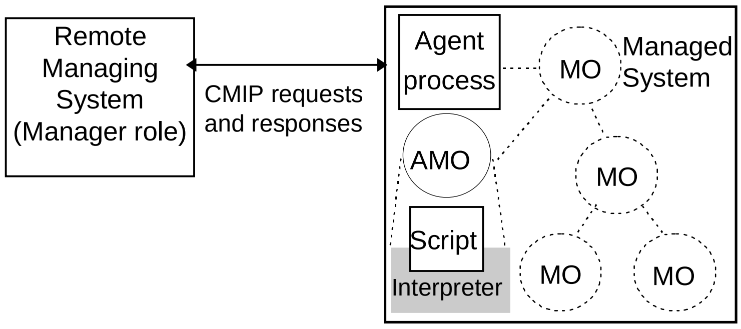

Remote Managing System (Manager role)
CMIP requests and responses
Agent process
MO
Managed System
AMO
MO
Script
Interpreter
MO
MO

Figure 1 The Active Managed Object Model.

2.1 Local MO Access

This can be used to extend the capabilities of managed systems so that they can accept and interpret delegated programs, expressed in a suitable scripting language. The latter enables access to other managed objects in the local system through the full range of operations available at the object boundary i.e. get, set, action etc. The delegated logic assumes that all the other objects are local and accesses them through their distinguished names in that system e.g. *{subsystemId=nw, entityId=x25}*. The results, collected by the AMO, are then accessible to a managing application via the agent process in the normal way (Figure 2).

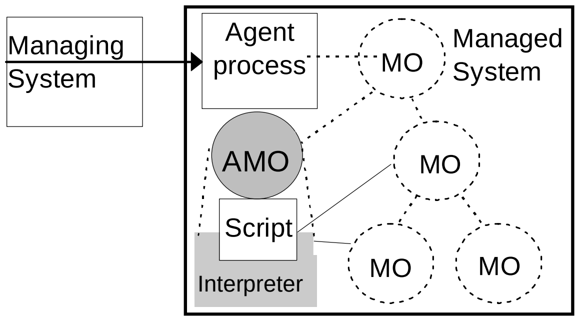
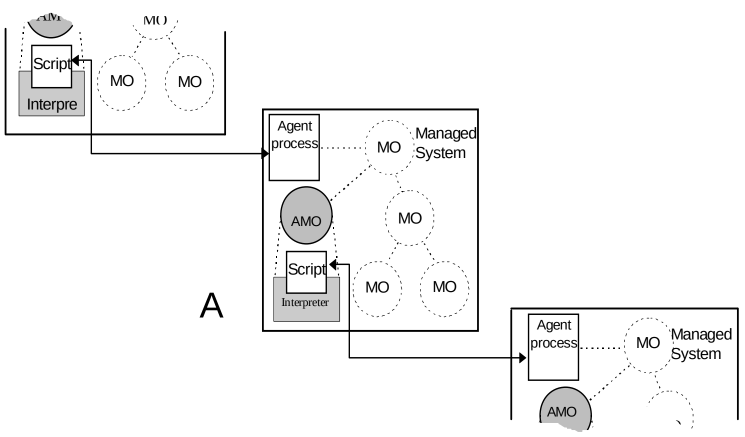


Figure 2 Access to local MOs

Within a TMN system this capability can be used, for example, to perform some standard or proprietary TMN management function and make the results available to an application in a manager role. It can also be used to provide autonomous control within a managed system.

2.2 Access to Remote Managed Systems



An example of this is shown in Figure 3. Here the managed system marked ‘**A**’ in the centre of the figure contains an AMO which controls an interpreter offering the full CMIS client service[14]. This service is being used to manage a subordinate managed system. At the same time, ‘**A**’ is itself being managed by a higher level managing system.

Figure 3 Hierarchical Management viaAMOs.

This gives fully hierarchical management controlled by scripts which can be downloaded at runtime. Strictly, it is not correct to describe the systems in Figure 3 as ‘managed systems’ since they embody the managing and the managed roles. For remote access all MOs must be addressed through global rather than local names. The global name of a MO consists of the managed system identifier (that can be resolved to its address using the Directory), followed by the distinguished name of the MO to be accessed. The scripting language and underlying infrastructure may provide location transparency so that local and remote object accesses are indistinguishable. Local accesses will retain the CMIS properties but without using the CMIP stack. In TMN terms, local accesses will occur through a q reference point while remote accesses through a Q interface.

It should be noted that the AMOs in Figure 2 and Figure 3 are identical from the informational point of view. Their management definition remains the same, while the difference lies solely in the capabilities of the scripting language. In the first case, operations within the local managed system are supported - one can expect these operations to be quick and cheap as there probably is a single address space. In the second case, operations may be performed on **any** managed system. Access is via an OSI management agent offering the normal facilities available through CMIS such as multiple object selection, filtering, access control, etc.

Although the systems in Figure 3 are shown performing remote operations there is no reason why they should not access local objects by directing CMIS requests at their local agent processes. This way, there will be no need for the script programmer to distinguish between local and remote objects, but the advantages of using local operations will be lost.

3 THE RUNTIME ENVIRONMENT

After having introduced AMOs, we now investigate their functionality in some detail. Our aim is to construct a language independent system so that the AMO can work with more than one language and interpreter. Therefore, we must identify the language independent aspects of the system and define an information model for an AMO which encapsulates these. The identified purposes of AMOs can be summarised as:

- To provide management access to scripts; i.e. to store information on scripts and identify the interpreters that will execute them.
- To initiate and terminate execution of scripts according to managing systems requests and local rules.
- To communicate with a running script for the purpose of passing parameters and notifications to scripts and retrieving results.
- To provide/present/structure results for managing applications.

3.1 AMO Functionality

The script and its invocation are the real resource that is managed by the managing application via the AMO. Therefore, the AMO must represent all the aspects of the script that could be of interest to the managing application and that are likely to be changed during its lifetime.

The first requirement is to deliver the script to the managed system and store it there. This is done by storing the script as an attribute in the AMO, and delivering it using regular CMIS operations. It is important to note that we assume that our system will be persistent, that is, all the AMO components will survive any system failure.

Once a script is started, it represents a separate thread of control within the managed system. This thread will exist until either a managing application requests termination, the script voluntarily terminates or the managed system forces termination (perhaps because the script is consuming too many resources). Typically, a script will use an event-driven programming style. The events of interest being, for example, notifications from other MOs within the managed system, receipt of communication from other managed systems, timer events and receipt of signals from the manager application or the local agent process. The execution of the thread will normally be suspended whilst it waits for an event.

It is important that the managing application is not *obliged* to access the AMO and script during its execution for reasons other than an emergency, or to retrieve results and statistics. This means that the AMO must carry all the knowledge that concerns the execution of the script as directed by the managing application. However, although, as stated above, the managing application is not obliged to access the AMO during script execution it may do so in order to vary aspects of script execution - for example, to adjust polling frequency.

At the very least, a script must be able to communicate with its local environment and with its managing application. One role for a script could be to provide sophisticated, autonomous processing of local events. Such events will engender notifications from MOs which the AMO must pick up and pass to its script through some sort of asynchronous signalling mechanism. We propose that the AMO class inherits from the discriminator class which will enable it to receive notifications and filter them before generating signals. Normally the filter will be provided by the managing application which created the AMO and can be changed at any time. A signalling mechanism can also be employed to allow asynchronous communication from the managing application. This is invoked using the CMIS M-ACTION service.

It is evident that a managed system with AMOs and scripts embedded within it has many of the characteristics of a multiprocessing, multi-user operating system. It is unwise in such a system to assume that all scripts are benign; it is important that the managed system can defend itself against rogue scripts. This defence, one can envisage, could be exercised through pre-emptive scheduling mechanisms parameterised by some sort of priority value. Furthermore, an interpreter for a safe language would provide a major defence against malign scripts, by forbidding all access other to the script's own state variables and the MIB.

It is important to note that any management operations invoked by the script must behave exactly as if they were invoked directly by a managing process, because the AMO acts on behalf of a specific managing application. The normal access control mechanisms are applied to the attributes of the AMO and to operations on managed objects invoked by the script. The issues involving the mechanism by which this is done are under investigation. It is evident, however, that by identifying the managing application that created the AMO in every management operation, access control can be performed in a uniform fashion.

From the requirements above, we can summarise a number of features which must be present in the AMO information model:

- **Actions which can be used by the managing system to start, stop execution and to signal the script thread.** The result of the first action will be the creation of a new thread of execution for the script. The second action will result in a previously created script thread being terminated. The side-effects of this must be further investigated, as there is the problem of destroying a script while executing, which can cause corruption of management information. The third action will cause the AMO to deliver a signal to the script. The exact mechanism through which such signals are handled within a script are language dependent. Typically, the effect will be to cause the script to exit from a waiting state and execute the appropriate code.
- **A notification of script termination and exceptions.** The AMO should be able to notify the managing system about the script execution (mode of completion, execution error etc.). asynchronously. The normal mechanisms are employed with the AMO issuing a notification which triggers an event report through a discriminator.
- **A notification to be invoked from within the script.** It is important that the script should be able to trigger notifications voluntarily. Thus the script programmer can ensure that a notification (and hence an event report) is issued when a situation that requires further involvement by a managing application is identified. Again, the precise way in which this facility is invoked is language dependent.
- **A filter.** This is used to filter notifications coming from within the managed system and determine whether these should result in signals being passed to the script thread.
- **An interval timer mechanism.** This can be used in order to simulate polling procedures. The AMO will signal the script at specified intervals - minimising the need for managing application intervention. Note that polling can also be implemented by calling a timeout function from within the script. However, a mechanism controlled through the AMO, has the benefit of allowing the managing application to adjust operations at runtime.

- **An attribute to identify the script to be run and the controlling managing application**. The first will contain a string representation of the script. It will be set by the managing entity when the AMO is created. The second attribute is essential for the operation of the AMO system because it has to appear in all communications with local or remote systems, as the script inherits all the permissions that the issuing managing application owns.

4 FACILITIES OF THE SCRIPTING LANGUAGE

As stated above, the AMO has been designed so as to impose minimal constraints on the language employed for the script. We envisage scripts implementing anything from one line statistical calculations to complex control policies - so a range of languages may be deployed to suit these different applications. In this section we examine two aspects of the facilities these languages must provide. First we look at the facilities which are the counterpoint to the generic facilities provided by the AMO - those that implement API 1 in Figure 4. Next we outline the requirements for a general purpose language suitable for the TMN environment. This must provide the full functionality offered by the OSI model - API 2 in Figure 4.

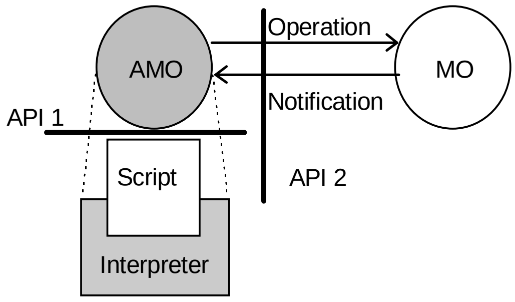


Figure 4 Script APIs.

4.1 The interface to the AMO

It is useful to think of the script executing in conjunction with an AMO as analogous to a user space process executing in a general multitasking operating system. Interactions take place in two directions. The script must be able to read and write certain AMO attributes so that, ultimately, it may communicate with the remote managing application (similar to Unix ioctl() calls). It must also be able to cause the AMO to emit notifications and react to signals from the AMO (similar to the Unix signal()/Kill() mechanism). The script can be started during AMO creation or through a subsequent activation action while a termination signal will be passed to it at deletion in order to terminate gracefully.

In the following example script, the scheduling primitives can be seen, as well as an example of the CMIS extensions to the scripting language (described in Section 4.2).

```
RegisterTimers (900000, scollect)                                  Initialisation code
ifnumber=Get systemId=grappa@interfacesId=NULL ifNumber;
UpTime=Get systemId=grappa@internetSystemId=NULL sysUpTime;
proc scollect () {                                                 Callback function
  date = getSystemDate();
  upTime=Get systemId=grappa@internetSystemId=NULL sysUpTime;
  if (upTime<Uptime) issuenotification(systemId=grappa “has been reset!!”);
  for (j==1, j<=ifnumber, j++) {
    dn="systemId=grappa@interfacesId=NULL@ifEntryId=SnmpInteger=“j;
    inucastpkts = Get (dn “ifInUcastPkts”);
    outucastpkts = Get(dn “ifOutUcastPkts”);}
  SetOutputInfo (date, UpTime, inucastpkts, outucastpkts);
}
```

In the above example, the procedure `scollect()` is executed every fifteen minutes. Several values are retrieved from the local MIB-II, written in the `OutputInfo` AMO attribute and a notification is emittted if needed. This notification can be sent as an event report or logged, depending on the actions of the managing application (creation of a discriminator).

4.2 A general purpose language for TMN management

In this section we examine the required facilities of the scripting language in order to enable the expression and realisation of management policies. The base language facilities with respect to the available data and control structures need to be comparable to those of compiled programming languages. Object-oriented aspects, i.e. classes, inheritance and polymorphism, are necessary for structuring more complex scripts. Scripting languages that can be thought as potential candidates are Tcl [15]/ Scheme[16] and their object-oriented extensions and of course Java[17]. These languages need to be extended with management facilities that will enable interaction with the TMN environment and the MIB. It is these extensions we consider in this section.

A script may access managed objects that are 'local' to the AMO i.e. part of the same Management Information Tree (MIT) across a TMN interface, or 'remote' as part of another MIT and TMN interface. Remote object accesses may incur an increased latency as an external representation of the 'method call' will travel across the network, involving protocol overhead etc. As such, the script should be aware of the system in which it executes through the name of the top MIT object so that it can distinguish between local and remote access. On the other hand, the syntactic aspects of object access should be in principle the same for both local and remote objects. Global distinguished names may be used to provide location transparency while local distinguished names will default to the local environment.

It appears at first that this unification is difficult: local access can be modelled as managed object method invocation, in the same fashion as in any object-oriented environment. On the other hand, remote access should reflect the facilities of CMIS, where access is mediated by an agent. The difference is crucial and, in a first consideration, it appears to have an effect on the language extensions. As it has been shown in [18], higher level abstractions on top of CMIS are possible, providing the illusion of direct object access. In fact, it is possible to unify the two types of extensions, assuming that CMIS facilities such as scoping and filtering are also available for local access. In fact, such facilities increase the available flexibility and simplify the script logic, as well as optimising the management traffic for remote access. Managed objects may be addressed through their distinguished names (see example script in section 4.1), or through (implementation dependent) object references i.e. handles or pointers. Unifying local and remote access means of course that association establishment has to be completely transparent to the script i.e. hidden.

The style of interaction should be both synchronous and asynchronous. A synchronous style of interaction has method call semantics and results in natural, linear program flow but blocks the thread of execution until the call returns. An asynchronous style of interaction has message passing semantics and requires the management of state since the result will be returned through a 'callback'. Asynchronous facilities are of paramount importance in single threaded environments as they prevent blocking the whole application for remote accesses with increased latency.

5 AN INFORMATION MODEL FOR AN AMO

In the light of the analysis in Sections 3 and 4 we can now give a more formal specification of an AMO which inherits from theeventForwardingDiscriminator managed object class.

ATTRIBUTES

`Id/Name`	Identifies the AMO and name it in the OSI MIT.
`Authorisation` (a DN)	Identifies the managing system that created the AMO. It will contain the distinguished name of the managing which will be used as input to access control functions.
`Script` (string)	Holds the delegated script.
`Timer` (sequence of integers)	Used to change the intervals between script executions without having to alter the script itself.
`Filter` (CMISFilter)	A filter for notifications from local MOs. Notifications which pass the filter result in signals to the script.
`Parameters` (name-value pairs)	To be provided to the script when it starts. The syntax is a set of pairs that indicate the name of a parameter and its value.
`InputInfo` (name-value pairs)	Settable by the managing application, readable by the script every time it is changed.
`OutputInfo` (name-value pairs)	Gettable by the managing application, writeable by the script to indicate some results from its execution.
`Priority` (integer)	Indicates the priority that should be given to the script execution.

Apart from the standard notifications or actions that any managed object can include, the AMO will also support the following. The need for these was discussed in Section 3.1.

NOTIFICATIONS

`TerminationInfo`	Triggered when the script terminates.
`InformManager`	Triggered as a result of the `issuenotification()` function.

ACTIONS

`ActivateSThread`	The AMO activates the script thread upon receipt of this action.
`DestroySThread`	The script thread is deleted.

6 IMPLEMENTATION EXPERIENCE

In this section we present considerations regarding the implementation of AMOs. The purpose of this presentation is twofold: to explain how the AMO concept has been implemented in our environment i.e. the OSIMIS TMN platform; and to identify the necessary requirements in order to implement AMOs on other TMN platforms.

The first important consideration is related to the choice of the underlying scripting language. In most TMN platforms, APIs like those depicted in Figure 4 are implemented in C/C++, so the first important requirement for the scripting language is to be extensible and able to interface easily to C/C++.

One of the reasons for choosing Tcl as opposed to e.g. Scheme as the scripting language to verify the AMO concept, apart from the existence of the Safe-Tcl interpreter, was the ease with which it interfaces to C/C++. The interface between Tcl (or any similar scripting language) and the 'encapsulating' C/C++ environment is in two directions:

- from Tcl to C/C++, for accessing local or remote managed objects and for manipulating the AMOs own attributes and emitting notifications
- from C/C++ to Tcl, for starting the AMO script, receiving notifications from local or remote managed objects and receiving input from actions invoked on the AMO (including setting its attributes, deletion etc.).

The next important consideration has to do with combined event handling in the scripting language and the encapsulating C/C++ environment i.e. the TMN application in agent role. Both these systems need to deal with events with respect to communication endpoints and timer alarms. In our case the starting point is the OSIMIS system which implements a managed system as a single Unix process without thread support. As an AMO script will execute in the same operating system process with (part or all of) the encapsulating management agent, it is necessary to be able to combine their events so that there is a central listening and dispatching loop, serving one or more AMO scripts and the surrounding agent. There are two different possibilities for accomplishing this:

1. through a scripting language interpreter that can be interrogated about the endpoints and alarms it deals with so that they can be combined with those of the underlying TMN C/C++ system; or
2. through a scripting language that can be extended with the endpoints and alarms of the underlying TMN C/C++ system.

In both cases, the assumption is that the target central mechanism can be extended with 'foreign' descriptors and alarms. In our implementation we have followed the second approach as it suited both the nature of Tcl and the OSIMIS process coordination mechanism. An extension of the OSIMIS event handling mechanism passes control to Tcl and integrates with it the OSIMIS C++ endpoints and timer alarms.

Finally, the most important implementation consideration is related to the TMN platform APIs for accessing other managed objects, local and remote, and interacting with the AMOs own attributes, reacting to actions and emitting notifications. A key feature of scripting languages such as Tcl (or Scheme) is that the main data types are the string and list while other complex types can be emulated through these. An important requirement for any underlying TMN platform is that its APIs should support the manipulation of attributes, actions, notifications and their values through string representations. The same applies also to distinguished names, filter expressions and other CMIS-level parameters. The OSIMIS APIs support string expressions in addition to the native C/C++ types, so it has been straightforward to map these onto Tcl language extensions. When accessing local managed objects, scoping and filtering may be supported in addition to accessing objects on a one-by-one basis; this provides additional flexibility and is particularly important for presenting the same access paradigm for both local and remote objects, as it was explained in Section 4.2. In addition, access control functionality is necessary as the script / encapsulating AMO assumes the identity of the managing application that 'owns' it. In OSIMIS, the local managed object access API allows the evaluation of scoping and filtering parameters. In addition, an object instance modelling the Access Decision Function (ADF) is globally available and this allows for the evaluation of access control rights in order to be able to grant or deny access. When accessing remote managed objects, these facilities are available through the OSIMIS RMIB manager support infrastructure. Local and remote managed object accesses can be distinguished through the global name prefix: the scripting language knows about the environment in which it executes through the name of the top MIT object e.g. {c=GB, o=UCL, ou=CS, cn=ATM-CM-OS, networkId=ATM}.

In summary, the key requirements in order to be able to implement AMOs on any TMN platform with C/C++ APIs are the following: a flexible scripting language that integrates easily with C/C++ and that is safe for the local system; the possibility for a combined process coordination mechanism that integrates the scripting language and C/C++ TMN platform events; and flexible managed object access APIs for local and remote objects that support scoping, filtering and access control bcally, as well as accept string parameters.

7 SUMMARY AND FURTHER WORK

In this paper we have identified the need for programmable management facilities within the TMN environment and have investigated how the execution of such programs may be controlled using existing TMN mechanisms. We have identified two APIs which need to be available to program scripts; one provides access to the local environment in which the script is embedded, the other provides access to MOs in local and remote systems. These APIs have been analysed in some detail and a scheme for their implementation within the UCL OSIMIS system has been presented.

The control of the script execution is effected by representing the script and its execution as a MO - a concept we have named an 'Active Managed Object'. The requirements for the attributes, notifications and actions of an AMO have been studied and our conclusions have been presented.

Currently a pilot implementation which implements a subset of the facilities of the two APIs above exists, using the Tcl scripting language. This implementation will be used to assess the effectiveness of the AMO concept and will gradually be extended to include more of the required facilities, as well as to assess the efficiency of the chosen language.

8 REFERENCES

[1] ITU/CCITT Recommendation M.3400 - TMN Management Functions, October 1992.

[2] ITU/CCITT Recommendation M.3010 - Principles For A Telecommunications Management Network, October 1992.

[3] ITU-T X.701, Information Technology - Open Systems Interconnection - Systems Management Overview, June 1991.
[4] ITU-T X.720, Information Technology - Open Systems Interconnection - Structure of Management Information - Part 1: Management Information Model, January 1992.
[5] ITU-T X.734, Information Technology - Open Systems Interconnection - Systems Management - Part 5: Event Report Management Function, 1992.
[6] ITU-T X.738, Information Technology - Open Systems Interconnection - Systems Management - Part 11: Metric Objects and Attributes, 1994.
[7] ITU-T X.739, Information Technology - Open Systems Interconnection - Systems Management - Part 13: Summarisation Function, 1994.
[8] G. Goldszmidt and Y.Yemini, Evaluating Management Decisions via Delegation, in *IFIP International Symposium of Network Management*, April 1993.
[9] Jean-Charles Grégoire, Management with Delegation, in *IFIP AIPs Techniques for LAN and MAN Management*, Paris France, 1993.
[10] P. Steenekamp and J. Roos, A Framework for Policy-based Agents: Implementation of an Application Management Scenario, in*IFIP/IEEE DSOM Workshop*, Italy, 1996.
[11] J. Shönwälder and H. Langerdörfer, Tcl Extensions for Network Management Applications, in *3rd Tcl/Tk Workshop*, Toronto, 1995.
[12] A.Vassila and G.Knight, Introducing Active Managed Objects for Effective and Autonomous Distributed Management, in *Bringing Services to People, IS&N Conference,* Heraklion Greece, 1995.
[13] ISO/IEC DIS 10164-21 - Information Technology - Open Systems Interconnection - Command Sequencer for Systems Management, 1996.
[14] ITU-T X.710 - Information Technology - Open Systems Interconnection - Common Management Information Service/Protocol, Version 2.
[15] J.K.Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley, 1994.
[16] William Clinger and Jonathan Rees (eds.), Revised^4 Report on the Algorithmic Language Scheme, November 1991.
[17] Sun Microsystems, The Java^{TM} Language: A White Paper, 1995.
[18] G.Pavlou, G.Knight, K.McCarthy and S.Bhatti, The OSIMIS Platform: Making OSI Management Simple, in *Integrated Network Management IV pp.480-493*, Chapman and Hall, 1995.

9 BIOGRAPHIES

Anastasia Vassila graduated from the Computer Science Department of the University of Crete in 1993. Currently she is a PhD student in UCL under the supervision of Graham Knight. She does research on integrated network and systems management, emphasising on autonomous, event-driven management in the OSI/TMN framework.

George Pavlou received his Diploma in Electrical and Mechanical Engineering from the National Technical University of Athens in 1982 and his MSc in Computer Science from University College London in 1986. He has since worked in the Computer Science department at UCL mainly as a researcher but also as a lecturer. He is now a Senior Research Scientist and has been leading research efforts in the area of management of broadband networks and services.

Graham Knight graduated in Mathematics from the University of Southampton in 1969 and received his MSc in Computer Science from University College London in 1986. He has since worked in the

Computer Science department at UCL mainly as a researcher and lecturer. He is now a Senior Lecturer and has led a number of research efforts in the department. These have been mainly concerned with two areas: network management and ISDN. Currently he is leading the UCL effort in three EU-funded projects in the areas of network and systems management and broadband networks.