The OSI Management Information Service
User's Manual

Version 1.0, for system version 3.0

Preface

TBA

Acknowledgements

A lot of people have contributed to OSIMIS, the following list includes the major contributors.

Graham Knight of UCL introduced me to the area of network and systems management. He first suggested the possibility of a "generic managed system" and has influenced the OSIMIS high-level concepts and direction. He also contributed the first version of the example UNIX managed object and wrote a nice tutorial, parts of which have been incorporated in Section 5.1 of this manual.

Simon Walton, formerly of UCL and currently at UCLA, wrote the first embryonic CMIS/P implementation back in 1987 and implemented the first (non-generic) OSI management agent for the ISODE implementation of the ISO/CCITT Transport Protocol. Though these have been largely redesigned, some of his design concepts have been influential, especially regarding the CMIS API.

Saleem N. Bhatti of UCL implemented the log control systems management function, the OSI Internet MIB and the CMIS filter parsing and printing stuff and has contributed the SMI Attribute and Syntax support section of this manual (5.3). He has also being giving excellent feedback regarding the OSIMIS concepts and APIs and has influenced some of the most important GMS design decisions, so he qualifies as the deputy OSIMIS engineer.

James Cowan of UCL has written the MIB browser (he is the InterViews guru), he has also produced a prototype version of the full "Remote MIB" high-level access API and has been working on a GDMO MIB compiler - these two will be incorporated in future OSIMIS releases.

Kevin McCarthy of UCL has implemented the proxy version of the OSI Internet MIB which will be incorporated in future OSIMIS releases.

Derick Jordaan and *Andreas Mann* of IBM ENC Heidelberg have provided very useful general feedback and Derick also contributed a prototype implementation of an experimental "Shadow MIB" high-level access API (not included in this release). *Wilko Eschebach*, again of IBM ENC, has been porting the latest OSIMIS versions to AIX.

Adarsh Sethi, formerly of IBM Research Zurich and currently with the University of Delaware has ported OSIMIS initially to AIX and provided useful feedback. *Metin Feridun*, again of IBM Research Zurich, ported later versions to AIX and also tested interoperability with the IBM NetView CMIP and provided logical bug fixes.

Finally, *Bill Anderson* of MITRE ported OSIMIS to 386BSD and provided general feedback.

George Pavlou
London, England
January 1993

1. Introduction

It should be noted first that this is version 1.0 of the OSIMIS User's Manual describing version 3.0 of the software. The reason the documentation has a different version number to the software is because it is not yet complete. The full version of the documentation for the version 3.0 of the system should be also version 3.0, we expect this to happen in the next months.

TBA

1.1 Fanatics Need Not Read Further

First of all, OSIMIS uses ISODE to operate an upper layer management stack over either pure OSI environments (TP0/X.25 or TP4/CLNP) or over TCP/IP using the RFC1006 method. A lightweight OSI management stack (CMOT) is also possible directly over TCP/IP. TCP/IP zealots and OSI purists will sigh but the world will never be perfect.

Then, there is the great debate between OSI and Internet management. Though having OSI in its acronym, OSIMIS is meant to be a platform in which the word Open is not exclusively conceived in the OSI sense. Though currently an OSI management platform, there are plans for extensions: OSI to Internet proxy infrastructure and systems and, more important, potential for application of the same concepts and infrastructure for SNMP generic managed systems and access methods.

Furthermore, there is the debate between OSI and ODP, zealots of the latter claiming there is no need for management protocols/models with respect to systems management but the same communications mechanisms can be used to manage distributed "objects". And also, the extreme aspect that management is not needed at all, all is needed is intelligent self-organising protocols.

The role of OSIMIS in this maelstrom is to show the usability and power of OSI management and to combine it with the strengths and the installed base of the Internet-capable resources and systems in order to provide real management solutions. Advanced ODP-based approaches and their compatibility with the more traditional OSI and Internet protocol-based ones will also be investigated when they mature. Finally, OSIMIS itself and the possibilities it offers is the answer to the claim that management is not needed.

1.2 A Note on the Implementation

OSIMIS does not claim to be production software. Despite that, effort has been made during the development to employ good software practices which result in efficient implementations. Data copying is absolutely minimal, though this may create problems to inexperienced management application implementors as the actual data is passed across instead of copies. It is advised that implementors read very carefully the *Attr* class specification in Section 4 of this manual.

Though no performance measurements have been conducted, experience has shown OSIMIS-based applications to be very efficient, in the context of course of the ISODE stack and its ASN.1 handling. Formal performance studies will be conducted in the future.

1.3 Changes From Previous Releases

It is noted first that by previous releases are meant only the versions 2.98 and 2.99 - the versions 2.95 and 2.97 were only made public after public demand and despite our will as the system was still under development.

In this version of the manual, we do not list enhancements and additions but rather changes in APIs and configuration which will affect already existing implementations.

In the CMIS M_Get primitive, zero for the *nattrs* parameter used to denote "all managed object attributes". Though this is still true, you should use now the manifest constant GET_ALLATTRS for all and GET_NOATTRS for no attributes as their values may change in future releases (see M_Get manual page, Section 3). These are currently defined as:

```
#define GET_ALLATTRS    0
#define GET_NOATTRS    -1
```

Some of the MOClassInfo methods used at the MO::initialiseClass method have now a few additional parameters. These are the following:

```
int  setClass (char* className, int nbindings, int nattrs,
               int ngroups, int nevents, int nactions, int npackages = 0);

int  setAttr (char* attrName, int attrId,
               Bool settable = False, Attr* dfltVal = NULLATTR);

int  setGroupAttr (char* groupName, int groupId, int ngroupAttrs);

int  setAction (char* actionName, int actionId, Attr* action);
```

The additional parameters are needed to optimise the use of storage space and in the case of set and action to provide information on set capability, the default value and the action template respectively (see *MOClassInfo* specification, Section 5).

The information on the settability of attributes and the default value are now kept by the managed object class and not by the attribute (see above). This means that the makeSettable and makeUnsettable methods do not longer exist Also, the first two of the following Attr class methods are now deprecated and should be replaced by the last two (see also the *Attr* class specification, Section 4):

```
// deprecated methods, still there but will be removed

void        replace (void* value);
CMISErrors  replace (PE encodedValue);

// they replace the previous two methods

void        replace (void* value);
CMISErrors  set (PE encodedValue);
```

The MO class set and action method interfaces have now slightly changed:

```
virtual CMISErrors  set (CMISModifyOp modifyOperator,
                         int attrId, int classLevel, void* setValue)

virtual CMISErrors  action (int actionId, int actionLevel,
                         void* actionValue, void** actionReply);
```

The set and action value and the action reply are now C data structures instead of ASN.1 presentation elements. Also, in the case of set, the attribute value is set by the GMS, the set method needs only to implement any interactions to the real resource (see also *MO* class specification, Section 5).

The initialiseSyntaxes call is now used to initialise the OSIMIS ETCDIR and load ASN.1 syntaxes, it is imperative that you use this one as the first call in both agents and managers (see manual page, Section 3).

The agent main program has changed, consult the $(TOP)/agent/sma/Sma.cc file and use it after you adopt it to your needs. Also, the Create.cc file in the main agent directory is now in the GMS library and the Create.h should be renamed to <agent>.h e.g. Sma.h. It contains the same information as before i.e. the agent's managed object classes and creation information, and remember that it should have an entry for EVERY managed object class, not only those with initial creation information.

The Log Control systems management function which as introduced in version 2.99 had an "ad-hoc access control mechanism" to prevent access to log records if this was desirable. This was implemented at compile time through a manifest constant in $(TOP)/agent/gms/EventLog.h. In this version, this is a run time option through a flag in $(ETCDIR)/osimistailor, see the $(TOP)/README file for details.

Finally, the OSIMIS ETCDIR oidtable.oc and isobjects files you used to register MIB information are no longer used, you can copy the standard OSIMIS ones. All the MIB information should be now registered in the oidtable.gen and oidtable.at files, in the former anything without associated syntax i.e. class, name bindings, groups, packages and general errors while in the latter information with association syntax i.e. attributes, actions and notifications.

2. Overview

TBA, consult the $(TOP)/README file for the moment.

3. Communication Services

This section will eventually describe in detail the Common Management Information Service API. All the information is currently here but in the form of manual pages instead of a tutorial section.

In reading these, you will need the *mparm.h* and *msap.h* CMIS header files. The former contains all the interface data structures used in the CMIS primitives ("management parameters") while the latter contains all the indication/confirmation related structures that contain the information sent together with prototypes for the primitives ("management service access point").

As background reading and support, you may also consult the following sections of the ISODE User’s manual: Volume 1 Chapter 2 "Association Control", Chapter 3 "Remote Operations" and Chapter 5 "Encoding of Data-Structures". Also relevant are the chapters on the ASN.1 compilers, Volume 4 Chapter 5/6 "POSY/PEPY" and Chapter 7 "PEPSY" and on using the object identifier tables Volume 5 Chapter 17 "Programming the Directory".

NAME

M_InitialiseReq - establish a management association

SYNOPSIS

```
#include <isode/msap.h>

int M_InitialiseReq (callingTitle, callingAddr, calledTitle, calledAddr,
                     context, protvrsn, fununits, access, info, mc, mi)

AEI callingTitle, calledTitle;
char * context;
struct PSAPaddr * callingAddr, * calledAddr;
int protvrsn, fununits;
External * access, * info;
MSAPConnect * mc;
MSAPIndication * mi;
```

DESCRIPTION

M_InitialiseReq() attempts to establish a management association with another management application. It provides a synchronous interface and a successful return is equivalent to the M-ASSOCIATE.CONFIRMATION event. Its arguments are the following:

calledTitle and calledAddr identify the remote application's entity title and presentation address. They may be created from the host and service names using str2aei() and aei2addr(). These involve accessing the OSI directory using the high performance name service or the stub-directory (default), depending on configuration.

callingTitle and callingAddr are the calling application's entity title and presentation address. They may be created in the same way as calledTitle and calledAddr. These parameters are optional, so they may also be left NULLAEI and NULLPA respectively.

context is the application context for the association (i.e. "management").

protvrsn is an integer whose bits indicate the CMIP versions supported, at present it should be 2 for version two (bit 1 set).

fununits is an integer whose bits are intended to indicate the capabilities of the caller. The first four bits may be set to indicate the following capabilities:

Bit	Functional Unit
0	multipleObjectSelection
1	filter
2	multipleReply
3	extendedService

access is an application defined parameter used for access control. It is a pointer to a struct type_UNIV_EXTERNAL as in <isode/pepy/UNIV-types.h> (type-defined as External). It may be created using the routine external_build() (see manual entry) and free'd using external_free(). This parameter is optional: NULLMACCESS or NULLEXTERN may be used if there are no access control requirements.

info is user information to be sent with the association request. It is an external type (see access) and NULLEXTERN may be used if no information is to be sent.

mc is a pointer to a MSAPConnect structure which is updated if the association establishment is successful and contains information about the association, including the association descriptor mc->mc_sd to be used in all references to it.

mi is a pointer to a MSAPIndication structure which is updated only if the association establishment procedure fails and contains in mi->mi_abort the reason for the failure - in particular mi->mi_abort.ma_data is a human readable reason.

DIAGNOSTICS

OK is returned if the association establishment is successful, NOTOK otherwise.

SEE ALSO

ISODE User's Manual Vol.1 Chapter 2 (Association Control), str2aei(), aei2addr(), external_build(), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_Init - initialise a management responder

SYNOPSIS

```
#include <isode/msap.h>

int M_Init (vecp, vec, ms, mi)

int vecp;
char ** vec;
MSAPStart * ms;
MSAPIndication * mi;
```

DESCRIPTION

M_Init() initialises a management responder. It should be called by a dynamic responder as soon as it invoked and by a static responder every time a request for a new association arrives. A successful return is equivalent to the M-ASSOCIATE.INDICATION event. Its arguments are the following:

vecp and vec, in the case of a dynamic responder, should be the argv and argc program arguments, while in the case of a static responder they are transparently passed by the isodeserver() event dispatcher.

ms is a pointer to a MSAPStart structure which is updated only if the call succeeds and contains association information, including the association descriptor ms->ms_sd to be used in all references to it.

mi is a pointer to a MSAPIndication structure which is updated only if the association establishment procedure fails and contains in mi->mi_abort the reason for the failure - in particular mi->mi_abort.ma_data is a human readable reason.

DIAGNOSTICS

OK is returned upon success, NOTOK otherwise.

SEE ALSO

ISODE User’s Manual Vol.1 Chapter 2 (Association Control), isodeserver(), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_InitialiseResp - respond to a management association request

SYNOPSIS

```
#include <isode/msap.h>

int M_InitialiseResp (ms, status, respTitle, respAddr, context, protvrsn,
                      fununits, access, info, mi)

MSAPStart * ms;
int status;
AEI respTitle;
struct PSAPaddr * respAddr;
char* context;
int protvrsn, fununits;
External * access, * info;
MSAPIndication * mi;
```

DESCRIPTION

M_InitialiseResp() responds to a management association request. A successful return is equivalent to the M-ASSOCIATE.RESPONSE event. Its arguments are the following:

ms is a pointer to the MSAPStart structure filled previously by the M_Init() primitive.

status denotes if the association request is accepted or rejected. ACS_ACCEPT should be used to accept the association request, otherwise ACS_TRANSIENT or ACS_PERMANENT should be used for a transient or permanent rejection respectively.

respTitle and respAddr are the application's entity title and presentation address. They may be created from the host and service names using str2aei() and aei2addr(). These involve accessing the OSI directory using the high performance name service or the stub-directory (default), depending on configuration. These parameters are optional, so they may also be left NULLAEI and NULLPA respectively.

context is the application context for the association. It may be left NULLCP if the requester's application context is acceptable.

protvrsn is an integer whose bits indicate the CMIP versions supported, at present it should be 2 for version two (bit 1 set).

fununits is an integer whose bits are intended to indicate the capabilities of the caller. The first four bits may be set to indicate the following capabilities:

Bit	Functional Unit
0	multipleObjectSelection
1	filter
2	multipleReply
3	extendedService

access is an application defined parameter used for access control. It is a pointer to a struct type_UNIV_EXTERNAL as in <isode/pepy/UNIV-types.h> (type-defined as External). It may be created using the routine external_build() (see manual entry) and free'd using external_free(). This parameter is optional: NULLMACCESS or NULLEXTERN may be used if there are no access control requirements.

info is user information to be sent with the association response. It is an external type (see access) and NULLEXTERN may be used if no information is to be sent.

mi is a pointer to a MSAPIndication structure which is updated only if the call fails and contains in mi->mi_abort the reason for the failure - in particular mi->mi_abort.ma_data is a human readable reason.

DIAGNOSTICS

OK is returned upon success, NOTOK otherwise.

SEE ALSO

ISODE User’s Manual Vol.1 Chapter 2 (Association Control), M_InitialiseReq(), M_Init(), str2aei(), aei2addr(), external_build(), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_TerminateReq - orderly management association release request

SYNOPSIS

```
#include <isode/msap.h>

int M_TerminateReq (msd, reason, info, mi)

int msd, reason;
PE info;
MSAPIndication * mi;
```

DESCRIPTION

M_TerminateReq() requests the orderly release of a management association. It provides a synchronous interface and a successful return is equivalent to the M-RELEASE.CONFIRMATION event. Its arguments are the following:

msd is the association descriptor.

reason is the reason for the association release request. It should be one of ACF_NORMAL, ACF_URGENT, ACF_UNDEFINED.

info is user information to be sent with the association release request. This is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using procedures produced by the posy/pepy or pepsy ASN.1 compilers.

mi is a pointer to a MSAPIndication structure which is updated only if the association release procedure fails and contains in mi->mi_abort the reason for the failure - in particular mi->mi_abort.ma_data is a human readable reason.

DIAGNOSTICS

OK is returned if the association has been successfully released, NOTOK otherwise.

SEE ALSO

ISODE User’s Manual Vol.1 Chapter 2 (Association Control), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_AbortReq - request the abrupt release of a management association

SYNOPSIS

```
#include <isode/msap.h>

int M_AbortReq (msd, info, mi)

int msd;
External * info;
MSAPIndication * mi;
```

DESCRIPTION

M_AbortReq() requests the abrupt release of a management association. A successful return it is equivalent to a M-ABORT.REQUEST event and the association is immediately released with any data queued possibly lost. Its arguments are the following:

msd is the association descriptor.

info is user information to be sent with the association abort request. This is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using procedures produced by the posy/pepy or pepsy ASN.1 compilers.

mi is a pointer to a MSAPIndication structure which is updated only if the call fails and contains in mi->mi_abort the reason for the failure - in particular mi->mi_abort.ma_data is a human readable reason.

DIAGNOSTICS

OK is returned upon success, NOTOK otherwise.

SEE ALSO

ISODE User's Manual Vol.1 Chapter 2 (Association Control), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

m_initialise, m_terminate - set up and release a management association.

SYNOPSIS

```
#include <isode/msap.h>

int m_initialise(agent, host)

char *agent, *host;

int m_terminate(msd)

int msd;
```

DESCRIPTION

m_initialise() attempts to set up an association to an agent application.

agent expresses the remote logical management application "qualifier". In OSIMIS there are two such application qualifiers, SMA for the ISO Transport and the example UNIX MIB agent and OIM-SMA for the OSI Internet MIB agent (OSI view of the SNMP MIB-II for TCP/IP, as described in RFC 1214).

host specifies the name of the host where the application runs - the application "designator". It should be the name with which the host has been registered in the isoentities database or the global Directory. Note that for Internet hosts, this is usually the last component of its address e.g. lemma and NOT lemma.cs.ucl.ac.uk.

If a connection is established then m_initialise() returns a positive integer which is a descriptor for the new management association.

m_terminate() will attempt a "graceful release" of the association to which msd is the descriptor.

RETURN VALUES

Each functions return OK on success and NOTOK on failure.

FILES

$(ETC)/isoentities - PSAP address information of agents.

DIAGNOSTICS

Should be obvious.

SEE ALSO

M_InitialiseReq(3N), M_TerminateReq(3N)

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS, and the RACE projects NEMESYS and ICM.

NAME

M_Get - Get Management Information

SYNOPSIS

```
#include <isode/msap.h>

int M_Get (msd, invoke, obj_class, obj_inst, scope, filter, access, sync, nattrs, attrs, mi)

int msd, invoke;
MID obj_class;
MN obj_inst;
CMISScope * scope;
CMISFilter * filter;
External * access;
CMISSync sync;
int nattrs;
MIDentifier attrs[];
MSAPIndication * mi;
```

DESCRIPTION

M_Get is a remote operation request to retrieve management information. It is always directed from an application in a managing role to one in an agent role and is (obviously) a confirmed service: a result or error is expected. Upon successful return, it is equivalent to a M-GET.REQUEST event.

The call returns as soon as the APDU is queued, it does not wait for the result/error. M_WaitReq() must be used to wait for the latter i.e. an asynchronous remote operations interface is used. The arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the parameters to the get operation:

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an object identifier or an integer (global or local ID respectively). Usually the object class is an OID, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid().

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn() assuming that naming attributes have been registered in oidtable.at.

scope may be used to select a set of managed objects to perform the management operation. It is a pointer to a CMISScope structure which contains two integers: scope->sc_type, which may take one of the values Sc_BaseObject, Sc_FirstLevel, Sc_WholeSubtree, Sc_IndividualLevel or Sc_BaseToNthLevel and scope->sc_level, which should indicate the selected level if any. This parameter is optional: NULLMSCOPE may be used if scoping is not required.

filter is the CMIS Filter to be applied to the scoped managed objects. Only the ones for which the filtering expression evaluates to true will be selected for the operation. Consult <isode/mparm.h> and IS 9596 (CMIP). This parameter is optional: NULLMFILTER may be used if filtering is not required.

access is an application defined parameter used for access control. It is a pointer to a struct type_UNIV_EXTERNAL as in <isode/pepy/UNIV-types.h> (type-defined as External). It may be created using the routine external_build() (see manual entry) and free'd using external_free(). This parameter is optional: NULLMACCESS may be used if there are no access control requirements.

sync defines the synchronisation requirements for the management operations across more than one selected managed objects. The value s_atomic may be used if all operations should succeed or none should be performed. The value s_bestEffort may be used for an attempt on a best effort basis. Syncronisation is only meaningful when scoping is used. NULLMSYNC (effectively s_bestEffort) may be used if there are no synchronisation requirements.

attrs are the identifiers for the management attributes to be retrieved. It is an array of MIDentifier structures (see above), while nattrs is the number of elements in that array. By specifying nattrs to be GET_ALLATTRS or GET_NOATTRS all or no attributes are requested respectively. In this case the attrs argument is irrelevant. Attribute identifiers are usually object identifiers, in which case attrs[x].mid_type should be set to MID_GLOBAL and attrs[x].mid_global should contain the actual OID. The latter may be created from the "dot notation" using str2oid() or from the attribute textual description registered in oidtable.at, using name2oid().

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), external_build(), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_GetRes - Get Management Information Result

SYNOPSIS

```
#include <isode/msap.h>

int M_GetRes (msd, invoke, linked, obj_class, obj_inst, cur_time, nattrs, attrs,
                    error, error_info, mi)

int msd, invoke, linked;
MID obj_class;
MN obj_inst;
char* cur_time;
int nattrs;
CMISGetAttr attrs[];
CMISErrors error;
CMISErrorInfo* error_info;
MSAPIndication* mi;
```

DESCRIPTION

M_GetRes is the result to the M_Get remote operation request, returning management information. It is always directed from an application in an agent role to one in a managing role. Upon successful return, it is equivalent to a M-GET.RESPONSE event. Its arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation. In the case of a single reply or the very last (non-linked) of a series of linked replies, invoke should have the same value as the operation request invoke ID. In the case of a linked reply, it may have any value.

linked denotes if the reply is linked. In the case of a single reply or the very last (non-linked) of a series of linked replies, linked should be NONLINKED (zero) while in the case of a linked reply, it should have the same value as the operation request invoke ID.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the get result/error parameters. In the case of an error other than m_getListError, only the error and error_info parameters should be passed as explained later while the rest may be left NULL<X>. A m_getListError, which means that some of the requested attributes are not returned because of errors, is actually treated as a (partial) result.

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an OID or an integer (global or local ID respectively). Usually the object class is an object identifier, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid(). This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMID may be used.

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at. This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMN may be used.

cur_time is the current time at which the response is generated. It is a character string representation of the ASN.1 GeneralizedTime and may be created using gent2str() and tm2ut() (the latter converts between a UNIX tm time structure and the ISODE UTCtime, which should be passed to the former). This parameter is optional and may be omitted by using NULLCP.

attrs are the retrieved management attributes. It is an array of CMISGetAttr structures, while nattrs is the number of elements in that array.
attrs[x].ga_id is the attribute identifier as received in M_Get().
attrs[x].ga_error is the error occurred while trying to access that attribute and may have one of the values: m_accessDenied if access was denied for security reasons, m_noSuchAttribute if the managed object does not contain the requested attribute or m_noError if there was no error. In the latter case, attrs[x].ga_val should contain the actual attribute value. This is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using procedures produced by the posy/pepy or pepsy ASN.1 compilers (NULLPE may be used in the case of an error). In the case of a general get request error other than m_getListError, nattrs and attrs may be left 0 and NULLMGETATTR respectively.

error is the error occurred during the M_Get() operation. The possible error codes and their semantics are:

m_noError - no error
m_noSuchObjectClass - the specified class was not recognised
m_noSuchObjectInstance - the specified object instance was not recognised
m_accessDenied - the specified managed object could not be accessed because of security reasons
m_invalidScope - the specified scope parameter was invalid
m_invalidFilter - the specified filter parameter was invalid
m_syncNotSupported - the specified (atomic) synchronisation is not supported
m_classInstanceConflict - the specified instance does not belong to the specified class
m_complexityLimitation - one of the specified scope, filter or sync parameters was too complex
m_getListError - some of the attributes could not be accessed because of errors
m_operationCancelled - the operation was cancelled by a M-CANCEL-GET
m_processingFailure - a general failure occurred while processing the operation (usually out of memory)

According to the error encountered, error_info, which points to a CMISErrorInfo structure, should contain the following information.

m_noError, m_getListError, m_accessDenied, m_operationCancelled:
no error information is returned and error_info may be NULLMERRORPARM.

m_noSuchObjectClass:
error_info -> ei_noSuchObjectClass should contain the object class.

m_noSuchObjectInstance:
error_info -> ei_noSuchObjectInstance should contain the object instance.

m_invalidScope: error_info -> ei_invalidScope should contain the scope parameter.

m_invalidFilter:
error_info -> ei_invalidFilter should contain the filter parameter.

m_syncNotSupported:
error_info -> ei_syncNotSupported should contain the sync parameter.

m_classInstanceConflict:
error_info -> ei_classInstanceConflict.cic_class should contain the object class and
error_info -> ei_classInstanceConflict.cic_inst should contain the instance.

m_complexityLimitation:
error_info -> ei_complexityLimitation.cl_scope may contain the scope (optional),
error_info -> ei_complexityLimitation.cl_filter may contain the filter (optional) and
error_info -> ei_complexityLimitation.cl_sync may contain the sync parameter (also optional).

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

m_processingFailure:
error_info -> ei_processingFailure.pf_class should contain the object class,
error_info -> ei_processingFailure.pf_inst may contain the object instance (optional),
error_info -> ei_processingFailure.pf_error.mp_id should contain the error identifier and
error_info -> ei_processingFailure.pf_error.mp_val should contain the error information.

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

A series of linked replies should be terminated by an empty non-linked reply with invoke equal to the request invocation identifier:

M_GetRes (msd, invoke, NONLINKED, NULLMID, NULLMN, NULLCP, 0, NULLMGETATTR, NULLMERROR, NULLMERRORINFO, mi);

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_Set/M_SetConf - Set Management Information

SYNOPSIS

#include <isode/msap.h>

int M_Set (msd, invoke, obj_class, obj_inst, scope, filter, access, sync, nattrs, attrs, mi)

int msd, invoke;
MID obj_class;
MN obj_inst;
CMISScope* scope;
CMISFilter* filter;
External* access;
CMISSync sync;
int nattrs;
CMISSetAttr attrs[];
MSAPIndication* mi;

int M_SetConf (msd, invoke, obj_class, obj_inst, scope, filter, access, sync, nattrs, attrs, mi) /* parameters as above */

DESCRIPTION

M_Set/M_SetConf are remote operation requests to modify management information. They are always directed from an application in a managing role to one in an agent role. M_Set is an unconfirmed service while M_SetConf is a confirmed one: a result or error is expected. Upon successful return, they are equivalent to a M-SET.REQUEST event.

The M_SetConf call returns as soon as the APDU is queued, it does not wait for the result/error. M_WaitReq() must be used to wait for the latter i.e. an asynchronous remote operations interface is used. The arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the parameters of the set operation:

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an object identifier or an integer (global or local ID respectively). Usually the object class is an OID, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from the "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid().

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at.

scope may be used to select a set of managed objects to perform the management operation. It is a pointer to a CMISScope structure which contains two integers: scope->sc_type, which may take one of the values Sc_BaseObject, Sc_FirstLevel, Sc_WholeSubtree, Sc_IndividualLevel or Sc_BaseToNthLevel and scope->sc_level, which should indicate the selected level if any. This parameter is optional: NULLMSCOPE

may be used if scoping is not required.

filter is the CMIS Filter to be applied to the scoped managed objects. Only the ones for which the filtering expression evaluates to true will be selected for the operation. Consult <isode/mparm.h> and IS 9596 (CMIP). This parameter is optional: NULLMFILTER may be used if filtering is not required.

access is an application defined parameter used for access control. It is a pointer to a struct type_UNIV_EXTERNAL as in <isode/pepy/UNIV-types.h> (type-defined as External). It may be created using the routine external_build() (see manual entry) and free'd using external_free(). This parameter is optional: NULLMACCESS may be used if there are no access control requirements.

sync defines the synchronisation requirements for the management operations across more than one selected managed objects. The value s_atomic may be used if all operations should succeed or none should be performed. The value s_bestEffort may be used for an attempt on a best effort basis. Syncronisation is only meaningful when scoping is used. NULLMSYNC (effectively s_bestEffort) may be used if there are no synchronisation requirements.

attrs are the management attributes to be set. It is an array of CMISSetAttr structures, while nattrs is the number of elements in that array.
attrs[x].sa_modify is the modify operator to be applied to the particular attribute. It may take one of the values m_replace, m_setToDefault, m_addValue or m_removeValue, the last two applying only to set-valued attributes.
attrs[x].sa_id is a MIDentifier structure (see above) identifying the attribute. Attribute identifiers are usually object identifiers, in which case attrs[x].sa_id.mid_type should be set to MID_GLOBAL and attrs[x].sa_id.mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the attribute textual description registered in oidtable.at, using name2oid().
attrs[x].sa_val contains the actual attribute value to be set. It is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using procedures produced by the posy/pepy or pepsy ASN.1 compilers. The attribute value is optional if the modify operator is m_setToDefault, in which case NULLPE may be used.

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), external_build(), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_SetRes - Set Management Information Result

SYNOPSIS

```
#include <isode/msap.h>

int M_SetRes (msd, invoke, linked, obj_class, obj_inst, cur_time, nattrs, attrs,
                error, error_info, mi)

int msd, invoke, linked;
MID obj_class;
MN obj_inst;
char* cur_time;
int nattrs;
CMISSetAttr* attrs;
CMISErrors error;
CMISErrorInfo* error_info;
MSAPIndication* mi;
```

DESCRIPTION

M_SetRes is the result to the M_SetConf remote operation request. It is always directed from an application in an agent role to one in a managing role. Upon successful return, it is equivalent to a M-SET.RESPONSE event. Its arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation. In the case of a single reply or the very last (non-linked) of a series of linked replies, invoke should have the same value as the operation request invoke ID. In the case of a linked reply, it may have any value.

linked denotes if the reply is linked. In the case of a single reply or the very last (non-linked) of a series of linked replies, linked should be NONLINKED (zero) while in the case of a linked reply, it should have the same value as the operation request invoke ID.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the set result/error parameters. In the case of an error other than m_setListError, only the error and error_info parameters should be passed as explained later while the rest may be left NULL<X>. A m_setListError, which means that some of the requested attributes were not set because of errors, is actually treated as a (partial) result.

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an object identifier or an integer (global or local ID respectively). Usually the object class is an OID, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid(). This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMID may be used.

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at. This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMN may be used.

cur_time is the current time at which the response is generated. It is a character string representation of the ASN.1 GeneralizedTime and may be created using gent2str() and tm2ut() (the latter converts between a UNIX tm time structure and the ISODE UTCtime, which should be passed to the former). This parameter is optional and may be omitted by using NULLCP.

attrs are the new values of the management attributes that were set. It is an array of CMISSetAttr structures, while nattrs is the number of elements in that array.
attrs[x].sa_id is the attribute identifier as received in M_SetConf.
attrs[x].ga_error is the error occurred while trying to set that attribute and may have one of the values:
m_accessDenied if access was denied for security reasons,
m_noSuchAttribute if the managed object does not contain the attribute
m_invalidAttributeValue if the specified attribute value was invalid
m_invalidOperation if the modify operator specified can not be performed on the specified attribute
m_invalidOperator if the modify operator specified is not recognised and
m_noError if there was no error.
attrs[x].ga_val is the new value of the attribute if it was set correctly or the supplied value in M_SetConf in the case of an error. The attribute value is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using procedures produced by the posy/pepy or pepsy ASN.1 compilers.
attrs[x].ga_modify contains the unrecognised modify operator in the case of a m_invalidOperator error, otherwise it may be m_noModifyOp.

In the case of a general set request error other than m_setListError, nattrs and attrs may be left 0 and NULLMSETATTR respectively.

error is the error occurred during the M_SetConf operation. The possible error codes and their semantics are:

m_noError - no error
m_noSuchObjectClass - the specified class was not recognised
m_noSuchObjectInstance - the specified object instance was not recognised
m_accessDenied - the specified managed object could not be accessed because of security reasons
m_invalidScope - the specified scope parameter was invalid
m_invalidFilter - the specified filter parameter was invalid
m_syncNotSupported - the specified (atomic) synchronisation is not supported
m_classInstanceConflict - the specified instance does not belong to the specified class
m_complexityLimitation - one of the specified scope, filter or sync parameters was too complex
m_setListError - some of the attributes could not be accessed because of errors
m_processingFailure - a general failure occurred while processing the operation (usually out of memory)

According to the error encountered, error_info, which points to a CMISErrorInfo structure, should contain the following information.

m_noError, m_setListError, m_accessDenied:
no error information is returned and error_info may be NULLMERRORPARM.

m_noSuchObjectClass:
error_info -> ei_noSuchObjectClass should contain the object class.

m_noSuchObjectInstance:
error_info -> ei_noSuchObjectInstance should contain the object instance.

m_invalidScope:
error_info -> ei_invalidScope should contain the scope parameter.

m_invalidFilter:
error_info -> ei_invalidFilter should contain the filter parameter.

m_syncNotSupported:
error_info -> ei_syncNotSupported should contain the sync parameter.

m_classInstanceConflict:
error_info -> ei_classInstanceConflict.cic_class should contain the object class and
error_info -> ei_classInstanceConflict.cic_inst should contain the instance.

m_complexityLimitation:
error_info -> ei_complexityLimitation.cl_scope may contain the scope (optional),
error_info -> ei_complexityLimitation.cl_filter may contain the filter (optional) and
error_info -> ei_complexityLimitation.cl_sync may contain
the sync parameter (also optional).

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

m_processingFailure:
error_info -> ei_processingFailure.pf_class should contain the object class,
error_info -> ei_processingFailure.pf_inst may contain the object instance (optional),
error_info -> ei_processingFailure.pf_error.mp_id should contain the error identifier and
error_info -> ei_processingFailure.pf_error.mp_val should contain the error information.

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

A series of linked replies should be terminated by an empty non-linked reply with invoke equal to the request invocation identifier:
M_SetRes (msd, invoke, NONLINKED, NULLMID, NULLMN, NULLCP, 0, NULLMSETATTR, NULLMERROR, NULLMERRORINFO, mi);

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User’s Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_Action/M_ActionConf - Perform a Management Action

SYNOPSIS

```
#include <isode/msap.h>

int M_Action (msd, invoke, obj_class, obj_inst, scope, filter, access, sync,
                    action_type, action_info, mi)

int msd, invoke;
MID obj_class;
MN obj_inst;
CMISScope* scope;
CMISFilter* filter;
External* access;
CMISSync sync;
MID action_type;
PE action_info;
MSAPIndication* mi;

int M_ActionConf (msd, invoke, obj_class, obj_inst, scope, filter, access, sync, action_type, action_info,
mi) /* parameters as above */
```

DESCRIPTION

M_Action/M_ActionConf are remote operation requests to perform a management action. They are always directed from an application in a managing role to one in an agent role. M_Action is an unconfirmed service while M_ActionConf is a confirmed one: a result or error is expected. Upon successful return, they are equivalent to a M-ACTION.REQUEST event.

The M_ActionConf call returns as soon as the APDU is queued, it does not wait for the result/error. M_WaitReq() must be used to wait for the latter i.e. an asynchronous remote operations interface is used. The arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the parameters of the action operation:

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an object identifier or an integer (global or local ID respectively). Usually the object class is an OID, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from the "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid().

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at.

scope may be used to select a set of managed objects to perform the management operation. It is a pointer to a CMISScope structure which contains two integers: scope->sc_type, which may take one of the values Sc_BaseObject, Sc_FirstLevel, Sc_WholeSubtree, Sc_IndividualLevel or Sc_BaseToNthLevel and scope-

>sc_level, which should indicate the selected level if any. This parameter is optional: NULLMSCOPE may be used if scoping is not required.

filter is the CMIS Filter to be applied to the scoped managed objects. Only the ones for which the filtering expression evaluates to true will be selected for the operation. Consult <isode/mparm.h> and IS 9596 (CMIP). This parameter is optional: NULLMFILTER may be used if filtering is not required.

access is an application defined parameter used for access control. It is a pointer to a struct type_UNIV_EXTERNAL as in <isode/pepy/UNIV-types.h> (type-defined as External). It may be created using the routine external_build() (see manual entry) and free'd using external_free(). This parameter is optional: NULLMACCESS may be used if there are no access control requirements.

sync defines the synchronisation requirements for the management operations across more than one selected managed objects. The value s_atomic may be used if all operations should succeed or none should be performed. The value s_bestEffort may be used for an attempt on a best effort basis. Syncronisation is only meaningful when scoping is used. NULLMSYNC (effectively s_bestEffort) may be used if there are no synchronisation requirements.

action_type is the type of the management action. It is a pointer to a MIDentifier structure (see above). The action type is usually an object identifier, in which case action_type->mid_type should be set to MID_GLOBAL and action_type->mid_global should contain the actual OID. The latter may be created from the "dot notation" using str2oid() or from the action textual description registered in oidtable.at, using name2oid().

action_info contains the action information. It is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using the posy/pepy or pepsy ASN.1 compilers. This parameter is optional: NULLPE should be used if there is no action information.

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), external_build(), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_ActionRes - Management Action Result

SYNOPSIS

```
#include <isode/msap.h>

int M_ActionRes (msd, invoke, linked, obj_class, obj_inst, cur_time, action_reply,
                 error, error_info, mi)

int msd, invoke, linked;
MID obj_class;
MN obj_inst;
char* cur_time;
CMISParam* action_reply;
CMISErrors error;
CMISErrorInfo* error_info;
MSAPIndication* mi;
```

DESCRIPTION

M_ActionRes is the result to the M_ActionConf remote operation request. It is always directed from an application in an agent role to one in a managing role. Upon successful return, it is equivalent to a M-ACTION.RESPONSE event. Its arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation. In the case of a single reply or the very last (non-linked) of a series of linked replies, invoke should have the same value as the operation request invoke ID. In the case of a linked reply, it may have any value.

linked denotes if the reply is linked. In the case of a single reply or the very last (non-linked) of a series of linked replies, linked should be NONLINKED (zero) while in the case of a linked reply, it should have the same value as the operation request invoke ID.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the action result/error parameters. In the case of a non-linked reply error or a m_processingFailure linked reply one, only the error and error_info parameters should be supplied: the rest may be NULL<X>. In the case though of a linked reply error other than m_processingFailure, the obj_class and obj_inst parameters should be supplied as well as the error and error_info ones. This is despite the fact that there may be duplication of information in the case of a m_noSuchAction or m_noSuchArgument error as the object class should be also supplied in the error information.

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an object identifier or an integer (global or local ID respectively). Usually the object class is an OID, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid(). This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMID may be used.

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at. This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMN may be used.

cur_time is the current time at which the response is generated. It is a character string representation of the ASN.1 GeneralizedTime and may be created using gent2str() and tm2ut() (the latter converts between a UNIX tm time structure and the ISODE UTCtime, which should be passed to the former). This parameter is optional and may be omitted by using NULLCP.

action_reply contains the reply information. It is a pointer to a CMISParam structure which contains an identifier/value pair.
action_reply->mp_id is the action identifier as received in M_ActionConf.
action_reply->mp_val is the action reply information. This is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using procedures produced by the posy/pepy or pepsy ASN.1 compilers.

In the case of an action error, action_reply may be NULLMPARM.

error is the error occurred during the M_ActionConf operation. The possible error codes and their semantics are:

m_noError - no error
m_noSuchObjectClass - the specified class was not recognised
m_noSuchObjectInstance - the specified object instance was not recognised
m_accessDenied - the specified managed object could not be accessed because of security reasons
m_invalidScope - the specified scope parameter was invalid
m_invalidFilter - the specified filter parameter was invalid
m_syncNotSupported - the specified (atomic) synchronisation is not supported
m_classInstanceConflict - the specified instance does not belong to the specified class
m_complexityLimitation - one of the specified scope, filter or sync parameters was too complex
m_noSuchAction - the specified action type is not supported by this object class
m_noSuchArgument - the specified action information was not recognised
m_invalidArgumentValue - the specified action information is invalid
m_processingFailure - a general failure occurred while processing the operation (usually out of memory)

According to the error encountered, error_info, which points to a CMISErrorInfo structure, should contain the following information.

m_noError:
no error information is returned and error_info may be NULLMERRORPARM.

m_noSuchObjectClass:
error_info -> ei_noSuchObjectClass should contain the object class.

m_noSuchObjectInstance:
error_info -> ei_noSuchObjectInstance should contain the object instance.

m_accessDenied:
if the reply is not linked, no error information is returned and error_info may be NULLMERRORPARM. If though the reply is linked
error_info -> ei_linkedActionAccessDenied should contain the action type.

m_invalidScope:
error_info -> ei_invalidScope should contain the scope parameter.

m_invalidFilter:
error_info -> ei_invalidFilter should contain the filter parameter.

m_syncNotSupported:
error_info -> ei_syncNotSupported should contain the sync parameter.

m_classInstanceConflict:
error_info -> ei_classInstanceConflict.cic_class should contain the object class and
error_info -> ei_classInstanceConflict.cic_inst should contain the instance.

m_complexityLimitation:
error_info -> ei_complexityLimitation.cl_scope may contain the scope (optional),
error_info -> ei_complexityLimitation.cl_filter may contain the filter (optional) and
error_info -> ei_complexityLimitation.cl_sync may contain the sync parameter (also optional).

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

m_noSuchAction:
error_info -> ei_noSuchAction.nsa_class should contain the object class
error_info -> ei_noSuchAction.nsa_type should contain the action type.

m_noSuchArgument:
error_info -> ei_noSuchArgument.nsa_class may contain the object class (optional),
error_info -> ei_noSuchArgument.nsa_type should contain the action type.

m_invalidArgumentValue:
error_info -> ei_invalidArgumentValue.iav_id should contain the action type,
error_info -> ei_invalidArgumentValue.iav_val may contain the action information (optional).

m_processingFailure:
error_info -> ei_processingFailure.pf_class should contain the object class,
error_info -> ei_processingFailure.pf_inst may contain the object instance (optional),
error_info -> ei_processingFailure.pf_error.mp_id should contain the error identifier and
error_info -> ei_processingFailure.pf_error.mp_val should contain the error information.

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

A series of linked replies should be terminated by an empty non-linked reply with invoke equal to the request invocation identifier: M_ActionRes (msd, invoke, NONLINKED, NULLMID, NULLMN, NULLCP, NULLMPARM, NULLMERROR, NULLMERRORINFO, mi);

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User’s Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_Create - Create a Managed Object

SYNOPSIS

```
#include <isode/msap.h>

int M_Create (msd, invoke, obj_class, obj_inst, instance_type, reference_inst,
        access, nattrs, attrs, mi)

int msd, invoke;
MID obj_class;
MN obj_inst, reference_inst;
int instance_type;
External* access; int nattrs;
CMISParam attrs[];
MSAPIndication* mi;
```

DESCRIPTION

M_Create is a remote operation request to create a managed object i.e. to enable the management of the associated real resource. It is always directed from an application in a managing role to one in an agent role and it is a confirmed service: a result or error is expected. Upon successful return, it is equivalent to a M-CREATE.REQUEST event.

The call returns as soon as the APDU is queued, it does not wait for the result/error. M_WaitReq() must be used to wait for the latter i.e. an asynchronous remote operations interface is used. The arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the parameters to the create operation:

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an object identifier or an integer (global or local ID respectively). Usually the object class is an OID, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid().

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at. This parameter is optional as the remote agent may use its own name for managed objects from particular object classes. In this case NULLMN should be used.

instance_type is the type of the object instance. It may take one of the values CA_OBJECT_INST, specifying the name of the actual object to be created or CA_PARENT_INST, specifying the name of the parent object under which the object will be created (in the latter case, the remote agent will assign the relative object's name). If the obj_inst is not specified, NULL may be used for the inst_type.

reference_inst is a managed object instance of the same class as the object to be created, which may be used to determine initial attribute values, depending on the managed object class specification. It may be created in the same way as the obj_inst and it is an optional parameter, in which case NULLMN should be

used.

access is an application defined parameter used for access control. It is a pointer to a struct type_UNIV_EXTERNAL as in <isode/pepy/UNIV-types.h> (type-defined as External). It may be created using the routine external_build() (see manual entry) and free'd using external_free(). This parameter is optional: NULLMACCESS may be used if there are no access control requirements.

attrs are the attribute identifiers/values for the management attributes to be initialised. It is an array of CMISParam structures, while nattrs is the number of elements in that array.
attrs[x].mp_id is a MIDentifier structure (see above) identifying the attribute. Attribute identifiers are usually object identifiers, in which case attrs[x].mp_id.mid_type should be set to MID_GLOBAL and attrs[x].mp_id.mid_global should contain the actual OID. The latter may be created from the "dot notation" using str2oid() or from the attribute textual description registered in oidtable.at, using name2oid().
attrs[x].mp_val contains the initial attribute value which is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using procedures produced by the posy/pepy or pepsy ASN.1 compilers.

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), external_build(), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_CreateRes - Create Managed Object Result

SYNOPSIS

```
#include <isode/msap.h>

int M_CreateRes (msd, invoke, obj_class, obj_inst, cur_time, nattrs, attrs,
                 error, error_info, mi)

int msd, invoke;
MID obj_class;
MN obj_inst;
char* cur_time;
int nattrs;
CMISParam* attrs;
CMISErrors error;
CMISErrorInfo* error_info;
MSAPIndication* mi;
```

DESCRIPTION

M_CreateRes is the result to the M_Create remote operation request, It is always directed from an application in an agent role to one in a managing role. Upon successful return, it is equivalent to a M-CREATE.RESPONSE event. Its arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the create result/error parameters. In the case of an error, only the error and error_info parameters should be passed: the rest may be left NULL<X>.

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an OID or an integer (global or local ID respectively). Usually the object class is an object identifier, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid(). This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMID may be used.

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at. This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMN may be used.

cur_time is the current time at which the response is generated. It is a character string representation of the ASN.1 GeneralizedTime and may be created using gent2str() and tm2ut() (the latter converts between a UNIX tm time structure and the ISODE UTCtime, which should be passed to the former). This parameter is optional and may be omitted by using NULLCP.

attrs are the initial values of all the management attributes. It is an array of CMISParam structures (identifier/value pairs), while nattrs is the number of elements in that array. attrs[x].mp_id is the attribute identifier and attrs[x].ga_val is the initial attribute value. This is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using

procedures produced by the posy/pepy or pepsy ASN.1 compilers.
In the case of a create request error, nattrs and attrs may be left 0 and NULLMPARM respectively.

error is the error occurred during the M_Create operation. The possible error codes and their semantics are:

m_noError - no error
m_noSuchObjectClass - the specified class was not recognised
m_duplicateManagedObjectInstance - the specified managed object already exists
m_noSuchReferenceObject - the specified reference object does not exist
m_accessDenied - the specified managed object could not be created
m_noSuchAttribute - a specified attribute was not recognised
m_noSuchObjectInstance - the specified parent object instance was not recognised
m_classInstanceConflict - the specified instance may not be created as a member of the specified class
m_invalidObjectInstance - the specified object instance violates the naming rules
m_missingAttributeValue - a required attribute value was not specified
m_invalidAttributeValue - a specified attribute value was invalid
m_processingFailure - a general failure occurred while processing the operation (usually out of memory)

According to the error encountered, error_info, which points to a CMISErrorInfo structure, should contain the following information.

m_noError, m_accessDenied:
no error information is returned and error_info may be NULLMERRORPARM.

m_noSuchObjectClass:
error_info -> ei_noSuchObjectClass should contain the object class.

m_duplicateManagedObjectInstance:
error_info -> ei_duplicateObjectInstance should contain the object instance.

m_noSuchReferenceObject:
error_info -> ei_noSuchReferenceObject should contain the reference object.

m_noSuchAttribute:
error_info -> ei_noSuchAttribute should contain the unrecognised attribute ID.

m_noSuchObjectInstance:
error_info -> ei_noSuchObjectInstance should contain the object instance.

m_classInstanceConflict:
error_info -> ei_classInstanceConflict.cic_class should contain the object class and
error_info -> ei_classInstanceConflict.cic_inst should contain the instance.

m_invalidObjectInstance:
error_info -> ei_invalidObjectInstance should contain the object instance.

m_missingAttributeValue:
error_info -> ei_missingAttributeValue.mav_attrs (an array of MIDentifier structures) should contain attribute identifiers for which initial values were required but not supplied and
error_info -> ei_missingAttributeValue.mav_nattrs should be the number of the attribute identifiers in that array.

m_invalidAttributeValue:
error_info -> ei_invalidAttributeValue.mp_id should contain the attribute identifier and
error_info -> ei_invalidAttributeValue.mp_val should contain the invalid attribute value.

m_processingFailure:
error_info -> ei_processingFailure.pf_class should contain the object class,
error_info -> ei_processingFailure.pf_inst may contain the object instance (optional),
error_info -> ei_processingFailure.pf_error.mp_id should contain the error identifier and
error_info -> ei_processingFailure.pf_error.mp_val should contain the error information.

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_Delete - Delete Managed Object(s)

SYNOPSIS

```
#include <isode/msap.h>

int M_Delete (msd, invoke, obj_class, obj_inst, scope, filter, access, sync, mi)

int msd, invoke;
MID obj_class;
MN obj_inst;
CMISScope* scope;
CMISFilter* filter;
External* access;
CMISSync sync;
MSAPIndication* mi;
```

DESCRIPTION

M_Delete is a remote operation request to delete one or more managed objects i.e. to disable the management of the associated real resource(s). It is always directed from an application in a managing role to one in an agent role and it is a confirmed service: a result or error is expected. Upon successful return, it is equivalent to a M-DELETE.REQUEST event.

The call returns as soon as the APDU is queued, it does not wait for the result/error. M_WaitReq() must be used to wait for the latter i.e. an asynchronous remote operations interface is used. The arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails (APDU not queued).

The remaining arguments are the parameters to the create operation:

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an object identifier or an integer (global or local ID respectively). Usually the object class is an OID, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid().

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at.

scope may be used to select a set of managed objects to perform the management operation. It is a pointer to a CMISScope structure which contains two integers: scope->sc_type, which may take one of the values Sc_BaseObject, Sc_FirstLevel, Sc_WholeSubtree, Sc_IndividualLevel or Sc_BaseToNthLevel and scope->sc_level, which should indicate the selected level if any. This parameter is optional: NULLMSCOPE may be used if scoping is not required.

filter is the CMIS Filter to be applied to the scoped managed objects. Only the ones for which the filtering expression evaluates to true will be selected for the operation. Consult <isode/mparm.h> and IS 9596 (CMIP). This parameter is optional: NULLMFILTER may be used if filtering is not required.

access is an application defined parameter used for access control. It is a pointer to a struct type_UNIV_EXTERNAL as in <isode/pepy/UNIV-types.h> (type-defined as External). It may be created using the routine external_build() (see manual entry) and free'd using external_free(). This parameter is optional: NULLMACCESS may be used if there are no access control requirements.

sync defines the synchronisation requirements for the management operations across more than one selected managed objects. The value s_atomic may be used if all operations should succeed or none should be performed. The value s_bestEffort may be used for an attempt on a best effort basis. Syncronisation is only meaningful when scoping is used. NULLMSYNC (effectively s_bestEffort) may be used if there are no synchronisation requirements.

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), Vol. 5 Chapter 17 (Programming the Directory), external_build(), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_DeleteRes - Delete Managed Object Result

SYNOPSIS

#include <isode/msap.h>

int M_DeleteRes (msd, invoke, linked, obj_class, obj_inst, cur_time, error, error_info, mi)

int msd, invoke, linked;
MID obj_class;
MN obj_inst;
char* cur_time;
CMISErrors error;
CMISErrorInfo* error_info;
MSAPIndication* mi;

DESCRIPTION

M_DeleteRes is the result to the M_Delete remote operation request. It is always directed from an application in an agent role to one in a managing role. Upon successful return, it is equivalent to a M-DELETE.RESPONSE event. Its arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation. In the case of a single reply or the very last (non-linked) of a series of linked replies, invoke should have the same value as the operation request invoke ID. In the case of a linked reply, it may have any value.

linked denotes if the reply is linked. In the case of a single reply or the very last (non-linked) of a series of linked replies, linked should be NONLINKED (zero) while in the case of a linked reply, it should have the same value as the operation request invoke ID.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the delete result/error parameters. In the case of a non-linked reply error, only the error and error_info parameters should be supplied: the rest may be NULL<X>. In the case though of a linked reply error (it may only be m_accessDenied), the obj_class and obj_inst parameters should be supplied as well as the error and error_info ones (actually error_info may be omitted for that error code - see below).

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an OID or an integer (global or local ID respectively). Usually the object class is an object identifier, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid(). This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMID may be used.

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at. This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMN may be used.

cur_time is the current time at which the response is generated. It is a character string representation of the ASN.1 GeneralizedTime and may be created using gent2str() and tm2ut() (the latter converts between a UNIX tm time structure and the ISODE UTCtime, which should be passed to the former). This parameter

is optional and may be omitted by using NULLCP.

error is the error occurred during the M_Delete operation. The possible error codes and their semantics are:

m_noError - no error
m_noSuchObjectClass - the specified class was not recognised
m_noSuchObjectInstance - the specified object instance was not recognised
m_accessDenied - the specified managed object could not be accessed because of security reasons
m_invalidScope - the specified scope parameter was invalid
m_invalidFilter - the specified filter parameter was invalid
m_syncNotSupported - the specified (atomic) synchronisation is not supported
m_classInstanceConflict - the specified instance does not belong to the specified class
m_complexityLimitation - one of the specified scope, filter or sync parameters was too complex
m_processingFailure - a general failure occurred while processing the operation (usually out of memory)

According to the error encountered, error_info, which points to a CMISErrorInfo structure, should contain the following information.

m_noError, m_accessDenied:
no error information is returned and error_info may be NULLMERRORPARM.

m_noSuchObjectClass:
error_info -> ei_noSuchObjectClass should contain the object class.

m_noSuchObjectInstance:
error_info -> ei_noSuchObjectInstance should contain the object instance.

m_invalidScope:
error_info -> ei_invalidScope should contain the scope parameter.

m_invalidFilter: error_info -> ei_invalidFilter should contain the filter parameter.

m_syncNotSupported:
error_info -> ei_syncNotSupported should contain the sync parameter.

m_classInstanceConflict:
error_info -> ei_classInstanceConflict.cic_class should contain the object class and
error_info -> ei_classInstanceConflict.cic_inst should contain the instance.

m_complexityLimitation:
error_info -> ei_complexityLimitation.cl_scope may contain the scope (optional),
error_info -> ei_complexityLimitation.cl_filter may contain the filter (optional) and
error_info -> ei_complexityLimitation.cl_sync may contain the sync parameter (also optional).

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

m_processingFailure:
error_info -> ei_processingFailure.pf_class should contain the object class,
error_info -> ei_processingFailure.pf_inst may contain the object instance (optional),
error_info -> ei_processingFailure.pf_error.mp_id should contain the error identifier and
error_info -> ei_processingFailure.pf_error.mp_val should contain the error information.

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

A series of linked replies should be terminated by an empty non-linked reply with invoke equal to the request invocation identifier:
M_DeleteRes (msd, invoke, NONLINKED, NULLMID, NULLMN, NULLCP, NULLMERROR, NULLMERRORINFO, mi);

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_EventRep/M_EventRepConf - Send an Event Report

SYNOPSIS

```
#include <isode/msap.h>

int M_EventRep (msd, invoke, obj_class, obj_inst, event_type, event_time, event_info, mi)

int msd, invoke;
MID obj_class, event_type;
MN obj_inst;
char* event_time;
PE event_info;
MSAPIndication* mi;

int M_EventRepConf (msd, invoke, obj_class, obj_inst, event_type, event_time, event_info, mi) /* parameters as above */
```

DESCRIPTION

M_EventRep/M_EventRepConf are remote operation requests to send an event report. They are always directed from an application in an agent role to one in a managing role. M_EventRep is an unconfirmed service while M_EventRepConf is a confirmed one: a result or error is expected. Upon successful return, they are equivalent to a M-EVENT-REPORT.REQUEST event.

The M_EventRepConf call returns as soon as the APDU is queued, it does not wait for the result/error. M_WaitReq() must be used to wait for the latter i.e. an asynchronous remote operations interface is used. The arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the parameters to the event report operation:

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an object identifier or an integer (global or local ID respectively). Usually the object class is an OID, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid().

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at.

event_type is the type of the event. It is a pointer to a MIDentifier structure (see above). The event type is usually an object identifier, in which case event_type->mid_type should be set to MID_GLOBAL and event_type->mid_global should contain the actual OID. The latter may be created from the "dot notation" using str2oid() or from the event textual description registered in oidtable.at, using name2oid().

event_time is the current time at which the event report is generated. It is a character string representation of the ASN.1 GeneralizedTime and may be created using gent2str() and tm2ut() (the latter converts between a UNIX tm time structure and the ISODE UTCtime, which should be passed to the former). This

parameter is optional and may be omitted by using NULLCP.

event_info contains the actual event information. It is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using the posy/pepy or pepsy ASN.1 compilers. This parameter is optional: NULLPE should be used if there is no event information.

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi->mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_EventRepRes - Event Report Result

SYNOPSIS

```
#include <isode/msap.h>

int M_EventRepRes (msd, invoke, obj_class, obj_inst, cur_time, event_reply,
                   error, error_info, mi)

int msd, invoke;
MID obj_class;
MN obj_inst;
char* cur_time;
CMISParam* event_reply;
CMISErrors error;
CMISErrorInfo* error_info;
MSAPIndication* mi;
```

DESCRIPTION

M_EventRepRes is the result to the M_EventRepConf remote operation request. It is always directed from an application in a managing role to one in an agent role. Upon successful return, it is equivalent to a M-EVENT-REPORT.RESPONSE event. Its arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

The remaining arguments are the event report result/error parameters. In the case of an error, only the error and error_info parameters should be supplied: the rest may be NULL<X>.

obj_class is the managed object class. It is of type MID, a pointer to a MIDentifier structure which may contain an object identifier or an integer (global or local ID respectively). Usually the object class is an OID, in which case obj_class->mid_type should be set to MID_GLOBAL and obj_class->mid_global should contain the actual OID. The latter may be created from a "dot notation" using str2oid() or from the class textual description registered in oidtable.oc, using name2oid(). This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMID may be used.

obj_inst is the managed object instance (i.e. name). It is of type MN, a pointer to a MName structure which may contain a full distinguished name, a subset ("local") distinguished name or a character string ("non-specific" name). Usually the object instance is a distinguished name, in which case obj_inst->mn_type should be set to MN_DN and obj_inst->mn_dn should contain the actual distinguished name. The latter may be created from a string representation using str2dn(), assuming that naming attributes have been registered in oidtable.at. This parameter is optional and may be omitted in the case of a single reply referring to the base object. In this case NULLMN may be used.

cur_time is the current time at which the response is generated. It is a character string representation of the ASN.1 GeneralizedTime and may be created using gent2str() and tm2ut() (the latter converts between a UNIX tm time structure and the ISODE UTCtime, which should be passed to the former). This parameter is optional and may be omitted by using NULLCP.

event_reply contains the reply information. It is a pointer to a CMISParam structure which contains an identifier/value pair.
event_reply->mp_id is the action identifier as received in M_ActionConf.
event_reply->mp_val is the event reply information. This is a presentation element and may be created from the internal representation manually, using the psap library primitives, or automatically, using

procedures produced by the posy/pepy or pepsy ASN.1 compilers. This parameter is optional: NULLPE may be used if there is no event reply information. In the case of an event report error, event_reply may be left NULLMPARM.

error is the error occurred during the M_EventRepConf operation. The possible error codes and their semantics are:
m_noError - no error
m_noSuchObjectClass - the specified class was not recognised
m_noSuchObjectInstance - the specified object instance was not recognised
m_noSuchEventType - the specified event type was not recognised
m_noSuchArgument - the specified event information was not recognised
m_invalidArgumentValue - the specified event information is invalid
m_processingFailure - a general failure occurred while processing the operation (usually out of memory)

According to the error encountered, error_info, which points to a CMISErrorInfo structure, should contain the following information.

m_noError:
no error information is returned and error_info may be NULLMERRORPARM.

m_noSuchObjectClass:
error_info -> ei_noSuchObjectClass should contain the object class.

m_noSuchObjectInstance:
error_info -> ei_noSuchObjectInstance should contain the object instance.

m_noSuchEventType:
error_info -> ei_noSuchEventType.nsa_class should contain the object class
error_info -> ei_noSuchEventType.nsa_type should contain the action type.

m_noSuchArgument:
error_info -> ei_noSuchArgument.nsa_class may contain the object class (optional),
error_info -> ei_noSuchArgument.nsa_type should contain the event type.

m_invalidArgumentValue:
error_info -> ei_invalidArgumentValue.iav_id should contain the event type,
error_info -> ei_invalidArgumentValue.iav_val may contain the event information (optional).

m_processingFailure:
error_info -> ei_processingFailure.pf_class should contain the object class,
error_info -> ei_processingFailure.pf_inst may contain the object instance (optional),
error_info -> ei_processingFailure.pf_error.mp_id should contain the error identifier and
error_info -> ei_processingFailure.pf_error.mp_val should contain the error information.

The whole error parameter is optional, so error_info may be also NULLMERRORPARM.

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User’s Manual Vol. 1 Chapter 3 (Remote Operations), Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), Vol. 5 Chapter 17 (Programming the Directory), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_CancelGet - Cancel a Previously Released Get Request

SYNOPSIS

#include <isode/msap.h>

int M_CancelGet (msd, invoke, get_invoke, mi)

int msd, invoke, get_invoke;
MSAPIndication* mi;

DESCRIPTION

M_CancelGet is a remote operation request to cancel a previously released Get request. It is always directed from an application in a managing role to one in an agent role and it is a confirmed service: a result or error is expected. Upon successful return, it is equivalent to a M-CANCEL-GET.REQUEST event.

The call returns as soon as the APDU is queued, it does not wait for the result/error. M_WaitReq() must be used to wait for the latter i.e. an asynchronous remote operations interface. The arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

get_invoke is the invocation identifier of the operation to be cancelled.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User's Manual Vol. 1 Chapter 3 (Remote Operations), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

M_CancelGetRes - Cancel Get Result

SYNOPSIS

#include <isode/msap.h>

int M_CancelGetRes (msd, invoke, error, error_info, mi)

int msd, invoke;
CMISErrors error;
CMISErrorInfo* error_info;
MSAPIndication* mi;

DESCRIPTION

M_CancelGetRes is the result to the M_CancelGet remote operation request. is always directed from an application in an agent role to one in a managing role. Upon successful return, it is equivalent to a M-CANCEL-GET.RESPONSE event. Its arguments are the following:

msd is the management association descriptor.

invoke is the invocation identifier for the operation.

mi is a pointer to a MSAPIndication structure, which is updated only if the call fails i.e. the APDU is not queued.

error is the error occurred during the M_CancelGet() operation. The possible error codes and their semantics are:

m_noError - no error
m_noSuchInvokeId - the get invoke identifier parameter was not recognised.
m_mistypedOperation - the get invoke identifier parameter does not refer to a M-GET operation.
m_processingFailure - a general failure occurred while processing the operation (usually out of memory)

According to the error encountered, error_info, which points to a CMISErrorInfo structure, should contain the following information.

m_noError, m_mistypedOperation, m_processingFailure:
no error information is returned and error_info may be NULLMERRORPARM.

m_noSuchInvokeId:
error_info -> ei_noSuchInvokeId should contain the get invoke identifier.

DIAGNOSTICS

OK is returned upon success, NOTOK upon failure and the mi ->mi_preject structure is updated: mi-> mi_preject.mpr_reason contains the reason for the failure (an integer) and mi->mi_preject.mpr_data contains a human readable string.

SEE ALSO

ISODE User’s Manual Vol. 1 Chapter 3 (Remote Operations), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

initialiseSyntaxes - Set up ISODE syntax information and tailoring

SYNOPSIS

```
#include <isode/msap.h>

extern char *etcdir;
extern char *tailor;
typedef void (*VoidFunc)();

int initialiseSyntaxes(tailorName, syntaxFuncs)

char *tailorName;
VoidFunc *syntaxFuncs;
```

DESCRIPTION

initialiseSyntaxes() must be the first function ISODE function call made in a program. It has three functions :

The ETC directory

ISODE libraries use configuration files and tables that are usually located in a well known location. This is called the ETC directory. InitialiseSyntaxes() reads the path name for this directory as from the value of OSIMISETCPATH environment variable.

Tailoring

ISODE uses a tailor file to configure certain aspects of the ISODE stack, as well as provide other configuration information to the application. The tailor file must be located in the ETC directory for OSIMIS applications. If the value of tailorName is non-NULL, then initialiseSyntaxes() set the value of the global variable tailor to the full path name of the tailor file (for use by other parts of the program). If it is NULL then the default tailor file name "osimistailor" is used.

Syntax information

The ETC directory should contain two files that are used to configure the syntax information. These files are "oidtable.gen" and "oidtable.at". syntaxFuncs should point to a NULL terminated list of function pointers. These functions are called to set up the correct syntax information for use by the program. The value of syntaxFuncs may be one of the standrd OSIMIS ones (see <isode/msap/Syntax.h>) or it may be user-defined. The functions are basically calls to the dsap library routine add_attribute_syntax();

RETURN VALUES

OK on success
NOTOK on failure

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the correct OSIMIS ETC directory before starting this program.

FILES

$(ETC)/oidtable.[at,gen] - syntax and OID information.
$(ETC)/osimistailor - the default tailor file.

DIAGNOSTICS

Should be obvious.

NOTES

See the ISODE User’s Manual for more information on tailor files and also the dsap library routines used.

AUTHOR

Saleem N. Bhatti, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

ava2str, str2ava, attrv2str - convert between strings and values for syntaxes.

SYNOPSIS

```
#include <isode/msap.h>

int ava2str(oid, pe, name, value)

OID oid;
PE pe;
char **name, **value;

Ava *str2ava(nameStr, valueStr);

char *nameStr, *valueStr;

void free_ava(ava)

Ava *ava;

char *attrv2str(v, print_fnx);

void *v;
PRINT_FNX print_fnx;
```

DESCRIPTION

ava2str() takes an object identifier (oid) and a BER encoded value (pe) and fills in the human readable representations of both at the pointers name and value, respectively. The human readable form of the oid is a name (e.g. of name of an attribute), and the human readable form of the value is as given by the "print function" defined for that attribute syntax. name and value should be the addresses of two char pointers, which ava2str() will give values to. The memory pointed to be name and value after the function returns successfully should be freed by the user (using free(3V)) after use.

str2ava() takes two strings and returns a pointer to an Ava:

```
typedef struct          /* attribute value assertion */ {
    OID ava_oid;
    PE  ava_value;

} Ava;
```

the file $(ETC)/oidtable.at, and valueStr should be parseable by the syntax "parse function" defined for that entry. After use, ava should be freed using free_ava();

attrv2str() is similar to ava2str, except it takes a pointer to the C structure, v, that is an instance of a type and a pointer to the "print function", print_fnx, defined for that type in the syntax tables. The returned string should be freed() after (using free(3V)).

RETURN VALUES

ava2str() returns OK on success and sets the values of the name and value. On failure it returns NOTOK.

str2ava() returns NULL on failure.

attrv2str() returns NULL on failure.

FILES

$(ETC)/oidtable.[at,gen] - syntax and OID information.
$(ETC)/osimistailor - the default tailor file.

DIAGNOSTICS

str2ava() has many obvious diagnostics when compiled with the DEBUG option.

ava2str() and attrv2str() have no diagnostics.

NOTES

See the ISODE User’s Manual for more information on syntax function definitions.

AUTHORS

George Pavlou & Saleem N. Bhatti, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

sprintmfilter, str2mfilter, sprintscope, str2scope, sprintsync, str2sync - various parse and print functions for the MSAP library.

SYNOPSIS

```
#include <isode/msap.h>

char *sprintmfilter(mfilter)

CMISFilter *mfilter;

CMISFilter *str2mfilter(filterStr)

char *filterStr;

char *sprintscope(scope)

CMISScope *scope;

CMISScope *str2scope(scopeStr)

char *scopeStr;

char *sprintsync(sync)

CMISSync sync;

CMISSync *str2sync(syncStr)

char *syncStr;
```

DESCRIPTION

sprintmfilter() returns a pointer to a string that is a human readable representation of mfilter. The pointer returned is to a static area of memory that will be overwritten on the next call to the function.

str2mfilter() takes a string representation of a CMISFilter value (the kind returned by sprintmfilter()) and returns a pointer to a CMISFilter structure. The user must free the memory pointed to after use (using mfilter_free(3N)).

The format of the string representation of the filters is described in **FILTER EXPRESSIONS**.

sprintscope() returns a pointer to a string that is a human readable representation of scope. The pointer returned is to a static area of memory that will be overwritten on the next call to the function.

str2scope takes a string representation of a CMISScope value (the kind returned by sprintscope()) and returns a pointer to a CMISScope structure. The user must free the memory pointed to after use (using free(3V)). Acceptable values for scopeStr are:

baseObject

firstLevel

wholeSubtree

baseTo<n>[st | nd | rd | th]Level

where <n> is a positive integer., e.g.

baseTo1stLevel or baseTo4thLevel

sprintsync() returns a pointer to a string that is a human readable representation of sync. The pointer returned is to a static area of memory that will be overwritten on the next call to the function.

str2sync takes a string representation of a CMISSync value (the kind returned by sprintsync()) and returns a CMISSync value. Acceptable values for syncStr are:

atomic

bestEffort

FILTER EXPRESSIONS

A filter expression is used to construct a CMISFilter value. A CMISFilter contains attribute value assertions (AVAs) that are grouped with the logical operators AND, OR and NOT. The filter expression syntax is as follows :

(<cmisfilter>)

where <cmisfilter> is one of <notfilter>, <andfilter>, <orfilter> or <filteritem>.

The characters used to represent the logical operators are :

Character	**Operator**
"!"	NOT
"&"	AND
"|"	OR

A <notfilter> has the form :

(!(<cmisfilter>))

A <andfilter> has the form :

((<cmisfilter>) & (<cmisfilter>) ...)

A <orfilter> has the form :

((<cmisFilter>) | (<cmisfilter>) ...)

A <filteritem> has one of the two forms :

(<attributename>)

for creating a CMISFilter item with the assertion test for "present", or

(<attributename> <assertiontype> <attributevalue>)

for the other assertion types :

Character	**Assertion type**
"="	equality
":="	substrings
">="	greater or equal
"<="	less or equal
":<"	subset of
":>"	superset of
"><"	non-null intersection

With the substrings operator, the character "*" can be used as a wild card.

Some examples of filter expressions :

"((objectClass = eventRecord) & (eventType = linkUpEvent))"

"((objectClass = log) & (!(administrativeState = unlocked)))"

"((objectClass = log) & ((logId <= 2) | (logId >= 10)))"

"((wiseSaying := *hello*) | (!(wiseSaying = bye)))"

A "NULL" filter (one that always evaluates to true) can be created using :

"(NULL)"

(This is actually an empty AND filter.)

The use of the brackets, "(" and ")", is very important, as they are used to delimit the strings used to represent the components of the filter. Also, please enclose your filter expression in quote marks, as in the examples above, so that the your UNIX shell does not interpret special characters such as "!", "|", "&", ">", "<", "(" and ")".

PLEASE NOTE that for an attribute type to be used in a filter expression, there should be a "parse" function defined for its syntax and the function should be registered in the syntax tables. This parse function reads the attribute's value from a "pretty-printed" form and converts it to a value, i.e. a C structure. There is no methodology applied in OSIMIS (as yet) to the way in which values are "pretty-printed", however a loose convention is :

"scalar" values are represented as single strings e.g.,

```
logId = 1                    /* INTEGER */
objectClass = eventRecord    /* OID */
wiseSaying = Hello World     /* Strings */
administrativeState = unlocked   /* ENUMERATED */
```

"set" values are enclosed in curly brackets. e.g.,

```
availabilityStatus :< {inTest offLine offDuty}
nUsersThld = {Low:7 Switch:On High:10 Switch:On}
```

FILES

$(ETC)/oidtable.[at,gen] - syntax and OID information.

NOTES

The abstract syntax of a CMISFilter, CMISScope and CMISSync is given in *ISO 9596 : "Information Technology - Open Systems Interconnection - Common Management Information Protocol specification"*.

SEE ALSO

ava2str(3N), str2ava(3N)

DIAGNOSTICS

sprintmfilter() and str2mfilter() produce some fairly obvious diagnostics when compiled with the DEBUG flag. Both return NULL when passed invalid arguments.

sprintscope() returns the string "invalidScope" for a bad value of scope. str2scope returns NULL if scopeStr has an invalid value.

sprintsync() returns the string "invalidSync" for a bad value of sync. str2sync() returns s_invalid if syncStr is invalid.

BUGS

The filter expression parser is a bit shaky! You must have the correct number of brackets, as it keeps a count! For instance the filter expression below is missing the outermost set of brackets :

"(eventType = linkUpEvent) | (eventType = linkDownEvent)"

but instead of reading it as a badly formed OR filter, the parser reads it as the <filteritem> :

"(eventType = linkUpEvent)"

Additionally, superfluous brackets, e.g :

((eventType = linkUpEvent))

will cause it to fail.

AUTHORS

George Pavlou & Saleem N. Bhatti, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

external_build - Build an External structure
external_free - Free an External structure

SYNOPSIS

#include <isode/msap.h>

External * external_build (type, context, context_id, pe, octet_aligned)

int type;
char * context;
int context_id;
PE pe;
struct qbuf * octet_aligned;

void external_free (external)
External * external

DESCRIPTION

external_build() builds an External structure and returns a pointer to it. The External structure implements the ASN.1 type EXTERNAL and is the same as the struct type_UNIV_EXTERNAL defined in <isode/pepy/UNIV-types.h>.

type should be one of EXTERN_ASN1_TYPE, EXTERN_ARBITRARY or EXTERN_OCTET_ALIGNED. In the case of the first two, the actual external parameter is a presentation element (pe) while in the case of EXTERN_OCTET_ALIGNED the actual parameter is of type struct qbuf* (octet_aligned). If the type is EXTERN_ASN1_TYPE, the pe parameter may be created from the internal representation manually, using the psap library primitives, or automatically, using procedures produced by the posy/pepy or pepsy ASN.1 compilers.

context identifies the module that defines the syntax of the external data type (ASN.1 or other) and is an object identifier in dot notation e.g. "1.2.3.4". The context is negotiated at association establishment time in conjunction with an associated integer for future references. This integer may be used as the context_id to avoid carrying the context object identifier in each instance of the external type. In the latter case, NULLOID may be used for the context.

external_free() may be used to free the space previously allocated by external_build(). It is actually the same as the ISODE free_UNIV_EXTERNAL().

DIAGNOSTICS

external_build() returns an External* upon success, NULLEXTERN otherwise.

SEE ALSO

<isode/pepy/UNIV-types.h>, ISODE User's Manual Vol. 1 Chapter 5 (Encoding of Data Structures), Vol. 4 Chapters 5/6, 7 (POSY/PEPY, PEPSY), IS 9595/6 (CMIS/P).

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

mfree - free the management structures

SYNOPSIS

```
#include <isode/msap.h>

void mc_free (mc)
MSAPConnect * mc;

void ms_free (ms)
MSAPStart * ms;

void ma_free (ma)
MSAPAbort * ma;

void mi_free (mi)
MSAPIndication * mi;

void mid_free (mid)
MID mid;

void mn_free (mn)
MN mn;

void mparm_free (mparm)
CMISParam * mparm;

void mfilterdata_free (mfilter)
CMISFilter * mfilter;

void mfilter_free (mfilter)
CMISFilter * mfilter;
```

DESCRIPTION

These routines free any data that may have been allocated to the pointed structure. They do not free the structures themselves, apart from mfilter_free().

DIAGNOSTICS

None.

SEE ALSO

ISODE User’s Manual Section 3.

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

4. General Management System Support

This section contains information on general management support which may be common to both agents and managers, though most OSIMIS managers use do not use this infrastructure (yet). This support is two fold:

- support for writing asynchronous event-driven applications
- support for transparent ASN.1 handling

The former is of paramount importance to agents or hybrid-units and also essential to managers that need to receive asynchronous events from more than one sources e.g. many agents or an agent and a user interface etc. The latter enables to program management applications by dealing with the programming language data types instead of abstract structures representing ASN.1 constructs (presentation elements in ISODE, XOM objects in X/Open's XOM ASN.1 API etc.)

The relevant C++ classes in OSIMIS that implement such abstractions are the following:

i. the Knowledge Source class

ii. the Coordinator and ISODE Coordinator classes and

iii. the Attribute and Any Type classes

The Coordinator/Knowledge Source abstraction provides the capability of asynchronous event-driven management while the Attribute/Any Type abstraction supports transparent ASN.1 handling. In fact these abstractions are not specific to management but could be used by any other OSI applications. These are actually implemented in OSIMIS by the *kernel*(3n) library. The following two sections provide detailed information on the class APIs.

4.1 Support for Asynchronous Event-Driven Applications

Class KS

Inherits from:	None
Classes used:	Coordinator
Interface file:	GenericKS.h
Implementation file:	GenericKS.cc

Introduction

The Knowledge Source class is an abstraction of a general management application object, implementing some of the application's management intelligence. The term has its roots in blackboard systems, the blackboard in this case being the local MIB in agents, one or more remote MIBs in managers and the union of those in hybrid applications.

Knowledge sources have the capability of scheduling wake-ups at regular real time intervals in order to implement polling strategies. They may also register external communication endpoints to other management applications, real resources etc. on which they expect to receive management information; they are subsequently notified when information arrives. This is possible through the Coordinator object (see relevant specification) which implements a fully event-driven paradigm for management applications and whose presence and operation is completely transparent.

Knowledge sources may be used in both agent and manager applications. In manager applications they may be (some of) the objects that implement the management intelligence. In agent applications, they may be used to receive information from loosely coupled resources or to implement polling access to real resources in general, either for implementing cache-ahead schemes or in order to support notifications.

In agent applications there should always exist a special knowledge source which is the management protocol agent (CMIS/P, SNMP or other). This is distinguished by the Coordinator and treated specially.

Methods

```
class KS
{
    static KS*  _agent;

protected:
    // real time wake-up capabilities

    int         scheduleWakeUp (long, char*);
    int         scheduleWakeUps (long, char*, Bool = False);
    int         cancelWakeUps (char*);

    // listening on external communication endpoints

    int         startListen(int);
    int         stopListen(int);

    // for the special management agent

    static  int setAgent (KS*);

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

    virtual int wakeUp (char*);
    virtual int readCommEndpoint (int);
    virtual int shutdown (int);

    // ...
};
```

Real time wake-ups

The following set of methods allow to schedule and cancel wake-ups in real time.

int scheduleWakeUps *(long period, char* token, Bool onlyIfReporting);*

This method schedules wake-ups to take place every *period* seconds. The token should be unique character string that will distinguish the associated call-backs by any others possibly requested by the same knowledge source. NULLCP may be supplied if this distinction is not needed or if only one series of wake-ups is scheduled.

If the onlyIfReporting parameter is True, the knowledge source will be awaken only if there are event forwarding disriminators or log managed objects in the system. This is useful when the wake-ups are only used to support event notifications via polling. The default case (parameter not supplied) is wake-ups independently of the notification function. OK is returned upon success, NOTOK if the scheduling period is invalid i.e. less or equal to zero.

int scheduleWakeUp *(long period, char* token);*

This is the same as above but only one wake-up is scheduled after *period* seconds. This will always take place independently of the notification function.

int cancelWakeUps *(char* token);*

This will cancel either multiple or single wake-ups identified by *token*. OK is returned upon success, NOTOK if the cancel operation failed i.e. there was no such scheduled wake-up.

Listening to external communication endpoints

int startListen *(int fd);*

This method allows to register an external point of communication (a Unix file descriptor) on which to start listening for information. OK is returned upon success, NOTOK otherwise (if *fd* < 0 or already registered).

int stopListen *(int fd);*

This allows to deregister an external point of communication so that no more listening on that takes place. OK is returned upon success, NOTOK if the operation failed i.e. there was no such file descriptor registered.

Callbacks

The following two methods are call-backs for wake-ups and notifications regarding data at an external communication endpoint. The third notifies that the application is shutting down. They should be supplied in derived classes.

virtual int wakeUp *(char* token);*

This is called as a result of the wake-up methods. The *token* is the one supplied in the scheduling call. The method should return OK or NOTOK.

virtual int readCommEndpoint *(int fd);*

This is called if there is data at the external communication endpoint identified by the *fd* file descriptor. The method should return OK or NOTOK.

virtual int shutdown *(int fd);*

This is called to notify that the management application is shutting down (exiting). It should be used to release external links gracefully, delete control managed objects in remote agents etc. It should return OK or NOTOK.

The special management agent knowledge source (if present) receives calls for any file descriptors it has registered that describe management associations for CMIS and the UDP transport port for SNMP. It also receives last a call with *fd = -1* as an indication to release its MIB which may terminate interaction to real resources or subordinate agents.

Class Coordinator

Inherits from:	None
Classes used:	KS, List
Interface file:	Coordinator.h
Implementation file:	Coordinator.cc

Introduction

The Coordinator class coordinates activity in management applications by acting as a central point for all external communication endpoints and scheduled alarms. Only one instance of this or any derived classes is present in a management application realised as one UNIX process.

It uses the UNIX select facility, enhanced in ISODE as xselect to be portable across UNIX platforms, to implement a fully event driven scheme with respect to all external communications. A first-come-first-served policy is exercised with respect to both external communications and scheduled real-time alarms (wake-ups).

This class is written in such a way to allow integration with other packages that have their own coordinating mechanisms. This will be necessary for managing applications with graphical user interfaces that receive asynchronous events both from the keyboard and the network e.g. event reports, asynchronous replies to requests etc. It may also be desirable for integrating OSIMIS applications with distributed platforms such as DCE, ANSAware etc. In case integration with another package is needed, the coordinating mechanism of that package is used centrally and a special OSIMIS coordinator (a derived class) should be written to work with the latter.

Methods

```
class Coordinator
{
    static Coordinator* _instance;
    // ...

public:
    // interface to other coordinating mechanisms

    void        setAlarmMode (Bool);
    long        getNextWakeUpIntvl ();
    void        getCommEndpointMask (fd_set*, int*);
    int         serviceWakeUp ();        // alarm handler (in serviceAlarm)
    void        shutdown (Bool = False); // terminate handler (in terminate)

    virtual int commEndpointMaskChange (int, int);
    virtual int nextWakeUpIntvlChange (long);


    // changing the communication endpoint listening mechanism

    virtual int readCommEndpoints ();


    // initialisation methods

    static void serviceAlarm (int);
    static void terminate (int);
    void        listen ();

    // ...
};
```

Interface to other coordinating mechanisms

The following set of methods allow to use the coordinator together with the coordinating mechanisms of other packages in central role. in real time.

void setAlarmMode *(Bool mode);*

When the OSIMIS coordinator is in central role, it handles alarms through the UNIX alarm system call. When created, the coordinator thinks by default that it is in central role but it can "be told" that this is not the case through this method i.e. setAlarmMode(False). In this case, it still manages the list of wake-ups for all the knowledge sources but without scheduling the next alarm through alarm(3).

long getNextWakeUpIntvl *();*

This allows another coordinating mechanism to get the interval for the next OSIMIS wake-up. This should be done once after the OSIMIS initialisation has finished and after every OSIMIS related wake-up (see serviceWakeUp).

void getCommEndpointMask *(fd_set* mask, int* nfds);*

This allows another coordinating mechanism to get the mask of OSIMIS external communication endpoints (file descriptors). *mask* is the descriptor mask and *nfds* the number of the highest file descriptor plus one which; these are updated through this call.

int serviceWakeUp *();*

This allows another coordinating mechanism to tell the OSIMIS coordinator to service a particular real-time wake-up. This call should be always combined with getNextWakeUpIntvl immediately afterwards to find out the period for the next OSIMIS wake-up. That other mechanism should keep track if the next alarm for the whole system is an OSIMIS one or not.

This method is also essentially the alarm handler when there is no other coordinating mechanism. It needs though to be wrapped-up in the static **void serviceAlarm** *(int);* method because UNIX expects signal handlers to have a single integer argument - in non-static methods the hidden *this* pointer is added.

void shutdown **(Bool doExit);**

This allows another coordinating mechanism to tell the OSIMIS coordinator that the whole system is exiting so that knowledge sources are notified and other OSIMIS cleaning-up is done. In this case the *doExit* argument should be False so that the exit(3) call is not executed.

This method is also essentially the termination handler when there is no other coordinating mechanism, in which case *doExit* is True. It needs though to be wrapped-up in the static method **void terminate** *(int);* because UNIX expects signal handlers to have a single integer argument - in non-static methods the hidden *this* pointer is added.

virtual int commEndpointMaskChange *(int fd, int mode);*

This allows another coordinating mechanism to redefine it in a derived class so that it arranges to take care of any change in the OSIMIS mask during program execution. *fd* is the file descriptor that changed and *mode* will be *MASK_ADD* for adding and *MASK_REMOVE* for removing it from the global mask.

virtual int nextWakeUpIntvlChange *(long interval);*

This allows another coordinating mechanism to redefine it in a derived class so that it arranges to take care of any change in the OSIMIS next wake-up interval during program execution. *interval* is the new next wake-up interval.

Changing the communication endpoint listening mechanism

virtual int readCommEndpoints *();*

This method enables to redefine the mechanism through which the event-driven model with respect to external communication endpoints is exercised. The default mechanism is ISODE's enhancement of select(2) *xselect*. Redefining this may be needed for packages with hidden descriptors, in which case xselect cannot be used. This is exactly the case with ISODE association control through the *iserver_wait* mechanism, see derived class IsodeCoordinator.

Initialisation

The following methods are used only when the OSIMIS coordinating system does coordinate i.e. when it is not integrated with the coordinating mechanism of another package as the master. They should be used to initialise the single coordinator instance and start-up the central listening procedure.

static void serviceAlarm *(int signal);*
static void terminate *(int signal);*

These are the signal handlers, they should be used to catch the UNIX SIGALRM, SIGINT, SIGQUIT and SIGTERM signals. As they are static, they do not need the instance pointer, the latter can be accessed through the private variable _instance through which the serviceWakeUp and shutdown methods are called respectively. They are needed because of the standard format of UNIX signal handlers (see before). The standard way to use them from the main program is:

```
signal(SIGALRM, (SIGHANDLER) Coordinator::serviceAlarm);
signal(SIGINT,  (SIGHANDLER) Coordinator::terminate);
signal(SIGQUIT, (SIGHANDLER) Coordinator::terminate);
signal(SIGTERM, (SIGHANDLER) Coordinator::terminate);
```

void listen *();*

This method realises the central listening process for events. It should be called after all initialisation has finished. It never returns but the shutdown method is called (through terminate) upon receipt of a termination signal.

Class ISODECoordinator

Inherits from:	Coordinator
Classes used:	None
Interface file:	IsodeCoord.h
Implementation file:	IsodeCoord.cc

Introduction

The ISODECoordinator class is a special coordinator for ISODE applications that wish to receive ACSE association requests. Such processes are all management agents, all hybrid applications (both agents and managers) and those managers that may receive management association requests for event reporting.

The reason a special coordinator is needed is that ISODE uses a special mechanism to initialise a process listening for associations (iserver_init) and to listen for incoming association requests or data on existing associations (iserver_wait) - see ISODE User Manual Volume 1 Chapter 2 "Association Control". These hide the "TSAP" descriptor so that explicit use of xselect is not possible. All this class does is to simply redefine the virtual *int readCommEndpoints ()* method to use iserver_wait instead of xselect.

4.2 Support For Transparent ASN.1 Handling

Class Attr

Inherits from: None

Classes used: None

Interface file: GenericAttr.h

Implementation file: GenericAttr.cc

Introduction

The Attr class is a superclass of all management attributes. It contains the actual attribute value as a C language data structure as the ISODE ASN.1 compilers (pepsy or posy/pepy) do not work with C++. It may also contain an ASN.1 presentation element corresponding to that value if the latter has been encoded in order to optimise ASN.1 processing i.e. avoid encoding a value every time it is requested through the management protocol.

This class defines a set of virtual methods which may be redefined in derived classes. Such classes for the generic attribute types i.e. counter, gauge, counter-threshold, gauge-threshold and tide-mark, commonly used data types e.g. strings, integer, real, time etc. and common attribute types e.g. administrative and operational state, destination address etc. are provided by OSIMIS. It is almost certain though that applications introducing new managed objects will need additional attribute types. The documentation of this class together with the general guidelines in the section describing the specific OSIMIS attributes provide the framework for introducing these.

Methods

```
class Attr
{
    // ...

protected:
    // methods related to the ASN.1 syntax - to be provided in derived classes

    virtual PE          _encode ();
    virtual void*       _decode (PE);
    virtual void        _free ();
    virtual char*       _print ();
    virtual void*       _copy ();
    virtual int         _compare (void*);
```

```
public:
    // methods to encode, decode, free, print, copy, compare and store
    // the attribute value

    PE                  encode ();
    void*               decode (PE);
    void                ffree ();
    void                ffree (void*);
    char*               print ();
    void*               copy ();
    int                 compare (void*);
    char*               asnMemoryDump (int*);
    int                 asnFileDump (FILE*);

    // methods to access and modify the contained attribute value

    void*               getval ();
    void                setval (void*);
    void                replval (void*);

    // methods that may be redefined in derived classes to associate behaviour
    // or simply manipulate the value according to syntax (add/remove)

    virtual void*       get ();
    virtual int         set (void*);
    virtual int         setDefault (void*);
    virtual int         add (void*);
    virtual int         remove (void*);

    // methods to respond to CMIS operations

    CMISErrors          get (PE*);
    CMISErrors          set (PE);

    virtual Bool        filter (int, void*);
    virtual Bool        filterSubstring (int, void*, Bool);

    // ...
};
```

Virtual syntax manipulation methods

The following set of methods should be provided in derived classes. They allow the attribute syntax to be manipulated. They all operate on the contained attribute value apart from _decode which decodes a supplied presentation element. They are essentially simple wrap-ups of the procedures used for the syntax tables (see relevant section).

The only ones that are really necessary are _encode, _decode, _free and _print. _compare is not currently used but it may be used in the future to automate filtering while _copy can be emulated through _encode and _decode at a moderate performance cost.

One could avoid providing these methods if the specific attribute type is derived from AnyType which is in turn derived from this class (see AnyType class specification). This would necessitate the use of syntax tables and would also incur a table look-up operation every time such an attribute type is instantiated. This is not generally recommended since providing these methods is very easy and accords to a standard pattern (see below).

virtual PE _encode *();*

This method should encode the contained value which could be accessed through the getval method. It is a simple wrap-up of the encoding procedure for the syntax tables which in turn uses the encoder generated by the ASN.1 compiler. Memory is allocated for the resulting presentation element.

virtual void* _decode *(PE pe);*

This method should just decode its argument and return the result, a pointer to a C data type. It is a simple wrap-up of the decoding procedure for the syntax tables which in turn uses the decoder generated by the ASN.1 compiler. Memory is allocated for the type. NULLVD should be returned upon failure i.e. the presentation element does not correspond to the syntax.

virtual char* _print *();*

This method should print the contained value which may be accessed through the getval method. A common way of printing the value is by using the print procedure for the syntax tables which pretty-prints the value to a presentation stream (see relevant section), in conjunction to the *attrv2str(void*, PRINT_FNX)* msap library procedure. The actual value could be accessed through the getval method. Memory is allocated for the returned string.

virtual void _free *();*

This method should free the contained value which could be accessed through the getval method.

virtual void* _copy *();*

This method should copy the contained value which may be accessed through the getval method. Memory is allocated for the copied value. This method uses currently _encode and _decode to produce a copy but it may not be redefined in derived classes to create a direct copy. The only reason to redefine it would be in order to reduce the processing overhead the default encode/decode approach incurs but this is not essential.

virtual int _compare *(void*);*

This method should compare its argument to the contained value which may be accessed through the getval method. It should return zero if the values are equal, non-zero otherwise. This method is not currently used but in the future it will be used to automate filtering i.e. to

avoid supplying the filter method. This will require an additional argument denoting the comparison mode e.g. equality, subset etc. and a comprehensive of return values. At present there is no reason for providing it.

An implementation example for the integer data type is shown below. Note that the _copy and _compare methods are redefined only to show how this can be done, they are not essential as explained previously. You will notice the need for casting which is because the data type is not known at the this level (a void*).

```
inline PE    Integer::_encode ()
{ return int_enc((int*) getval()); }

inline void* Integer::_decode (PE pe)
{ return int_dec(pe); }

inline void  Integer::_free ()
{ free((char*) getval()); }

inline char* Integer::_print ()
{ return attrv2str(getval(), (PRINT_FNX) int_print); }

inline void* Integer::_copy ()
{ return int_cpy((int*) getval()); }

inline int   Integer::_compare (void* val)
{ return int_cmp((int*) val, (int*) getval()); }
```

Public syntax manipulation methods

The following set of methods use the virtual syntax manipulation methods described above and allow to encode, decode, free, print, copy, compare and store manipulate ASN.1 values. They all return NULL upon failure which means that either the corresponding _<syntaxHandler> method has not been redefined or the attribute value does not map to the attribute syntax. These errors may happen only for undebugged programs apart from mismatches in decoding when the wrong value may come from the network.

PE encode *();*

This method encodes and returns the contained value. The returned presentation element is not a copy and should NOT be free'd. NULLPE is returned upon failure.

void* decode *(PE pe);*

This method decodes its argument and returns the result, a pointer to a C data type. NULLVD is returned upon failure. The decoded value is NOT set in the attribute, the latter is simply used for its syntax knowledge as a decoding engine. The result should be free'd

(using possibly the ffree(void*) method) when no longer needed.

void* print *();*

This method prints the contained value, NULLCP is returned upon failure. The returned string should be free'd using free or delete when no longer needed.

void* copy *();*

This method returns a copy of the contained value. The returned value should be free'd (using possibly the ffree(void*) method) when no longer needed.

int compare *(void* value);*

This method compares its argument to the contained value and returns zero if the values are equal, non-zero otherwise.

char* asnMemoryDump *(int* len);*

This method returns the contained value as an ASN.1/BER stream of octets and updates the length argument.

int asnFileDump *(FILE* outFile);*

This method dumps the contained value as an ASN.1/BER stream of octets to a file. It returns OK if is succeeds, NOTOK otherwise.

Non behaviour-related attribute value access

The following set of methods allow to access and manipulate the contained attribute value without triggering any associated behaviour. For example, a specific attribute type may be redefined to relate to a tightly coupled real resource. In this case, the get method (see below) will be redefined to return the real resource value while getval will only return the value within the attribute, probably corresponding to the last real resource access.

These methods are mainly needed by implementors of derived classes e.g. the virtual syntax manipulation methods use getval to access the attribute value. They are also needed elsewhere, so they need to be public.

void* getval *();*

This method returns the contained (pointer to the) C language data type.

void setval *(void* newValue);*

This method sets the contained value to the supplied one. Memory for the data type supplied should have been previously allocated. Memory for the previous value (if any) is released.

void replval *(void* newValue);*

This method replaces the contained value with the supplied one. By replace is meant that memory for the previous value is NOT released. This is useful when the memory of the data type stored is to be re-used: in that case, the value should be obtained, altered and replaced. This is particularly useful for complex data types where only a particular element needs to be modified.

Possibly behaviour-related attribute value access

The following set of methods allow to access and manipulate the contained attribute value and may be them redefined to associate real-resource behaviour, apart from setDefault which simply uses set.

The set method may be also redefined to perform additional checks or offer a friedlier interface e.g. to allow passing an integer with no allocated memory for the Integer attribute type.

The add and remove methods should be always redefined in derived classes for set-valued types in order to be of any usefulness. This is necessary because there is no knowledge for perfoming the add/remove operation at this level.

The setDefault method needs only to be redefined if special (non real resource) behaviour should be associated to the set-to-default operation e.g. for a CounterThreshold or TideMark the value of the associated Counter or Gauge respectively is needed.

Associating an attribute to a real resource should only be used when the latter is "tightly-coupled" to the agent i.e. shares a common address space. When the real-resource is loosely coupled, this knowledge should be preferably put in the managed object in order to optimise access to the real resource by grouping requests for more than one attribute.

virtual void* get *();*

This method returns the contained (pointer to the) C data type In the case of a tightly-coupled real resource it may be redefined to actually fetch that value. If not, it is equivalent to *getval*.

virtual int set *(void* newValue);*

This method sets the contained value to the supplied one It may be redefined to perform additional checks on the value range or to actually set the value in the case of a tightly-coupled real resource. If not associated to a real resource, it is equivalent to *setval*. It returns OK upon success and NOTOK on failure (invalid value).

virtual int add *(void* addValue);*

This method should always be redefined for a settable set-valued attribute and should simply add its argument to the contained value. It may be also redefined to additionally associate

the attribute to a real resource. It may return NOTOK if the supplied value is invalid or the method has not been redefined.

virtual int remove *(void* addValue);*

This method should always be redefined for a settable set-valued attribute and should simply remove its argument from the contained value. It may be also redefined to additionally associate the attribute to a real resource. It may return NOTOK if the supplied value is invalid, if any of the elements to be removed is not present or the method has not been redefined.

virtual int setDefault *(void*);*

This method simply uses set to set the attribute to the supplied default value. It is mentioned here that the managed object class knows the default value for every settable attribute and this is supplied through this method. There is no point redefining it to associate the attribute to a real resource and this could be done for the set method and serve this one as well. The only reason for redefining it is when an additional value is needed from somewhere else to deduce the default value e.g. for a counter threshold the value of the associated counter is needed etc.

Responding to CMIS requests

CMISErrors get *(PE* encodedValue);*

This is only used by the GMS and calls first the get method of the previous group which may relate the attribute to a real resource. MIB implementors need never use it but is mentioned here for completeness.

CMISErrors set *(PE encodedValue);*

This method is used by the GMS to set an attribute value. It calls the set method of the previous group which may relate the attribute to a real resource.

This method may be also used in a managed object's constructor for initialising an attribute value when the object is created through a CMIS M-CREATE request. It returns m_noError upon success or m_invalidAttributeValue if the value cannot be decoded or it is invalid for the operation (out of range etc.)

virtual Bool filter *(int mode, void* assertedValue);*

This method should be currently supplied for every new attribute type as the compare method is not comprehensive enough to support filtering. The *mode* could have one of the following values:

FI_Equality - equality

FI_GreaterOrEqual - greater than or equal

FI_LessOrEqual - less than or equal

FI_SubsetOf - subset of

FI_SupersetOf - superset of

FI_NonNullSetIntersect - non-null set intersection

The second and third are not applicable to set-valued attributes while the last three are only applicable to those. A typical way of providing this method is to use the *virtual void* get()* method to access the current value which may fetch it from an associated real resource and then perform a comparison with *assertedValue* according to *mode*. True should be returned upon success and False upon failure.

virtual Bool filterSubstring *(int mode, void* assertedValue, Bool first);*

This method needs to be supplied only for string types and OSIMIS will supply these but it is mentioned here for completeness. The *mode* could have one of the following values:

FI_Substring_Initial - initial substring

FI_Substring_Any - any substring

FI_Substring_Final - final substring

The *first* argument is True for the first substring assertion and False for any subsequent ones as this method will be called a number of times, equal to the number of substring assertions. The first time the current value is accessed by using the *virtual void* get()* method which may fetch it from an associated real resource. Then every time a comparison with *assertedValue* according to *mode* is performed and if successful, a static pointer is advanced to the end of the asserted string in the value. This is necessary as the order of the string assertions is important. For example, using the * as a wild-carding character, the assertions

*Its*when*know** and *Its*know*when**

are NOT the same. If an assertion is wrong False should be returned and the method will not be called any further for this filtering operation (logical ANDing of the assertions).

Class AnyType

Inherits from:	Attr
Classes used:	None
Interface file:	GenericAttr.h
Implementation file:	GenericAttr.cc

Introduction

OSIMIS uses parts of the ISODE QUIPU X.500 implementation for manipulating ASN.1 syntaxes through the use of object identifier / syntax tables. These are the oidtable.gen for general object identifiers and the oidtable.at for identifiers with associated syntax i.e. management attributes, actions and notifications. Using those tables results in a large amount of software and data being loaded with management applications and incurs a table look-up every time a new attribute is instantiated. This should be better avoided by agents but could be used by managers where memory and processing requirements are less critical.

The Attr class is designed so that the use of tables can be avoided by explicitly redefining the virtual syntax manipulation methods. The AnyType class is an extension of that class which provides the syntax manipulation methods through the tables. It is designed for use in higher level manager APIs such as the "Remote MIB" or others where only "syntax-related" use of attributes is expected. Though this class could be used as a base class for specific attribute types with the only gain of avoiding to redefine those methods, this is strongly discouraged as it will result in making agents using (MO classes using) those attributes bound to the use of tables.

Methods

```
class AnyType
{
    sntx_table* _syntax;

    // ...

public:
    // constructors

    AnyType (OID, PE);
    AnyType (char*, void*);
}
```

Constructors

This class has constructors which provide information about the syntax in addition to the initial value. This is used to find the syntax entry in the syntax tables which should have already been loaded.

AnyType *(OID attributeType, PE encodedAttributeValue);*

This constructor uses an attribute value assertion (attribute type/value) as the initial information and it is likely to be used when information is coming from the network. *attributeType* is the object identifier for the attribute type as passed in the CMIS API and *encodedAttributeValue* is the ASN.1/BER encoded attribute value, again as needed by the CMIS API.

Trying to build an attribute may fail when the attribute type and value do not match, the syntax is unknown i.e. not registered in the tables, when the value is malformed etc. In this case, a valid pointer will be returned from the create operation when using the *new* operator. The way to check if the attribute has been correctly created is by getting its value using the Attr get or getval methods and check if this is NULLVD. In this case, the attribute should be deleted using the *delete* operator when created using *new*.

Notice that if the call is successful, the value (a presentation element) will be free'd when the attribute will be destroyed, so you should be careful not to free it twice e.g. if it has come through the CMIS API, the pointer to it in the MSAPIndication structure should be set to NULLPE before calling mi_free. If the call has failed, the value is not free'd.

AnyType *(char* suntaxName, void* initialValue);*

This constructor uses the syntax name as registered in the tables e.g. "Integer", "GaugeThreshold" etc. as the initial information. The *initialValue* is a pointer to the value for which space should have been allocated. Trying to build an attribute fail if the syntax is not registered in the tables or the value does not match that syntax (the attribute finds that out by trying to encode it). In this case, the same procedures as for the previous constructor should be followed while the value is left intact (not free'd).

5. The Generic Managed System

This section will eventually describe fully the Generic Managed System (GMS) which is implemented by the *gms*(3n) and *smisntx*(3n) libraries. At present, it contains the following:

i. a tutorial on the implementation of the UNIX managed object which uses the GMS facilities

ii. details on the APIs of the managed object support offered by the GMS i.e. specification of the MOClassInfo, MO, and Top C++ classes

iii. a description of the Structure/Definition of Management Information ASN.1 attributes and syntaxes offered by the GMS

5.1 A Tutorial Introduction

5.1.1 Implementing Managed Object Classes

At present, this section contains a tutorial on the implementation of the UNIX managed object which uses the GMS facilities and is intended to help people who will be implementing managed objects using OSIMIS. In this section such people are referred to either as *GMS users* or *MO implementors*. The first part of this section describes the important aspects of the GMS mainly through the use of tutorial examples. The second part gives a more formal description of the various classes and APIs used. It is assumed that readers are familiar with the OSI management model and C++.

The key task for an MO implementor is to derive a C++ class to represent the new MO. The starting point will be a definition of the MO class - typically in GDMO format in some standard. This will include, amongst other things, the identity of the parent MO class in the inheritance hierarchy and a list of the additional attributes the new class is to contain.

This pattern will be reflected in the C++ class; a new class being derived from a parent class and the additional attributes being added. In many cases these additional attributes will be drawn from the standard types provided by the GMS (gauge, counter etc.). If this is not the case, a new C++ class must be implemented to represent the new attribute type.

If the Management Information Base was like a normal database then that would be the end of the story. Unfortunately it is not; there must also be mechanisms to ensure that changes in the real resource are reflected in attribute values and vice versa. An important aim in the design of the GMS is to ensure that these mechanisms are as flexible as possible. This is in recognition of the fact that few of the resources to be managed will have been built with OSI management in mind. Therefore, the MO implementor has to be given the freedom to use whatever management hooks are available no matter how poorly these fit the OSI management model.

In practice, most of the code written by a GMS user will concern interfacing MOs to real resources. The other MO interface - the one to the agent is provided for free by the GMS through its generic MO class; this is sufficient for most purposes.

5.1.1.1 Deriving New Managed Object Classes

The derivation of C++ classes to represent MO classes will be the normal starting point for any new MIB implementation. All such classes must ultimately be derived from class *Top* which is itself derived from the C++ class *MO*. This section and its successor illustrate the main issues involved by reference to the UxObj1 class the definition of which may be found in the file *$(TOP)/agent/ux_mib/UxMO1.h*.

The UxObj1 MO class does not do anything very useful. It has five mandatory attributes (loosely) related to the Unix operating system:

uxObj1Id - an arbitrary string which identifies an instance of the class i.e. the Relative Distinguished Attribute. Since there is only ever one instance this does not achieve anything much but it has to be present.

sysTime - the current time according to the system clock.

wiseSaying - an arbitrary string which may be set using the OSI management service.

nUsers - a gauge reflecting the current number of logged-on users.

nUsersThld - a gauge threshold related to the above.

There is also one attribute contained in a conditional package:

nUsersTideMark - records the highest value reached by the nUsers attribute.

There is one notification:

nUsersThldExceeded - triggered by the gauge threshold.

Referring to the file above we find the class definition starting with:

```
class UxObj1 : public Top, public KS    // classid {uclManagedObjectClass 50}
{
```

It can be seen that the class UxObj1 is derived directly from two classes; *Top* and *KS*. The class KS (Knowledge Source) is principally concerned with interfacing a set of related MO instances to the Coordinator. When the Coordinator handles timer or socket events it does so by calling a knowledge source method (see Section 4).

The fact that the UxObj1 MO class is one level down from Top in the inheritance hierarchy is recorded at the start of the class definition

```
#define UXOBJ1_LEVEL    1           /* Top 0 */
```

This is used within the definition of the method UxObj1:get(); see Section 5.1.2.2. The heart of the class definition is contained in the lines:

```
//  The attributes are currently identified by integers 0-5

#define UXOBJ1ID        0
#define SYSTIME         1
#define WISESAYING      2
#define NUSERS          3
#define NUSERSTHLD      4
#define NUSERSTIDEMARK  5
#define UXOBJ1NATTRS    6       /* the total number of attributes:
                                   mandatory + the conditional package ones */
#define TMPACKAGENATTRS 1       /* the conditional package number of attrs */

    Attr*       _attrs [UXOBJ1NATTRS];
```

This associates the six attributes of the UxObj1 class (including the one in the conditional package) with a set of integers and declares an array of pointers to the attributes themselves. A key point is that the six attributes declared here are only those which the UxObj1 class has in addition to those inherited from Top. Within the definition of Top you will find five more attributes declared:

```
#define OBJECTCLASS     0
#define NAMEBINDING     1
#define PACKAGES        2
#define ALLOMORPHS      3
#define TOPNATTRS       4

    Attr*       _attrs [TOPNATTRS];
```

At this point, neither the type of the attributes is specified nor are the attributes instantiated. This is done when the UxObj1 class is instantiated - see the description of the constructor UxObj1::UxObj1() in Section 5.1.1.2.

Since the attribute NUSERSTIDEMARK belongs to a conditional package it may, or may not be present. The integer TMPACKAGENATTRS is the number of attributes in the conditional package and Bool _tideMarkPackage; indicates whether the package is present or not. This mechanism is explained further below.

In order that the C++ classes correctly reflect the behaviour of the corresponding MO classes, a certain amount of information has to be available at run time which would normally be available only at compile time. For example, a MO instance needs to know not only what its class is but also what are its superclasses, it must also have access to the mappings between the integer attribute identifiers discussed above and the attribute OIDs from the MIB definition. Rather than replicate this information in every instance of the MO class, a single instance of the class MOClassInfo is created for each MO class instantiated within the system. Notice therefore that near the start of the UxObj1 class definition a static instance of the MOClassInfo class is declared:

```
   static MOClassInfo _uxobj1ClassInfo;//  Object containing MO information
                                       //  relating to all members of this
                                       //  class.
```

This must be present. The static storage class ensures that just one instance of the class will be created. The information in the MOClassInfo object is initialised in the method UxObj1::initialiseClass() which is called from the constructor - see Section 5.1.1.2.

Also, there is a boolean flag indicating whether or not the conditional package for the object is present. This flag is set from an argument that is passed to the constructor of UxObj1 from the create() method as described in Section 5.1.3.4. The mechanism for the use of conditional packages is an interim solution, it is expected to change in future OSIMIS versions.

The remainder of the MO class definition consists mainly of declarations of a few methods particular to the UxObj1 class. These are discussed in Section 5.1.3.1.

5.1.1.2 Instantiation

The tasks which must be performed on instantiation are best illustrated by reference to the UxObj1 class. First, let us look at the definition of the constructor UxObj1::UxObj1() in the file $(TOP)/agent/ux_mib/UxMO1.cc. This carries out five essential tasks:

- if this has not already been done, it calls a routine to initialise the MOClassInfo object (initialiseClass() - also in UxMO1.cc). This routine is written to a standard pattern which should be followed.
- it instantiates the attributes and sets up an array of pointers to them.
- it establishes the association between thresholds and the attributes they monitor
- it registers the class. This is essential to ensure that inherited attributes are handled correctly.
- it arranges how its communication with the real resources being managed should be controlled by the Coordinator. (scheduleWakeups(UXINTVL, NULLCP, True)).

Most of the attributes that will be used are derived from a few common types - gauges, counters etc. These are all provided by the GMS and it is only necessary to instantiate them. Their definitions may be found in $(TOP)/agent/gms/SmiAttr.h and they are documented below. Where attributes with non-standard behaviour are used, these must be implemented specially. There is an example of this type of attribute in the UxObj1 class - sysTime. The implementation of this attribute and other features of attribute types are discussed in Section 5.1.3.2.

As a consequence of executing the UxObj1 constructor, the constructor of the base MO class will be invoked. This will ensure that the new instance of the UxObj1 class will be properly bound in to the rest of the system. The CMIS agent will know it is there and will be able to GET and SET its attributes etc.

5.1.1.3 Knowledge Sources

The generic Knowledge Source (KS) object class (see $(TOP)/agent/gms/GenericKS.h) satisfies two main needs:

First, it provides some standard methods which may be invoked by the Coordinator when significant events occur. The most important of these are two "call-back" methods:

KS::readCommEndpoint (int)Which is invoked by the Coordinator when data is available to be read from a file-descriptor. This is the mechanism used in an event-driven regime to receive messages from the real resource.

KS::wakeUp (char*) Which is invoked by the Coordinator when a timer event occurs. This is the mechanism used to drive a polling regime.

These are virtual methods which must be specialised to do the right things for a particular MIB implementation (see UxObj1::wakeUP() in $(TOP)/agent/ux_mib/UxMO1.cc for example). In some cases, an MIB-specific KS class will be derived from the generic KS class and an instance of this class will be created at run-time (see below). In other cases, the class representing the MO will be derived from both the generic MO class and the generic KS class and there will never be an instance of the KS class as such. This latter is the approach used for the UxObj1 class.

Second, it can provide a "multiplexing/demultiplexing" facility. This is needed where some largely unrelated MO instances are updated through the same mechanism.

Consider the case of retrieving information from an intelligent communications processor which is implementing layers 1-4 of the OSI model. If the information retrieved in a message from the communications processor always relates to just one Layer then we have no problem since (presumably) all MO instances associated with a Layer are in a sub-tree of the containment tree. Therefore, we can associate the KS callback method with the class of the MO at the root of the sub-tree and arrange that this method, when invoked, distributes the information correctly between the objects in the sub-tree. However, if the information retrieved in a message can relate to several layers it will affect objects in several sub-trees. In this case we would create a stand-alone instance of the KS class to handle communications and timer events on behalf of all four sub-trees. This has been done in the case of the ISODE MIB; though, strictly speaking, it is not necessary at present since only Transport Layer objects are supported.

5.1.2 Communicating With Real Resources - Examples

5.1.2.1 Polling Example - the nUsers Attribute

The last line of the UxObj1 constructor makes the following call:

```
scheduleWakeUps(UXINTVL, NULLCP, True)
```

This is a method inherited from the KS class and requests polls at UXINTVL ms intervals - this request is passed on to the Coordinator object.

When the poll occurs the wakeUp() method is called. As noted in the previous section, this is a virtual method of the KS class but it is re-defined for the UxObj1 class. This re-definition may be found in UxMO1.cc and simply calls the update() method (see Section 5.1.3.1).

In this particular case, update() is really only concerned with the nUsers attribute. It uses the Unix "users" command to update that attribute and then calls the method triggerEvent(int, int) in the event that the associated threshold has been exceeded (see Section 5.1.3.3). Note that the obvious alternative communications mode - "upon external request" cannot be used here since we would only discover that the threshold had been exceeded when and if an external request was received.

5.1.2.2 Upon External Request Example - the sysTime Attribute

When a CMIS GET request is received by the agent, and after all the scoping and filtering has been completed, the GMS agent will call the Attr::get() methods for each of the required attributes in order to retrieve their current values. For many implementations the standard Attr::get() method for the attribute type will suffice. The sysTime attribute of the UxObj1 MO is one example where this is not so. In this case, timeliness is so important that an "on external request" strategy is required with the system clock being consulted from within the Attr::get() method. The relevant code may be found in the file $(TOP)/agent/ux_mib/UxAttr.cc:

```
//
//  UxTime::get() - read the system clock and return the value
//

void* UxTime::get()
{
    set();              //  Just call the Time set() method. This does
                        //  the appropriate initialisation
                        //  Then call the equivalent routine from
                        //  the parent class

    return Time::get();
}
```

The C++ class representing the UxTime attribute type is derived from the Time class supplied with the GMS. The Time class has a method, Time::set(), which, when called without arguments, simply updates the attribute value from the system clock. This method is used to perform the update prior to returning a value.

The implementation of the sysTime attribute illustrates the implementation of an "on external; request" strategy through the provision of a special UxTime::get() method for the attribute type. This is not the only means of implementing the strategy. The GMS also provides a mechanism which applies to the MO as a whole. This mechanism too is used in the UxObj1 class and is invoked through the MO::get() method - a virtual method of the base MO class. The CMIS agent always calls this method before retrieving any attribute value. For the UxObj1 class the virtual method is overridden by the following (from UxMO1.cc):

```
int  UxObj1::get (int attrId, int classLevel)
{
    if (classLevel != UXOBJ1_LEVEL && classLevel != MO_ALLEVELS)
        return Top::get(attrId, classLevel);

    switch (attrId) {

    case MO_ALLATTRS:
    case NUSERS:
        update();       // update the nUsers attribute
        break;

    default:
        break;
    }

    return OK;
}
```

Here the classLevel parameter specifies the level in the inheritance hierarchy (UXOBJ1_LEVEL = 1, i.e. one below Top). If this does not correspond to UxObj1 the call is passed up to the Top class. The effect of the rest of the routine is to ensure that the update() method is called whenever the access relates to the nUsers attribute. This ensures that the

value of this attribute will be timely when it is (almost immediately) retrieved by the CMIS Agent.

Note that there is no particular reason why the sysTime and nUsers attributes should be handled differently. It is done in this way merely to illustrate that the two mechanisms are available. However, there are cases where the mechanism used for nUsers has decided advantages. Consider again the "intelligent communications processor" example, this time using an "on external request" strategy. Suppose the refreshing of the attribute values was done in the Attr::get() methods as for sysTime. A CMIS request for multiple attributes would result in a plethora of refreshes as each Attr::get() method was called. Each refresh would involve communication with the communications processor - probably an expensive procedure. Fortunately we can take advantage of the fact that, when multiple attribute values are accessed, the CMIS Agent calls the MO::get() method for each one followed by one additional call with the attrId parameter set to MO_NOMOREATTRS. All of this happens prior to the retrieval of any attribute values by the CMIS Agent. This allows the implementor the possibility of writing a specialised MO::get() method which notes which attributes will be accessed and refreshes them all in one interaction with the communications processor when it sees the MO_NOMOREATTR value.

Although not used in the UxObj1 object, the GMS provides similar mechanisms to the above in the case of SET and ACTION operations.

5.1.2.3 Event Driven Example - the tpEntity Managed Object

Consider now the the TpEntity class which implements a MO class for an OSI Transport entity. Obviously some means must be found for accessing information held in the space of the Transport protocol code. In the case of the ISODE software, the Transport protocol code is present in every user process which is using ISODE. Therefore, accessing Transport protocol information necessarily implies some form of IPC. In fact, the chosen mechanism is to have the Transport protocol code emit UDP messages at significant moments. All these messages are directed to a common UDP port. The GMS system listens on this port and so receives information from all user processes employing the ISODE transport.

In Berkeley-based Unix systems, listening on a port implies listening on a socket. In fact, the agent system will be listening on several sockets. Some of these will be for IPC communication with real resources as for the TpEntity class, others will be for incoming requests from CMIS clients. It is essential that all these sockets are managed centrally otherwise a blocking read on a socket would halt the whole system. This socket management is handled by the Coordinator which executes a "select" system call on all the relevant sockets and passes incoming messages to the appropriate objects.

Referring to files in $(TOP)/agent/isode_mib, setting this up works as follows:

1. as noted in Section 5.1.2.3 a separate instance of the KS class is used to manage communication on behalf of all ISODE MOs. This is declared as a global in $(TOP)/agent/isode_mib/IsodKS.cc

2. the constructor for the TpEntity class registers the presence of a new instance of the class with the ISODE KS object through the use of its registerMO method. This can be found in the file $(TOP)/agent/isode_mib/TpEntity.cc .
3. the ISODE KS object arranges that a socket is created and configured for incoming UDP messages, and passes this socket to the Coordinator via the Coordinator::registerCommEndpoint() method. The Coordinator then uses the ISODE iserver_wait() call to wait for incoming messages on this (and other) sockets.
4. when a message is received at the socket, the Coordinator invokes a KS method to read the socket. In the case of the ISODE TP objects, this method is specialised in IsodeKS::readCommEndpoint() defined in $(TOP)/agent/isode_mib/IsodKS.cc. In addition to updating the TpEntity object this method also updates any TpConnection objects that might be present.

5.1.3 The Complete Example UNIX Managed Object

5.1.3.1 Methods

It must be emphasised that the vast bulk of the operations that need to be supported by a MO class are provided by the methods inherited by the basic C++ MO class. These include methods to GET and SET attribute values, to place an object instance correctly in the containment hierarchy according to its RDN, to locate object instances given a DN, to apply a filter to the attributes of an object and so on.

The UxObj1 object is fairly typical with just four specialised methods being required. These have mostly already been described. In summary they are:

UxObj1::buildReport (int, int, PE);*

This constructs an event report when the threshold on the number of users is triggered.

UxObj1::wakeUp (char);*

This is called periodically by the Coordinator as explained in Section 7. It simply calls the update() method - see below.

UxObj1::get (int, int);

This is called prior to the GMS agent reading attributes. It ensures that attributes are up-to-date before they are read by calling the update() method when necessary (See Section 5.4).

UxObj1::update ();

This updates the attributes. In this case, the principle effect is to update the nUsers attribute and to check whether the associated threshold has been exceeded.

The definitions of these methods may be found in $(TOP)/agent/ux_mib/UxMO1.cc.

5.1.3.2 Attributes

The attribute nUsers is of type "integer gauge". The GMS provides a C++ class to represent such a gauge. Like all such classes, this one is derived from the generic Attr class defined in $(TOP)/agent/gms/GenericAttr.h. The definition of the GaugeInt class itself may be found in ${Top}/agent/GMS/SmiAttr.h. "Standard" attribute types such as GaugeInt provide methods to:

- service requests from the CMIS Agent corresponding to CMIS requests (GET SET etc.)
- encode and decode attribute values in line with the ASN.1 syntax.
- allow the attribute values to be manipulated by user code so that they reflect the current state of the corresponding real resource.

When using these attribute types provided by the GMS, the user's principle task is to ensure that the attribute's value is up-to-date when it needs to be. There is an example of this in the UxObj1::update() method in the file $(TOP)/agent/ux_mib/UxMO1.cc:

```
// Adjust nUsers. Since there is a threshold associated with
// it we must check whether it has been triggered. If it has,
// set will return "On".

if (((GaugeInt*) _attrs[NUSERS]) -> set(words) == On) {
```

This is calling the GaugeInt::set(int) method in order to update the observed value. Note that, in this case, the returned value is checked in case the associated threshold has been triggered (see Section 10).

Whilst the UxTime class is not a standard one, it is straightforwardly derived from a standard class - the Time class. The distinction between these two C++ classes is required solely in order to implement the "on demand" refresh strategy for the UxTime class. As far as the OSI management information model is concerned there is no distinction - both correspond to the same attribute type. This is important since it implies that the two classes will share the same attribute value syntax and hence the same encoder and decoder - and these are already supplied for the Time class by the GMS. Were this not the case it would be necessary to write specialised encoders and decoders - a complex procedure using the PEPY ASN.1 compiler.

5.1.3.3 Notifications

The class Top provides a method called Top::triggerEvent() which should be called whenever a defined event has occurred - the MO implementor is responsible for ensuring that this is done. As a result of calling this method it may be that a CMIS M-EVENT-REPORT is issued, a log is updated, or both, or neither. Which it is is determined by the current state of the discriminator and log objects. Discriminators and log objects are specialised MOs provided entirely by the GMS; they contain filter constructs. When Top::triggerEvent() is

called, a temporary object is formed which contains information about the event; its OID, the class and DN of the object generating the event and so on. The filters of all current discriminators are applied to this temporary object. It any returns true then the appropriate action is taken (issue an EVENT REPORT or generate a log record). The GMS provides full support for the creation and deletion of discriminator and log objects in response to CMIS requests. This is a completely general mechanism and need not concern the MO implementor. All s/he has to do is to ensure that Top::triggerEvent() is called at the appropriate moment and provide a routine to build an event report or log record as needed.

There is one example of this in the UxObj1 class. A threshold is applied to the nUsers gauge and a notification should be generated when this threshold is triggered. The GaugeInt class has built-in knowledge about thresholds:

- it provides a method GaugeInt::associateThreshold() to tie together the gauge and the threshold.
- it checks whether the threshold has been exceeded each time the GaugeInt::set() method is called to alter the "observed value". The value returned by this method indicated whether or not the threshold has been triggered.

This can be seen in use by referring to $(TOP)/agent/ux_mib/UxMO1.cc. In the method UxObj1:UxObj1() the gauge/threshold association is established:

```
//  associate the gauge (nUsers) with the corresponding
//  gauge threshold (nUsersThld).
((GaugeInt*) _attrs[NUSERS]) -> associateThreshold
                    ((GaugeThresholdInt*) _attrs[NUSERSTHLD]);
```

In the method UxObj1:update() the observed value is updated using GaugeInt::set() and if the value returned by this is "On" then the Top::triggerEvent() method is called:

```
// Adjust nUsers. Since there is a threshold associated with
// it we must check whether it has been triggered. If it has,
// set will return "On".

if (((GaugeInt*) _attrs[NUSERS]) -> set(words) == On) {

    //  Now check whether an event report should be generated.
    //  EFDCheck will look through its list of EFDs and check whether
    //  any of them has a filter which evaluates to true. For any
    //  that do an event report will be sent. The event report
    //  is constructed by UxObj1::buildReport()

    triggerEvent(NUSERSTHLDEXCEEDED, UXOBJ1_LEVEL);
```

Assuming that a discriminator filter returns "true" a report must be generated. This is done by the UxObj1:buildReport() method which can be found in $(TOP)/agent/ux_mib/UxMO1.cc. This first sets up the information to be included in the report in a structure of type UxObj1Report (see UxRepAsn.h). It then calls a PEPY-generated routine (build_UxRep_UxObj1Report()) to construct the PE as required by ISODE. The relevant code is reproduced below:

```
report.uxobj1rep_sysTime = *((UxTime*) _attrs[SYSTIME]) -> getutc()
                                    // Time::getutc() returns a
                                    // pointer to a UTCtim struct.
                                    // The indirection implies a
                                    // struct copy.

report.uxobj1rep_nUsers = *(int*) _attrs[NUSERS] -> get();
                                    // GaugeInt::get returns an int

fprintf(stderr, "Time check %s users %d0,
            utct2str(&report.uxobj1rep_sysTime),
            report.uxobj1rep_nUsers);

// Construct a PE for the report

if (build_UxRep_UxObj1Report (info, 1, 0, NULLCP, &report) == NOTOK) {
```

The PEPY source from which the encoder and decoder are built may be found in the file $(TOP)/agent/ux_mib/UxRepAsn.py. The contents of this file will be meaningful only if you understand PEPY. If you do not, refer to the ISODE manual...

5.1.3.4 Creation

Finally, let us look at how things get started. In the directory in which you build your complete system ($(TOP)/agent/sma for example) there should be a copy of Create.cc. You should produce a customised Create.h which specifies which Managed Object classes are supposed to be present, whether inst ances of these should be created at start-up and whether they may be created as a result of CMIS requests. (They may also be created as a result of some event within the "real resource" - the opening of a new connection for example.) This information is contained in the moclasses[] array, the entry for the UxObj1 class in Create.h is:

```
//  class name          CMIS M_Create                   MIB initialise

    ...

    "uxObj1",           UxObj1::cmisCreate,             UxObj1::create
```

This specifies that an instance of the UxObj1 class may be created at initialisation time by calling the routine UxObj1::create(). If this entry were instead "NULLCREATE" then no instance could be created initially. UxObj1::create() is very simple and consists of the single line:

return new UxObj1(rdn, superior, True, True);

The value in the CMIS M_Create column indicates that a routine has been defined dynamically to create instances of the UxObj1 class in response to CMIS requests. The value NULLMCREATE can be used to indicate that such creations are not allowed.

5.2 Managed Object Support

Class MOClassInfo

Inherits from:	none
Classes used:	NameBinding, MO, the ISODE OID structure
Interface file:	GenericMO.h
Implementation file:	GenericMO.cc

Introduction

The MOClassInfo class is a meta-class describing a managed object class. It is similar to the notion of the Smalltalk/ObjectiveC class object and there is always one instance of it for every managed object class. It contains common information to all the objects of the class, such as attribute, group, event and action identifiers, name bindings and possibly a pointer to the first instance of the class from which subsequent instances may be addressed (the first regarding its creation in time).

Every managed object class should own a static instance of this class which will then be shared by all its members. This is usually defined within the managed object class declaration and is initialised by the initialiseClass() method of the latter, see the MO class specification for details. Most of the methods of this class are only used from the generic parts of the GMS to access attributes, actions etc. The only parts interesting the managed object class implementers are the ones regarding its initialisation with information and the addressing of the first managed object instance. These are described below.

Methods

```
class MOClassInfo
{
    // ...

public:
    Bool initialised ();
    int  setClass (char*, int, int, int, int, int, int = 0);
    int  setAttr (char*, int, Bool = False, Attr* = NULLATTR);
    int  setNameBinding (char*, char*, char*);
    int  setGroupAttr (char*, int, int);
    int  addGroupAttr (int, int);
    int  setEvent (char*, int);
    int  setAction (char*, int, Attr*);
    int  setPackage (char*, int);
    MO*  getFirst ();
    // ...
};
```

Initialising the class information

Bool initialised *();*

Usually, the class information this object holds is initialised when the first managed object instance of the associated class is created. After this initialisation, this method returns True so that subsequent object instances do not re-initialise it.

int setClass *(char* className, int nbindings, int nattrs, int ngroups, int nevents, int nactions, int npackages);*

This should be called first to initialise the class identifier and declare how many name bindings, attributes, attribute groups, notifications, actions and packages the class has so that storage space is allocated. *className* is as registered in oidtable.at. Note that the last parameter is optional for backwards compatibility reasons, you should supply it even if it is zero as it will be mandatory in future versions.

int setAttr *(char* attrName, int attrId, Bool settable, Attr* defaultValue);*

This should be called repetitively for all the attributes of the class. *attrName* is as registered in oidtable.at and the *attrId* is the integer tag assigned to it within the class. *settable* should be set to True is the attribute is settable and *defaultValue* should be the default value. Note that the two last parameters are optional and may be omitted if the attribute is not settable.

int setNameBinding *(char* nameBinding, char* superiorClass, char* namingAttr);*

This should be called repetitively for all the name bindings of the class. It should be called AFTER the attributes have been registered as above. The naming attribute is as registered in oidtable.at while the name binding and the superior class are as in oidtable.gen.

int setGroupAttr *(char* groupName, int groupId, int ngroupAttrs);*

This is similar to setAttr but for an attribute group. The group name should be registered in oidtable.gen and the *groupId* is the integer tag assigned to it within the class. *ngroupAttrs* declares the number of attributes in that group so that space is allocated.

int addGroupAttr *(int groupId, int attrId);* This should be called repetitively for all the attributes of each group in the class. The group and attribute ids are the integer tags assigned to them within the class. This method should be called **AFTER** the attributes have been registered as above.

int setEvent *(char* eventName, int eventId);*
int setPackage *(char* packageName, int packageId);*

These are similar to setAttr for events and packages. Events (notifications) should be registered in oidtable.at while packages in oidtable.gen.

int setAction *(char* actionName, int actionId, Action* actionTemplate);* This is similar to the above for actions, but *actionTemplate* is an additional parameter with a dummy value that is used by the GMS for decoding the action argument and encoding the action result. Actions are registered in oidtable.at and their syntax should be the ASN.1 CHOICE of their information and reply ASN.1 syntaxes. This is because there can be only one syntax associated with each entry in oidtable.at.

Accessing the first MO instance

MO* getFirst *();*

Usually, objects of the same class may be linked together in a doubly linked list, independently of their links in the MIT. In this case, the MOClassInfo object holds a handle to the first one while the list may be traversed using the getClassNext() and getClassPrev() MO methods. This method returns a pointer to the first instance of the associated class if the instances are linked together, NULLMO otherwise.

The managed object class implementer may control if the instances should be linked together through the registerClass() MO method. A static method of that class called by convention getClassInfo() should return a handle to the MOClassInfo object. That way, instances of a class may be accessed without searching through the MIT. For example, the first instance of a transport connection class may be accessed as follows:

```
TPConnection* tpConn = (TPConnection*) TPConnection::getClassInfo() -> getFirst();
```

Class MO

Inherits from:	None
Classes used:	MOClassInfo, Attr
Interface file:	GenericMO.h
Implementation files:	GenericMO.cc, MibAccess.cc, Create.cc

Introduction

MO is the abstract superclass of all managed object classes. It contains information related to the position of a managed object instance in the containment hierarchy. It also contains handles to management information held by classes derived from it so that access through the OSI management protocol (CMIS/P) may be automated. This class defines a set of virtual methods which may be redefined by derived classes to achieve the desired functionality.

Methods

```
class MO
{
    // ...
    /*
    static int          initialiseClass ();
    */

protected:
    // virtual methods (to be possibly provided in derived classes)

    virtual int         get (int, int);
    virtual CMISErrors  set (CMISModifyOp, int, int, void*);
    virtual CMISErrors  action (int, int, void*, void**);
    virtual CMISErrors  cmisDelete ();
    virtual int         buildReport (int, int, PE*);
    virtual CMISErrors  refreshSubordinate (RDN);
    virtual CMISErrors  refreshSubordinates ();

    // ...
```

```
public:
    // interface to managed objects, knowledge sources and the CMIS agent

    static int  initialiseMIB ();
    static MO*  getRoot ();

    MO*         getMO (DN, Bool = False);
    MO*         getMO (char*);
    MO*         getSubordinate (RDN, Bool = False);
    MO*         getSubordinate (char*);
    MO**        getSubordinates (Bool = False);
    MO**        getWholeSubtree (Bool = False);

    MO*         getSubordinate ();
    MO*         getPeer ();
    MO*         getSuperior ();

    /*
    static MOClassInfo*  getClassInfo ();
    */
    MO*         getClassNext ();
    MO*         getClassPrev ();

    OID         getClass ();
    char*       getClassName ();
    RDN         getRDN ();
    char*       getRDNString ();
    int         setRDN (RDN);
    int         setRDN (char*);
    DN          getDN ();
    char*       getDNString ();

    Attr*       getAttr (char*);
    void*       getAttrVal (char*);
    int         setAttrVal (char*, void*);
    int         replAttrVal (char*, void*);


    // managed object creation
    /*
    static MO*  create (RDN, MO*);
    static MO*  cmisCreate (RDN, MO*, MO*, int, CMISParam*,
                            CMISErrors*, CMISErrorInfo*, OID*);
    */

    // ...
};
```

These methods constitute the interface offered to managed objects, knowledge sources and the CMIS agent to access and modify the Management Information Base (MIB).

MIB initialisation

static int initialiseMIB *();*

This method is called in the main program of every agent and it initialises the MIB by reading the ETCDIR/mib.init file, other persistent objects from LOGDIR/mib/ and event logs from LOGDIR/logs. It returns OK upon success, NOTOK upon failure. **Managed object addressing**

The following set of methods allow to address managed objects in the Management Information Tree (MIT), that is the managed object containment hierarchy within that management application process. All, apart from the first one, may cause access to the associated real resources by calling the refreshSubordinate or refreshSubordinates methods described previously. This will occur if a fetch-on-request regime is exercised regarding real resource access. If this is not desirable, methods of the next group described later should be used.

static MO* getRoot *();*

This method returns the root which is the "handle" to the MIB. By being static, it allows always access to the MIB root as follows:
MO* root = MO::getRoot();

MO* getMO *(DN distinguishedName, Bool refresh);*

This method finds a managed object hanging off another one in the Management Information Tree (MIT). Its argument should be a portion of the distinguished name for the object, starting after the object to which the method is applied.

Some managed objects may be in the MIB but they may not have an in core representation in the MIT. This is true for example for log records which would be in secondary storage, routine table entries which could be in the operating system’s kernel address space etc. Usually, an object containing them (e.g. the log for log records) is aware of this and revives them when needed. If the *refresh* parameter is set to True, then in the process of searching for the object in the MIT, objects along that path will be told to revive their subordinates.

The above parameter is also present in other object addressing and scoping methods and is usually set to True when searching for objects after a CMIS request, which happens transparently. There is no need to use this facility when searching for objects from within the agent. The parameter has the default value False, which means that it can be omitted to avoid the reviving of objects.

The distinguished name may be constructed from a string format using the ISODE dsap library procedure "DN str2dn(char*)" when object identifier / syntax tables are in use. A pointer to the managed object is returned upon success, NULLMO otherwise.

As an example, consider the following MIT branch:
systemId = athena, subsystemId = 4, entityId = ISOTP, connectionId = 12345
Having a pointer to the subsystem object, the transport connection one may be accessed using a distinguished name with string representation:
"subsystemId=4@entityId=ISOTP@connectionId=12345"

MO* getMO *(char* distinguishedName);*

This is the same as above but offers a friendlier interface, allowing to use a string representation of the distinguished name. It assumes that object identifier / syntax tables are in use. This does not offer the facility of reviving objects as method.

MO* getSubordinate *(RDN relativeDistinguishedName, Bool refresh);*

This is similar to getMO but searches only for a first level subordinate. The getMO method may achieve the same effect using a one component distinguished name but with some performance cost. The *refresh* parameter is as described above. The relative distinguished name may be constructed from a string format using the ISODE dsap library procedure "RDN str2rdn(char*)" when object identifier / syntax tables are in use. A pointer to the managed object is returned upon success, NULLMO otherwise.

MO* getSubordinate *(char* relativeDistinguishedName);*

This is the same as above but offers a friendlier interface, allowing to use a string representation of the relative distinguished name. It assumes that object identifier / syntax tables are in use. This does not offer the facility of reviving objects as above.

MO getSubordinates** *(Bool refresh);*

This method returns an array of pointers to the first level subordinates, terminated by an empty cell i.e. containing NULLMO. The array returned is in static storage, so it should NOT be free'd. The *refresh* method offers the same facility as described above.

MO getWholeSubtree** *(Bool refresh);*

This is similar to the one above but it returns an array of pointers to all objects under the one on to which is applied.

Walking through the Management Information Tree

The following set of methods allow to access managed objects in the management information tree by simply walking the pointers that link them. As such, they do not cause any updates by accessing the associated real resources.

MO* getSubordinate *();*

The MIT is represented internally as a binary tree. This method returns the first of its immediate subordinates.

MO* getPeer *();*

This method returns the next peer object (sibling) in the containment hierarchy. Peer objects are all those having a common superior i.e. its immediate (first level) subordinates.

MO* getSuperior *();*

This method returns the object's superior.

Using these three methods and checking the relative distinguished name using the getRDN/getRDNString methods, one may achieve the same effect as using any of the methods in the previous group.

Accessing objects of the same class

The following set of methods allow to access managed objects of the same class. As it has been explained, objects of the same class may be linked together in a doubly linked list if the class implementor wishes so. If an object of the class is known, the list may be walked in both directions. If no object is known, the first one may be accessed through the MOClassInfo object for that class.

static MOClassInfo* getClassInfo *();*

This method does not actually belong to the MO class and it cannot be defined as virtual as it should be static. Implementors of of derived classes should provide it if they wish to offer access to all objects of the same class when no object of that class is known.

The method should return (a pointer to) the static MOClassInfo object that contains all the class associated information. The getFirst method of the latter may then be used to access the first instance. As an example, the following line of code allows to access the first transport connection:
TpConnection* firstConn = (TpConnection*) TpConnection::getClassInfo() -> getFirst();

MO* getClassNext *();*

This method returns the next (forward) one in the doubly linked list of all objects of the same class.

MO* getClassPrev *();*

This method returns the previous (backward) one in the doubly linked list of all objects of the same class.

It is noted here that the terms next/forward and previous/backward relate to the point in time when the managed objects were created: next/forward means later in time.

Accessing the object class and name

The following group of methods allow to find out information about an object such as its class and name. There are also methods that allow to change an object's name.

OID getClass *();*

This returns the object class as an object identifier. This is the leaf class of the inheritance tree. It may be printed in a friendly format using the *"char* oid2name(OID, int)"* dsap library procedure if object identifier / syntax tables are in use. The object identifier returned is not a copy and it should NOT be free'd.

char* getClassName *();*

This is the same as above but offers a friendlier interface, returning a string representation of the class. It assumes that object identifier / syntax tables are in use. The character string returned is in static storage and should NOT be free'd.

RDN getRDN *();*
DN getDN *();*

These return the object's relative or full distinguished name. The latter be printed in a friendly format using the dsap library procedures "char* rdn2str(RDN) and char* dn2str(DN)" when object identifier / syntax tables are in use. The name returned is not a copy and should NOT be free'd.

char* getRDNString *();* **char* getDNString** *();*

These are the same as above but offers a friendlier interface, returning a string representation of the name. It assumes that object identifier / syntax tables are in use. The character string returned should be free'd.

Accessing management attributes

The following group of methods allow to access management attributes within a managed object.

Attr* getAttr *(char* attrName);*

This method returns a (pointer to an) attribute object identified by attrName. The latter should be as registered in the object identifier tables. If the attribute name is wrong/non-existent, NULLATTR is returned. The attribute returned is not a copy and it should NOT be free'd.

void* getAttrVal *(char* attrName);*

This method returns the value of the attribute identified by attrName. If the attribute name is wrong/non-existent, NULLVD is returned. The attribute value returned is not a copy and it should NOT be free'd. This method is actually an inline wrap-up of the getAttr one, using the "void* getval()" virtual Attr method. The result should be casted to the appropriate data type. See also the Attr class specification.

int setAttrVal *(char* attrName, void* attrValue);*

This method sets the value of the attribute identified by attrName to attrValue. The old value within that attribute object is free'd. If the attribute name is wrong/non-existent, NOTOK is returned. The attribute value supplied should have allocated memory. This method is actually an inline wrap-up of the getAttr one, using the "void set(void*)" virtual Attr method. See also the Attr class specification.

int replAttrVal *(char* attrName, void* attrValue);*

This method replaces the value of the attribute identified by attrName to attrValue. The old value within that attribute object is NOT free'd. If the attribute name is wrong/non-existent, NOTOK is returned. The attribute value supplied should have allocated memory. This method is actually an inline wrap-up of the getAttr one, using the "void Attr::replval(void*)" method. See also the Attr class specification.

Virtual Methods

The following constitute a set of virtual methods that may be redefined by derived classes.

virtual int get *(int attrId, int classLevel);*

The get method should only be redefined when the class attributes need to be updated before responding to a CMIS M-GET request. This is needed when a fetch-on-request regime is exercised regarding access to the associated real resource. In the case of an event- or poll-driven approach, this method may not be redefined, incurring though a cost in management information timeliness.

Redefining this method can be avoided even on a fetch-on-request regime. This can be done by customising the attribute types for that object and associate them with the real resource i.e. redefine their get() method. This approach is fine when the information for the various attributes is in the local address space and is unrelated but can be inefficient when say kernel access is needed or when the actual attribute values reside in another process address space and should be accessed via an IPC mechanism. In this case, it is better for the MO to have this knowledge rather than the attributes in order to optimise access. It is this method where this knowledge should be.

The method should retrieve the necessary information and update the requested attribute(s), using methods of the Attr class (setval, replval) or customised methods of its derived classes i.e the attribute types. *attrId* is the integer index of that attribute in the array of attributes for the class and *classLevel* is the level of the class in the inheritance hierarchy tree (0 for Top). In case all the attributes are requested, the values of these parameters are MO_ALLATTRS and MO_ALLEVELS. The end of a series of attribute get requests pertaining to a CMIS request is indicated by a call with the values MO_NOMOREATTRS and MO_ALLEVELS.

Accessing the real resource on a per attribute basis is not usually efficient. Various schemes may be implemented to improve efficiency e.g. using timestamps on a per object or attribute basis etc. A simple typical scheme would be to attempt the real resource "read" operation after all the requested attributes have been marked as such. The organisation of this method would then look like:

```
int  <Class>::get (int attrId, int classLevel)
{
    static Bool update = False;
    // static state information here ...

    if (classLevel != <CLASS_LEVEL> && classLevel != MO_ALLEVELS)
      return <ParentClass>::get(attrId, classLevel);

    switch (attrId) {
    case <ATTR_X>:
      // mark attribute X as requested
      break;
    case <...>:
      // ...
    case MO_ALLATTRS:
      // mark all attributes as requested
      update = True;
      break;
    case MO_NOMOREATTRS:
      update = True;
      break;
    default:            // for attributes that need no "refresh"
      break;
    }

    if (update == True) {
      // fetch and update the requested attributes
      // reset state information ...
      update = False;
    }
    return OK;
}
```

A fetch-on-request regime is usually exercised when management information resides in accessible address space i.e. kernel, shared memory etc. It can also be exercised in the case the management information is held by another process, by another agent for which this is a proxy etc. The managed object may obtain the communication endpoint though the corresponding knowledge source. It should then request the information and perform a blocking read with a timne-out value until it receives the response. This can be done through the xselect() call, see ISODE User's Manual Volume 1 Section 2.4 "Select Facility".

The problem with this approach lies in the synchronous nature of the operation which results in all the activity in the agent ceasing until this information has been received. An asynchronous interface between the CMISAgent object and the MIB is clearly needed to allow the management agent to continue its operation as normal. Such a facility will be provided in future OSIMIS versions.

virtual CMISErrors set *(CMISModifyOp mode, int attrId, int classLevel, void* setValue);*

The set method should only be redefined if a set operation on a management attribute should result in modifying information in the associated real resource. Is is noted that there may be

attributes for which a set operation may not trigger a real resource associated update/action, such as thresholds etc. For these no action is required as they are handled by the generic parts of the MO class.

mode is the set operation mode (m_replace, m_addValue, m_removeValue or m_setToDefault). *attrId* is the integer index of the attribute in the array of attributes for the class and *classLevel* is the level of the class in the inheritance hierarchy tree (0 for Top). *setValue* is the value to be used in the set operation. Note that the same value will be used by the GMS to set the actual attribute if m_noError is returned so it should NOT be free'd. The only errors that should be returned is m_invalidAttributeValue if the value is out-of-range or otherwise inappropriate (though the decoder should tackle this in most cases) or m_processingFailure if something goes wrong to the interaction with the real resource. The end of a series of attribute set requests pertaining to one M-SET CMIS request is indicated by a call with the values MO_NOMOREATTRS and MO_ALLEVELS.

Similar optimisation methods as for the get() method above may be exercised.

virtual CMISErrors action *(int actionId, int classLevel, void* actionInfo, void** actionReply);*

This is similar to get and set regarding the *actionId* and *classLevel* arguments and real resource access. The only difference to set is that there is only one action as opposed to many attributes to set on one operation and a reply is possibly needed. *actionInfo* is the action information and *actionReply* is the reply which should be set (if any). You should NOT free the actionInfo and you should make sure that actionReply has allocated memory.

m_noError should be returned if everything went ok while m_invalidArgumentValue or m_processingFailure should be returned in case of an error in a similar fashion to the set method.

virtual CMISErrors cmisDelete *();*

This will delete the managed object if the latter can be deleted through CMIS. According to the class specification, this could be only allowed if there are no contained objects. In this case, the contained objects could be obtained by using the getSubordinates method with True as argument to possibly check the real resource. m_accessDenied should be returned if deletion is not possible e.g. there are subordinate objects in the above case.

If it is known that the class has subordinate objects and these should be deleted, this should be taken care in the destructor. A typical implementation of this method checks possibly for subordinate objects which may result in returning m_accessDenied and asimply calls "*delete this; return m_noError;*". The destructor should take care any cleaning-up, interactions with the real resource etc.

virtual int buildReport *(int eventId, classId, PE* report);*

This is used to build an event report in order to be forwarded and/or logged as a log record. It is called after the notification function has made sure that this is needed, so processing is

optimised. *eventId* and *classId* identify the event in a similar fashion to the attributes and action in the previous methods. *report* is a presentation element which needs to be constructed by using the encoder for the even report syntax.

A C data structure for the report needs first to be filled in with the information that should go in it which should be obtained by accessing the attributes. Be careful not to double-free information - using the attribute get and getval methods return the actual data, NOT copies. OK should be returned upon success, NOTOK upon failure (should only happen at development time if encoding failed).

virtual CMISErrors refreshSubordinates *();*

The existence of some managed objects is only known by accessing the associated real resource. This is true for example for resources that cannot notify the managed system when they are created or deleted, as for example for table entries held in an operating systems kernel or in a "non-intelligent" loosely coupled resource. In this case, answering to CMIS requests could be done in two ways: using a cache-ahead scheme through polling or simply a fetch-on-request regime.

The cache-ahead scheme has the drawback of introducing traffic between the managed system and the real resources while it also affects the timeliness of information. It cannot be avoided though, at least to some extent, if notifications are to be supported. The fetch-on-request scheme rectifies these problems but cannot support notifications and increases the response time to CMIS requests.

If the fetch-on-request scheme is adopted for managed objects that can dynamically change, some way is needed to "refresh" before accessing them through CMIS. The GMS approach is to put this knowledge in the containing object: that should redefine this method in order to fetch its subordinates from the real resource i.e. create new ones and delete those that have died. The method is called when the subordinates are needed because of scoping.

virtual CMISErrors refreshSubordinate *(RDN rdn);*

This is exactly the same as above but for a particular subordinate which is identified through its relative distinguished name. In this case, only the specified object should be refreshed which may create it, delete it or do nothing. In the latter case, it would be better to refresh its data (attributes) in order to avoid another real resource access later. This can be done using a timestamp which the get method will consult to avoid refreshing the requested attributes again.

Class Top

Inherits from:	MO
Classes used:	MOClassInfo, EFDiscriminator, EventLog, EventRecord, ObjClass, NmBinding, ObjIdList
Interface file:	Top.h
Implementation file:	Top.cc

Introduction

The Top class implements the root of the management information inheritance tree i.e. the class from which all the other managed object classes are derived. Its attributes describe the managed object itself for the purpose of management access and are fundamental for allowing the exploration of the MIB in a generic fashion. These attributes are the *objectClass*, *nameBinding*, *packages* and *allomorphs*. OSIMIS supports packages and (in a limited fashion) allomorphism.

The reason for describing this class is twofold: first, because of its position as the root of the managed object inheritance tree. Second and most important, because of the fact it offers the interface to the event notification management function (event reporting and log control). Only the methods related to that function and the constructors are described below, as only these are of interest to derived and other classes.

Methods

```
class Top
{
    // ...

protected:
    Top (RDN, MO*);
    Top (RDN, MO*, int, CMISParam*, CMISErrors*, CMISErrorInfo*);

public:
    int  triggerEvent (OID);
    int  triggerEvent (char*);
    int  triggerEvent (int, int);

    static int  relayEventReport (OID, DN, char*, OID, PE);

    // ...
};
```

Constructors

Top *(RDN rdn, MO* superior);*

This is a protected method and should be called only from the constructors of derived classes. *rdn* is the relative distinguished name of the managed object and *superior* is the superior object in the containment tree (MIT). This type of constructor is used for creating an object when the agent is initialised (reading the mib.init file) or as the result of real resource activity (e.g. connection creation).

Top *(RDN rdn, MO* superior, int nattrs, CMISParam* initialAttrs,*
CMISErrors cmisError, CMISErrorInfo* cmisErrorInformation);*

This is a protected method as above and it is called when a managed object is created as a result of a M-CREATE CMIS request. *rdn* and *superior* are as above, *nattrs* and *initialAttrs* are number and the initial attribute values. These are checked in case the *packages* attribute is initialised with one or more optional packages for derived classes. The latter should check the value of this attribute in their constructor and instantiate the package(s) accordingly. *cmisError* and *cmisErrorInformation* are the error and information in case things go wrong, m_invalidAttributeValue is only possible is possible at this level.

The notification function

The following methods offer an interface to the notification function which allows to forward and/or log a notification converted to an event report or log record accordingly. This depends on the presence of event forwarding discriminators and/or log managed objects in the local MIB and is completely transparent to derived managed object class implementors.

int triggerEvent *(OID eventType);*
int triggerEvent *(char* eventType);*
int triggerEvent *(int eventId, int classLevel)*

All these are overloaded functions that offer the same interface to the notification function. In the first the event type is an object identifier and in the second a friendly name as registered in the oidtable.at. In the third one, the event type is actually shown by two parameters: the *classLevel* which shows which level of the object in the class inheritance tree is associated to the event and the *eventId* which is an integer tag associated to the event at that level (see also the MO class). The obvious one to use from within a derived class is the last while the first two are more suitable for usage by other objects.

static int relayEventReport *(OID moClass, DN moInstance,*
char eventTime, OID eventType, PE eventInfo)*;

Hybrid units which are both agents and managers may receive event reports. These could be possibly forwarded and/or logged according to the presence of event forwarding discriminators and/or logs in the local MIB. This is possible through this static method as the event report is not emitted by any of the local objects. The arguments are the class, instance, event time, event type and event information as received from the other agent via CMIS.

5.3 Attribute and Syntax Support

This section describes the various attribute type definitions in the GMS. The GMS contains implementations of some "standard" attribute types that should fulfill the requirements for the basic DMI attribute types. However, it will probably be necessary to define your own attribute types and incorporate them into the agent software.

The remainder of this section is split roughly in two; first the attribute types that are implemented in the GMS are described, then the procedure for introducing the implementation of an attribute type is explained.

5.3.1 Attribute types in the GMS

This section describes the implementation of attribute types in the GMS. If the reader is not familiar with the process involved for implementing new syntaxes for ISODE, it is recommended that the next sub-section, **"Introducing your own syntaxes into the software"**, be read first. The GMS contains syntax implementations of the following attribute types, which defined in the DMI (ISO 10165-2) :

- Integer
- Real
- CMISFilter
- OIDList
- Count
- CounterThreshold
- Gauge
- GaugeThreshold
- TideMark
- AdministrativeState
- OperationalState
- DestinationAddress (A modified version of the Destination syntax.)
- LogFullAction
- AvailabilityStatus

Also, implementations of the following syntaxes have been used from the dsap library:

- ObjectIndentifier
- DistinguishedName
- Presentation Address
- UTCTime
- OctetString
- IA5String
- PrintableString
- NumericString

Although all of the syntax implementations have routines to encode and free C structures, decode PEs into C structures, and pretty-print C structures, not all have routines for parsers, comparing two instances of the same type, or for copying instances of the same type.

The implementation of the syntaxes can be found in *${TOP}/agent/gms/SmiSntx.h* and *${TOP}/agent/gms/SmiSntx.c*. Additionally, some of the syntax routines have been generated using the **pepy** tool from ISODE. The augmented ASN.1 source for use with **pepy** is *${TOP}/agent/gms/SmiAsn.py*. The syntax implementations are then used in a C++ class definition to implement the attribute types.

The attribute types in the GMS are implemented as C++ classes of base class **Attr**. The definitions of the C++ classes can be found in *${TOP}/agent/gms/SmiAttr.h* and *${TOP}/agent/gms/SmiAttr.cc*. These attribute type implementations are intended for use in mainly in agents though they can be used in managers. However, the attribute type implemented by class **AnyType** may be more useful in managers. This will take an OID and a value and perform table look-ups (from ISODE **libdsap.a** routines) to "form itself" into the correct type.

The following table summarise information concerning the syntax implementations and the C++ class implementations for each attribute type in the GMS :

Type [Note]	C data type	C++ class	parse	compare	copy
Integer	`int *`	`Integer`	•	•	•
Real	`double *`	`Real`	•	•	•
CMISFilter	`CMISFilter`	`Filter`		•	
OIDList**[1]**	`ObjIdListVal *`	`ObjIdList`	•		
Count	`int *`	`Counter`	•	•	•
CounterThreshold **[2]**	`CounterThresholdVal *`	`CounterThreshold`	•		•
Gauge **[2]**	`ObservedValue *`	`Gauge[Int, Real]`	•		•
GaugeThreshold **[2]**	`GaugeThresholdVal *`	`GaugeThreshold[Int, Real]`	•		•
TideMark **[3]**	`TideMarkVal *`	`TideMark[Min, Max]`			
AdministrativeState	`AdminStateVal *`	`AdministrativeState`	•		
OperationalState	`OperStateVal *`	`OperationalState`	•		
DestinationAddress	`DestAddressVal *`	`DestinationAddress`			
LogFullAction	`LogFullActionValue *`	`LogFullAction`	•	•	•
AvailabilityStatus	`AvailabilityStatusValue *`	`AvailabilityStatus`	•	•	•
ObjectIndentifier	`OID`	`ObjId`	•	•	•
DN (DistinguishedName)	`DN`	`DName`	•	•	•
UTCTime	**[4]**	`Time`	•	•	•
IA5String	`char *` **[5]**	`String`	•	•	•
OctetString	**[6]**	`OctetString`	•	•	•

Notes

"•" indicates that the particular syntax routine for performing this operation is available.

1. This type does not exist, as such, in the DMI, however it is provided for convenience. It's ASN.1 definition is `OIDList ::= SET OF OBJECT IDENTIFIER`.

2. For convenience, the C++ implementation of these attribute types exist in two forms; one for integer quantities, and one for real quantities.

3. For convenience, the C++ implementation of this attribute type exists in two forms; one for a minimum tidemark and one for a maximum tidemark.

4. For convenience, the C++ implementation of this attribute type stores this value in two formats; the `UTCtime` structure and in the slightly more flexible `long` (`time_t`) as used by `time(3)`.

5. This is a NULL terminated `char` array.

6. For convenience, the C++ implementation of this attribute type deals with `char *` and `int` (pointer + length). When using the syntax routines directly, e.g. from the syntax tables, a `struct qbuf *` is used.

Please also not that although some syntaxes do have compare functipons defined for them, they are not used in the GMS. The use of the compare function may be examined in the future.

5.3.1.1 C++ implementations of attribute types in the GMS

This section describes the `public` interface to the attribute types implemented in the GMS. Only methods specific to the attribute type are explained. The use of the protected methods is explained with the documentation for `class Attr`. The following sub-sections are headed by the name of the C++ class that implements each attribute type (see *${TOP}/agent/gms/SmiAttr.h*).

5.3.1.1.1 Integer

```
     class Integer : public Attr
{
   ...

public:
    inline void set (int val)          { int* i = (int*) getval();
                                         *i = val; replval(i); }
    inline int  set (void* val)        { set(*(int*) val); return OK; }
    Bool        filter (int, void*);

    inline      Integer ()             { int val=0; setval(int_cpy(&val)); }
    inline      Integer (int val)      { setval(int_cpy(&val)); }
};
```

inline void set *(int val);*

Sets the contained value to the supplied one. The memory used to store the previous value is reused.

inline int set *(void* val);*

Sets the contained value to *val* (`int *`).

inline Integer *();*

Instantiate an attribute instance and allocates memory for the `int` to be used for the lifetime of the attribute.

inline Integer *(int val);*

Instantiate an attribute instance and allocate memory for the `int` to be used for the lifetime of the attribute. Sets the value to *val*.

5.3.1.1.2 Real

```
     class Real : public Attr
{
   ...

public:
    inline void set (double val)        { double* r = (double*) getval();
                                          *r = val; replval(r); }
    inline int  set (void* val)         { set(*(double*) val); return OK; }
    Bool        filter (int, void*);

    inline      Real ()                 { double val=0; setval(real_cpy(&val));}
    inline      Real (double val)       { setval(real_cpy(&val)); }
};
```

inline void set *(double val);*

Sets the contained value to *val*.

inline int set *(void* val);*

Sets the contained value to *val* (`double *`).

inline Real *();*

Instantiate an attribute instance and allocate memory for the `double` to be used for the lifetime of the attribute.

inline Real *(double val);*

Instantiate an attribute instance and allocate memory for the `double` to be used for the lifetime of the attribute. Sets the value to *val*.

5.3.1.1.3 OctetString

```
     class OctetString : public Attr
{
   ...

public:
    inline void set (char* v, int l)    { char* tv = (char*) malloc(l+1);
                                          len = l; bcopy(v, tv, l);
                                          tv[l] = ' '; setval(tv); }
    inline void setlen (int l)          { tlen = l; }
    inline int  set (void* v)           { set((char*) v, len = tlen);
                                          return OK; }
    Bool        filter (int, void*);
    Bool        filterSubstring (int, void*, Bool);

    inline      OctetString (char* v, int l) { set(v, l); }
    inline      OctetString ()          { setval(strdup("")); len = 0; }

};
```

inline void set *(char* v, int l);*

The string *v* of length *l* is copied and stored.

inline void setlen *(int l);*

Sets the length of the string. This should be used in conjunction with **set** *(void* v)*.

inline int set *(void* v);*

This set current value to a copy of the string pointed to be *v* (`char *`). This should be used in conjunction with **setlen** *(int l)*.

inline OctetString *(char* v, int l);*

Instantiates an attribute instance and initilaises it with the values supplied. A copy of the string pointed to by *v* is made.

inline OctetString *();*

Instantiates an attribute instance.

5.3.1.1.4 String

```
     class String : public OctetString
{
   ...

public:
    inline int  set (void* val)         { OctetString::set((char*) val,
                                          strlen((char*) val)); return OK; }

    inline      String ()               {}
    inline      String (char* val)      : OctetString(val, strlen(val)) {}
};
```

inline void set *(void* val)*

The string pointed to be *val* is copied and stored. The string must NULL terminated.

inline String *();*

Instantiates an attribute instance.

inline String *(char* val);*

Instantiates an attribute instance and initialises it with values supplied. The string must be NULL terminated. A copy of the string pointed to by *val* is made.

5.3.1.1.5 DName

```
     class DName : public Attr
{
   ...

public:
    Bool        filter (int, void*);

    inline DName ()                     { setval(NULLDN); }
    inline DName (DN mydn)              { setval(mydn); }
};
```

inline DName *();*

Instantiates an attribute instance.

inline DName *(DN mydn);*

Instantiates an attribute instance with the value supplied. The user is responsible for allocating memory for the storage of *mydn*.

5.3.1.1.6 ObjId

```
     class ObjId : public Attr
{
   ...

public:
    Bool        filter (int, void*);

    inline ObjId ()                     { setval(NULLOID); }
    inline ObjId (OID myoid)            { setval(myoid); }
};
```

inline ObjId *();*

Instantiates an attribute instance with no value stored.

inline ObjId *(OID myoid);*

Instantiates an attribute instance with the value supplied. The user is responsible for allocating memory for the storage of *myoid*.

5.3.1.1.7 CounterThreshold

```
     class CounterThreshold : public Attr
{
   ...

public:
    int         add (int, int, Switch);
    int         add (void*);
    int         remove (void*);
    int         setDefault (void*);
    Bool        filter (int, void*);
    Switch      check (int, int);
    inline void associateCounter (Counter* cntr)
                                      { counter = cntr; }

    inline      CounterThreshold ()     { setval(NULL); counter = NULL; }
    inline      CounterThreshold (int level, int offset, Switch onoff)
                                      { add(level, offset, onoff);
                                        counter = NULL; }
};
```

int add *(int level, int offset, Switch onoff);*

Adds a threshold at *level* with an offset value of *offset* to switch to the value of *onoff*. Returns `OK` or `NOTOK`.

int add *(void* addThld);*

Add a threshold with the values provided in *addThld* (`CounterThresholdVal *`). The user is responsible for allocating memory for the storage of *addThld*. Returns `OK` or `NOTOK`.

int remove *(void* rmThld);*

Remove a threshold with the values provided in *rmThld* (`CounterThresholdVal *`). Returns `OK` or `NOTOK`.

int setDefault *(void* val);*

Set the default value of the threshold to the current value offset by *val* (`CounterThresholdVal *`). The value of the associated counter is checked and the offset given by *val* applied. The user is responsible for allocating memory for the storage of *val*. Returns `OK` or `NOTOK`.

Switch check *(int val, int prev);*

Check threshold levels by comparing current value, *val*, and previous value, *prev*.

inline void associateCounter *(Counter* cntr);*

This threshold will be applied to *cntr*. The user is responsible for allocating memory for the storage of *cntr*.

inline CounterThreshold *();*

Instantiate an attribute instance.

inline CounterThreshold *(int level, int offset, Switch onoff);*

Instantiate an attribute instance and initilaise with the value supplied.

5.3.1.1.8 GaugeThresholdInt and GaugeThresholdReal

These are derived from the base class :

```
class GaugeThreshold : public Attr
{
   ...

public:
    int         set (void*);
    int         add (void*);
    int         remove (void*);
    Bool        filter (int, void*);
    Switch      check (double, double);
```

```
    int         add (double, Switch, double, Switch);
};
```

int set *(void* newThld);*

Set the threshold to the value pointed to by *newThld* (`GaugeThresholdVal *`). The user is responsible for allocating memory for the storage of *newThld*. Returns `OK` or `NOTOK`.

int add *(void* addThld);*

Add to the current threshold the values provided in *addThld* (`GaugeThresholdVal *`). The user is responsible for allocating memory for the storage of *addThld*. Returns `OK` or `NOTOK`.

int remove *(void* rmThld);*

Remove from the current threshold the values provided in *addThld* (`GaugeThresholdVal *`). Returns `OK` or `NOTOK`.

Switch check *(double val, double prev);*

Check threshold levels by comparing current value, *val*, and previous value, *prev*.

int add *(double low, Switch onofflow, double high, Switch onoffhigh);*

Add to the current threshold, an element given by the values provided. Returns `OK` or `NOTOK`.

Two classes are provided for convenient instantiation :

```
class GaugeThresholdInt : public GaugeThreshold
{
public:
    inline      GaugeThresholdInt ()    : GaugeThreshold(OV_INTEGER) {}
    inline      GaugeThresholdInt (int low, Switch slow, int high, Switch
shigh)
                                        : GaugeThreshold(OV_INTEGER)
                                        { add(low, slow, high, shigh); }
};
```

```
class GaugeThresholdReal : public GaugeThreshold
{
public:
    inline      GaugeThresholdReal ()   : GaugeThreshold(OV_REAL) {}
    inline      GaugeThresholdReal (double low, Switch slow,
                                    double high, Switch shigh)
                                        : GaugeThreshold(OV_REAL)
                                        { add(low, slow, high, shigh); }
};
```

inline GaugeThresholdInt *();*

Instantiate an attribute instance for an `INTEGER` gauge.

inline GaugeThresholdInt *(int low, Switch slow, int high, Switch shigh);*

Instantiate an attribute instance for an `INTEGER` gauge, and initialise the value with those supplied.

inline GaugeThresholdReal *();*

Instantiate an attribute instance for a `REAL` gauge.

inline GaugeThresholdReal *(double low, Switch slow, double high, Switch shigh);*

Instantiate an attribute instance for a `Real` gauge, and initilaise values with those supplied.

5.3.1.1.9 TideMarkMin and TideMarkMax

These are derived from the base class :

```
class TideMark : public Attr
{
   ...

public:
    // the following is defined because setDefault is permitted
    // but not replace(set), add and remove
    inline int  set (void*)             { return NOTOK; }
```

```
    Switch      check (double);
    int         setDefault (void*);
    void        associateGauge (Gauge*);
};
```

inline int set *(void*);*

It is not possible to set the value of the tidemark - the tidemark monitors the value of an associated gauge.

Switch check *(double newVal);*

Check the tidemark against *newVal*.

int setDefault *(void *dflt);*

Set the default value to the current value of the associated gauge. *dflt* is not used, however, the user must provide a valid value for *dflt* for which memory has been allocated.

void associateGauge *(Gauge* g);*

Associate gauge *g* to this tidemark.

Two classes are provided for convenient instantiation :

```
class TideMarkMin : public TideMark {
public:
    inline TideMarkMin ()               : TideMark(TM_MIN) {}
};

class TideMarkMax : public TideMark {
public:
    inline TideMarkMax ()               : TideMark(TM_MAX) {}
};
```

inline TideMarkMin *();*

Instantiate an instance of a minimum tidemark.

inline TideMarkMax *();*

Instantiate an instance of a maximum tidemark.

5.3.1.1.10 Counter

```
     class Counter : public Attr
{
   ...

public:
    Bool        filter (int, void*);
    Switch      set (int);
    Switch      increment (int);
    inline int  set (void* val)         { set(*(int*) val); return OK; }
    inline Switch increment ()          { return increment(1); }
    inline void associateThreshold (CounterThreshold* thld)
                                        { (threshold = thld) ->
                                                associateCounter(this); }
    inline      Counter ()              { int val = 0; setval(int_cpy(&val));
                                          threshold = NULL; }
};
```

Switch set *(int newVal);*

Set the value to *newVal* and check the threshold.

Switch increment *(int incr);*

Increment the value by *incr* and check the threshold.

inline void set *(void* val);*

Set the value to that pointed to by *val* (`int`) and check the threshold. Returns `OK` or `NOTOK`.

inline Switch increment *();*

Increment the value by one and check the threshold.

inline void associateThreshold *(CounterThreshold* thld);*

Use the threshold provided by *thld*.

inline Counter *();*

Instantiate an attribute instance and allocate memory for storing the `int` to be used for the lifetime of the attribute. Sets the value to zero.

5.3.1.1.11 GaugeInt and GaugeReal

These are derived from the base class :

```
class Gauge : public Attr
{
   ...

public:
    inline int  getType ()              { return observedValue.ov_type; }
    inline void associateThreshold (GaugeThreshold* thld)
                                        { threshold = thld; }
    inline void associateTideMark (TideMark* tm)
                                        { (tideMark = tm) ->
                                                associateGauge(this); }
    inline Switch triggeredThreshold () { return thldTrigger; }
    inline Switch triggeredTideMark ()  { return tmrkTrigger; }

    Switch      set (double);
    inline int  set (void* val)         { set(*(double*) val); return OK; }

    Bool        filter (int, void*);
};
```

int getType *();*

The type of this gauge. Returns `OV_INTEGER` for **GaugeInt** or `OV_REAL` for **GaugeReal**.

inline void associateThreshold *(GaugeThresholdInt* thld);*

Use the threshold(s) provided by *thld*. The user is responsible for allocating memory for the storage of *thld*.

inline void associateTideMark *(TideMark* tm);*

Use the tidemark provided by *tm*. The user is responsible for allocating memory for the storage of *tm*.

inline Switch triggeredThreshold *();*

Test if the threshold is triggered.

inline Switch triggeredTideMark *();*

Test if the tidemark is triggered.

Switch set *(double newVal);*

Set this gauge to the value given by *newVal*, and check thresholds.

int set *(void* val);*

Set this gauge to the value pointed to by *val* (`double *`) and check thresholds. Returns `OK` or `NOTOK`.

Bool filter *(int, void*);*

Two classes are provided for convenient instantiation :

```
class GaugeInt : public Gauge
{
public:
    inline Switch increment (int incr)  { return set(*(int*) getval() + incr);
}
    inline Switch decrement (int decr)  { return set(*(int*) getval() - decr);
}
    inline Switch increment ()          { return increment(1); }
    inline Switch decrement ()          { return decrement(1); }
```

```
    inline      GaugeInt ()             : Gauge(OV_INTEGER)
                                        { int val = 0; setval(int_cpy(&val)); }
};
```

Switch increment *(int incr);*

Increment the value by *incr* and check the threshold.

Switch decrement *(int decr);*

Decrement the value by *decr* and check the threshold.

Switch increment *();*

Increment the value by 1 and check the threshold.

Switch decrement *();*

Decrement the value by 1 and check the threshold.

inline GaugeInt *();*

Instantiate an attribute instance.

```
class GaugeReal : public Gauge
{
public:
    inline Switch increment (double incr) { return set(*(double*) getval()
                                            + incr); }
    inline Switch decrement (double decr) { return set(*(double*) getval()
                                            - decr); }

    inline      GaugeReal ()            : Gauge(OV_REAL)
                                        { double val=0;
setval(real_cpy(&val));}
};
```

Switch increment *(double incr);*

Increment the value by *incr* and check the threshold.

Switch decrement *(double decr);*

Decrement the value by *decr* and check the threshold.

inline GaugeReal ()*;*

Instantiate an attribute instance.

5.3.1.1.12 Time

```
     class Time : public Attr
{
   ...
public:
    inline void* get ()                { if (!converted) convert();
                                         return getval(); }
    inline long getsec ()              { get(); return sectime; }
    inline UTC  getutc ()              { return (UTC) get(); }
    inline void set (long newtime)     { sectime = newtime;
                                         converted = False; }
    inline void set (UTC newtime)      { UTC ut = (UTC)getval(); *ut=*newtime;
#ifndef __BSDI__
                                         sectime = timelocal(ut2tm(newtime));
#else
                                         sectime = mktime(ut2tm(newtime));
#endif

                                         replval(ut); converted = True; }
    inline int  set (void* newtime)    { set((UTC) newtime); return OK; }
    inline void set ()                 { sectime = time((long*)0); // now
                                         converted = False; }
    Bool        filter (int, void*);

    inline Time ()                     { UTC ut = new UTCtime; setval(ut);
                                         set(); }
    inline Time (long mytime)          { UTC ut = new UTCtime; setval(ut);
                                         sectime = mytime; converted = False;}
};
```

inline void* get *();*

Returns the value in the C data type `UTCtime`.

inline long getsec *();*

Returns the value in the format described by `time(3)`.

inline UTC getutc *();*

Returns the value in the C data type `UTCtime`.

inline void set *(long newtime);*

Set the value to that given by *newtime*, which is `time(3)` format.

void set *(UTC newtime);*

Set the value to that given by *newtime*.

inline int set (void* newtime);

Set the value to that pointed to by *newtime* (`UTC`). Returns `OK` or `NOTOK`.

inline void set *();*

Set the value to that of the current system time as given by `time(3)`.

inline Time *();*

Instantiate an attribute instance and allocate memory for storing the `UTCtime` structure to be used for the lifetime of the attribute, and initialise attribute instance with a value of the current time as given by the system.

inline Time *(long mytime);*

Instantiate an attribute instance and allocate memory for storing the `UTCtime` structure to be used for the lifetime of the attribute. Sets the value to *mytime*.

5.3.1.1.13 Filter

```
     class Filter : public Attr
{
```

```
   ...

public:
    int set (void*);

    Filter ();
    inline Filter (CMISFilter* filter)  { setval(filter); }
};
```

int set *(void* val);*

Set value to that pointed to by *val* (`CMISFilter *`). The user is resposnible for allocating memory for the storage of *val*. Returns `OK` or `NOTOK`.

inline Filter *();*

Instantiate an attribute instance and initialise to a value that always returns "True" i.e. a NULL filter.

inline Filter *(CMISFilter* filter);*

Instantiate an attribute instance and set value to *filter*. The user is resposnible for allocating memory for the storage of *filter*.

5.3.1.1.14 ObjIdList

```
     class ObjIdList : public Attr
{
   ...

public:
    Bool        filter (int, void*);
    int         add (void*);

    inline ObjIdList ()                 { setval(NULL); }
    inline ObjIdList (OID oid)          { ObjIdListVal* v = new ObjIdListVal;
                                          v->ol_oid = oid; v->ol_next = NULL;
                                          setval(v); }
};
```

int add *(void* oid);*

Add *oid* (`OID`) to current value. The user is responsible for allocating memory for the storage of *oid*. Returns `OK` or `NOTOK`.

inline ObjIdList *();*

Instantiate an attribute instance.

inline ObjIdList *(OID oid);*

Instantiate an attribute instance. Set value to *oid*. The user is responsible for allocating memory for the storage of *oid*.

5.3.1.1.15 AdministrativeState

```
     class AdministrativeState : public Attr
{
   ...

public:
    inline void set (AdminStateVal newState)
                                       { AdminStateVal* state =
                                         (AdminStateVal*) getval();
                                         *state = newState; replval(state); }
    inline void set (void* val)        { set(*(AdminStateVal*) val); }
    inline int  setDefault ()          { AdminStateVal* state =
                                         (AdminStateVal*) getval();
                                         *state = as_unlocked;
                                         replval(state); return OK; }
    Bool        filter (int, void*);

    inline AdministrativeState ()      { AdminStateVal* state = new
                                         AdminStateVal; *state = as_unlocked;
                                         setval(state); makeSettable(); }
    inline AdministrativeState (AdminStateVal myState)
                                       { AdminStateVal* state =
                                         new AdminStateVal; *state = myState;
                                         setval(state); makeSettable(); }
};
```

inline void set *(AdminStateVal newState);*

Set value to that given by *newState*.

inline void set *(void* val);*

Set the value to that pointed to by *val* (`AdminStateVal`).

inline AdministrativeState *();*

Instantiate an attribute instance and allocate memory for storing the `AdminStateVal` to be used for the lifetime of the attribute. Sets the value to `as_unlocked`.

inline AdministrativeState *(AdminStateVal myState);*

Instantiate an attribute instance and allocate memory for storing the `AdminStateVal` to be used for the lifetime of the attribute. Sets the value to *myState*.

5.3.1.1.16 OperationalState

```
     class OperationalState : public Attr
{
   ...

public:
    inline void set (OperStateVal newState)
                                        { OperStateVal* state = (OperStateVal*)
                                          getval(); *state = newState;
                                          replval(state); }
    inline int  set (void* val)         { set(*(OperStateVal*) val);
                                          return OK; }
    Bool        filter (int, void*);

    inline OperationalState ()          { OperStateVal* state =
                                          new OperStateVal; *state=os_enabled;
                                          setval(state); }
    inline OperationalState (OperStateVal myState)
                                        { OperStateVal* state =
                                          new OperStateVal; *state = myState;
                                          setval(state); }
};
```

inline void set *(OperStateVal newState);*

Set value to that given by *newState*.

inline int set *(void* val);*

Set the value to that pointed to by *val* (`OperStateVal *`). Returns `OK` or `NOTOK`.

inline OperationalState *();*

Instantiate an attribute instance and allocate memory for storing the `OperStateVal` to be used for the lifetime of the attribute and sets the value to `os_enabled`.

inline OperationalState *(OperStateVal myState);*

Instantiate an attribute instance and allocate memory for storing the `OperStateVal` to be used for the lifetime of the attribute. Sets the value to *myState*.

5.3.1.1.17 DestinationAddress

```
     class DestinationAddress: public Attr
{
   ...

public:
    inline DestinationAddress ()
                                        { setval(NULL); }
    inline DestinationAddress (DestAddressVal* myAddr)
                                        { setval(myAddr); }
};
```

inline DestinationAddress *();*

Instantiate an attribute instance.

inline DestinationAddress *(DestAddressVal* myAddr);*

Instantiate an attribute instance and set value to *myAddr*. The user is responsible for allocating memory for the storage of *myAddr*.

5.3.1.1.18 LogFullAction

```
      class LogFullAction : public Attr {
   ...

public :

    void set(LogFullActionValue);
    int set(void *);

    Bool filter(int, void *);

    LogFullAction(); // default is lfa_wrap (0)
    LogFullAction(LogFullActionValue);
};
```

void set*(LogFullActionValue v);*

Set value to *v*.

int set(void *v);

Set value to v (`LogFullActionValue *`). Returns `OK` or `NOTOK`.

LogFullAction*();*

Instantiate an attribute instance and allocate memory for storing the `LogFullActionValue` to be used for the lifetime of the attribute. Sets the value to `lfa_warp`.

LogFullAction*(LogFullActionValue v);*

Instantiate an attribute instance and allocate memory for storing the `LogFullActionValue` to be used for the lifetime of the attribute. Sets the value to *v*.

5.3.1.1.19 AvailabilityStatus

```
      class AvailabilityStatus : public Attr {
   ...
```

```
public :

    int add(void *);
    int remove(void *);

    Bool filter(int, void *);

    int add(int);       // internal use only
    int remove(int);    // internal use only
    Bool isMember(int); // internal use only

    AvailabilityStatus(); // default is {} (empty set)
    AvailabilityStatus(AvailabilityStatusValue *);
};
```

int add*(void *v;)*

Add *v* (`AvailabilityStatusValue *`) to current value. Returns `OK` or `NOTOK`.

int remove*(void *v;)*

Remove *v* (`AvailabilityStatusValue *`) from current value. Returns `OK` or `NOTOK`.

int add*(int member);*

Add *member* to current value. Returns `OK` or `NOTOK`.

int remove*(int member);*

Remove *member* from current value. Returns `OK` or `NOTOK`.

Bool isMember*(int member);*

Tests if *member* is part of current value.

AvailabilityStatus*();*

Instantiate an attribute instance and initialise attribute instance with the value {} (empty set).

AvailabilityStatus*(AvailabilityStatusValue *v);*

Instantiate an attribute instance with the value *v*. The user is responsible for allocating memory for the storage of *v*.

5.3.2 Introducing your own syntaxes into the software

The implementation consists of two stages :

1. **The syntax routines** : The ASN.1 definition of a type must be represented in a form that is usable in a program, e.g. a C data structure, and also in a form that is suitable for communication, i.e. independent of both the programming language or machine architecture. Also, other routines may be required - to pretty-print the C structure, parse a human-readable, human-friendly string to produce a C structure, make a copy of the C structure, free memory allocated to the C structure or compare two instances of the C structure - to allow useful things operations to be performed with the information the C structure holds.
2. **The C++ class definition** : All attributes implementations must be of base class **Attr**. Some of the "generic" properties of attributes are already implemented in **Attr** and it remains for these properties to be enhanced and adjusted, according to requirements, through use of inheritance.

The C++ attribute implementation is used in MO implementations, i.e. the agent software. The syntax routines find use in both agents and managers.

5.3.2.1 Implementing the syntax routines

The syntax routines must be implemented to provide the interfaces described in Volume 5 of the ISODE Manual (Section 16.4, page 195). Every syntax **MUST** have an encode, a decode, a free and a print routine. If required, parse, copy and compare routines can also be implemented. Once the syntax is defined, the relevant files that include the syntax definition must be linked with every program that uses that definition.

The routines could be implemented with one of two methods: "hand-coded" or generated automatically from the ASN.1 using the **pepsy** tool from ISODE.

Using **pepsy** is normally a quick and fairly simple procedure, though the resulting C data structures often have "unfriendly" names and the routines generated may not be the most efficient. Also, **pepsy** does not generate the parse, copy or compare routines. More information on the use of **pepsy** can be found in Chapter 7 of Volume 4 of the ISODE Manual. After the syntax routines have been implemented, they must be incorporated into the software using the `add_attribute_syntax()`, as explained in Volume 5 of the ISODE Manual (Section 16.4, page 195).

Note : When using **pepsy** with ASN.1 that references/uses other ASN.1 types that should already be defined, e.g. from the msap library or dsap library, **pepsy** will probably fail. The data structures in the msap library and many from the dsap library were hand coded to provide user-friendly data structures and efficient syntax routines. The format of the data structures will probably be different to those that **pepsy** would expect them to be given the ASN.1 definition. If this situation occurs, then you must hand code all your syntax routines.

Hand-coding may take longer and require you to have more knowledge of the workings of PEs, but should result in a more "friendly" C data structure and a more efficient set of syntax routines.

5.3.2.2 The C++ class definition

The GMS assumes that the attribute implementations have some generic properties. This genericity is implemented by the C++ class **Attr**. All attribute implementations must have this class as their most base class.

This section describes how the syntax routines are incorporated into a C++ class definition, that can be used in the GMS. It assumes that reader is familiar with the implementation of syntax routines in ISODE (see previous section). The explanation will be given by way of example, using the fictitious syntax `MySyntax`, say, that is represented by a C type called `MySyntaxValue`. This representation is supported by the following syntax routines :

```
PE            encodeMySyntax(MySyntaxValue *v);

MySyntaxValue *decodeMySyntax(PE pe);

void          freeMySyntax(MySyntaxValue *v);

int           printMySyntax(PS ps, MySyntaxValue *v, int format);

MySyntaxValue *parseMySyntax(char *str);

int           copyMySyntax(MySyntaxValue *v);

int           compareMySyntax(MySyntaxValue *v1, MySyntaxValue *v2);
```

In the following text, *value* refers to the C data-type that represents the ASN.1 syntax and *attribute* is the instance of the C++ class definition that implements the attribute type - *value* is stored in *attribute*.

The remainder of this section is split into three. The first part describes the most basic requirements for implementing a C++ class that can be used by the GMS. The second part describes a slightly more complex implementation that makes more efficient use of memory,

and utilises some of the facilities offered by the GMS. The third part describes how to redefine the "CMIS interface" to the attribute if extremely sophisticated handling of the attribute value is required, for instance when it is required to report errors from a real resource that is represented by *value*.

5.3.2.2.1 Minimal requirements for the C++ attribute implementation

This simply requires that certain `protected virtual` methods of class **Attr** be redefined. The most basic requirements for a C++ implementation of the attribute type are as follows :

```
class MySyntax : public Attr {

protected :

   PE   _encode();
   void *_decode(PE pe);
   void _free();
   char *_print();

public :

   Bool filter(CMISFilterItem *);

   MySyntax() {}
};

inline PE
MySyntax::_encode()
{ return(encodeMySyntax((MySyntaxValue *) getval())); }

inline void *
MySyntax::_decode(PE pe)
{ return((void *) decodeMySyntax(pe)); }

inline void
MySyntax::free()
{ freeMySyntax((MySyntaxValue *) getval()); }

inline char *
MySyntax::_print()
{ return(attrv2str(getval(), (PRINT_FNX) printMySyntax)); }
```

The above C++ class definition would allow the use of `MySyntax` in the GMS. It is essential that the `_encode()`, `_decode()`, `_free()` and `_print()` be provided. In the above example, the constructor simple initiates an instance of the class with a NULL value. A *value* for the instance can be assigned by use of the public method `Attr::set(void *)`, e.g.

```
...

MySyntax      ms;
MySyntaxValue *msv1, *msv2;

msv1 = val1; /* msv1 is assigned a value value */
ms.set((void *) msv1);

...

msv2 = val2; /* msv2 is given a value */
ms.set((void *) msv1);

...
```

In this very simple definition, use of `Attr::set(void *)` is the only method by which assignments can be made. When `msv2` is assigned to `ms`, the memory used by `msv1` is freed by the GMS.

The `protected virtual` methods for `_copy()` and `_compare()` can be defined in a similar way, if required, but are currently not used by the GMS.

Although the definition of the `filter()` method is not shown, it is required so that CMIS filters can be applied to *value*. The method should be defined to cater for the possible assertions that may be made about *value*, e.g. less or equal, subset of, etc.

In this very simple implementation, the CMIS operations **add** and **remove** (as applied to set valued attributes) will not function properly. Facilities for dealing correctly with set valued objects exist in the GMS, and their use is described below.

5.3.2.2.2 A slightly more complex implementation

The GMS offers a flexible interface to class **Attr** that allows the implementor to access the facilities offered by the GMS as well as incorporate any proprietary needs.

For instance, `MySyntaxValue` represents a set valued attribute type and also wishes to avoid memory allocation/deallocation every time the *value* of *attribute* is changed. Also, the *attribute* instance represents a value that is taken from some real-resource and so *value* must be "refreshed" before it is read by the GMS, and changes made to *value* via CMIS must be echoed to the real resource.

For this, the following might be implemented :

```
class MySyntax : public Attr {

protected :

    PE   _encode();    // as before
    void *_decode(PE); // as before
    void _free();      // as before
    char *_print();    // as before

public :

    // Virtual methods from class Attr that will be redefined
    void *get();
    int  set(void *);
    int  add(void *);
    int  remove(void *);
    int  setDefault(void *);

    // CMIS methods
    Bool filter(int, void *); // as before

    // New methods for internal use
    Bool     isMember(MySyntaxElementValue *);
    MySyntax *intersection(MySyntax *);
    MySyntax *union(MySyntaxElement *);

    // Constructor
    MySyntax(MySyntaxValue *v = NULL);
```

```
};
```

and lets say that :

```
MySyntax ::= SET OF MySyntaxElement
```

and that `MySyntaxElement` is represented in C by a data-type called `MySyntaxElementValue`.

In the above definition, a lot more methods are defined. They shall be tackled in the way that they have been grouped above. The constructor first :

```
MySyntax::MySyntax(MySyntaxValue *v)
{ setval((void *) v); }
```

In this method, an initial can be supplied.

Now we shall deal with the redefined `virtual` functions.

```
void *
MySyntax::get()
{
    MySyntaxValue *v;

    /*
    ** Here, information from the real resource
    ** is retrieved and stored. v is updated.
    */

    v = valueFromRealResource;

    setval((void *) v);
```

```
    return((void *) v);
}
```

The call to `setval()` frees the old value an uses the new value `v`. In the next call to `setval()`, `v` will be freed, so `v` must be allocated storage space (e.g. using `malloc()`);

```
int
MySyntax::set(void *newV)
{
    Bool setOK = False;

    /*
    ** Here, newV is written to the real resource.
    ** if the write to the real resource was succesful, then setOK = True;
    ** v should be updated after interaction with real resource.
    */

    if (setOK)
                setval((void *) newV);

    return(setOK ? OK : NOTOK);
}
```

The set operation copies the new value to the real resource and also stores it in *attribute*, if the interaction with the real resource was succesful.

The definition of the methods `replace()`, `add()` and `remove()` would be along similar lines - `replace()` would replace the value of `v` (similar to set). For set valued operations `add()` would add a `MySyntaxElementValue` from `value` and `remove()` would remove a `MySyntaxElementValue` from `value`. `add()` and `remove()` return the manifest constants `OK` or `NOTOK`.

```
int MySyntax::setDefault(void *default)
```

```
{ setval(default); return(OK); }
```

This method sets *value* to `default`. The `default` value is stored by the GMS and is set-up when the object class containing this attribute instance is initiliased.

The implementor may also introduce some methods that are specific to this attribute. For instance, in this attribute we have :

```
Bool     isMember(MySyntaxElementValue *);
MySyntax *intersection(MySyntax *);
MySyntax *union(MySyntaxElement *);
```

which are for "internal" (i.e. non-CMIS) use.

5.3.2.2.3 The CMIS Interface to the attribute definition

So far, the interafce to the base class **Attr** has hidden as much as possible the underlying communication. However, there are two methods that should be pointed out.

CMISErrors Attr::get(PE);*

This is used by the agent software and can be ignored.

CMISErrors Attr::set(PE pe);

This function needs to be used, for instance, when creating a MO instance in response to a CMIS request. Use of ISODE means that attribute values are passed as ISODE Presentation Elements (PEs). The PEs arriving from the network are used directly to set the attribute values by passing the PEs to the attribute.

6. Generic Manager Support

TBA

7. Management Applications

TBA (see manual pages)

NAME

mibdump – OSI MIB retriever

SYNOPSIS

mibdump <agent> <host> [-c <class> [-i <instance>]
[-s <scope> [<sync>]] [-f <filter>]
[-a <attr> ...]]

DESCRIPTION

mibdump is a program that enables to connect to a a remote OSI management agent and retrieve management information. It establishes first a management association, requests the managed objects as specified through the command line arguments (using a M-GET CMIS request), pretty prints the output and/or errors and then releases the association and exits.

OPTIONS

The <agent> argument is mandatory and expresses the remote logical management application "qualifier". In OSIMIS there are two such application qualifiers, SMA for the ISO Transport and the example UNIX MIB agent and OIM-SMA for the OSI Internet MIB agent (OSI view of the SNMP MIB-II for TCP/IP, as described in RFC 1214).

The <host> argument specifies the name of the host where the agent runs - the management application "designator". It should be the name with which the host has been registered in the isoentities database or the global Directory. Note that for Internet hosts, this is usually the last component of its address e.g. athena and NOT athena.cs.ucl.ac.uk .

If no optional argument is given, the whole management information base is requested by assuming class=system, inst={}, scope=wholeSubtree sync=bestEffort and all attributes (an instance of class system with an empty local distinguished name is always the top object of OSI MIBs). This is an expensive operation and should better be avoided. If having done that it is realised that the remote MIB contains too many objects, the retrieval may be interrupted by typing CONTROL-C (^C) or CONTROL-\ (^\) on the terminal.

The ‘-c’ option selects the base object class. It should be used in conjunction with ‘-i’ for the instance (i.e. the object’s name). If it is set ‘-c system’, the -i option may be omitted as there is always only one instance of class system.

The ‘-i’ option selects the base object instance. This is a distinguished name in the ISODE string representation form and may be either a global or a local name, the latter omitting the relative name for the system object. For example,
systemId=athena@subsystemId=transport@entityId=isode and
subsystemId=transport@entityId=isode
are the global and local distinguished names respectively for a transport protocol entity managing the ISODE implementation of the latter. Local names are shorter and useful as there is no need to know in advance the relative name of the system object.

Having selected the base object, the operation may be applied to many objects using scoping/synchronisation and filtering. The ‘-s’ option selects the scope and possibly the synchronisation. The scope may be an individual level e.g. *1stLevel*, *2stLevel* etc., all the objects until a particular level e.g. baseTo3rdLevel or the whole subtree e.g. *wholeSubtree*. By specifying *baseObject*, *0thLevel* or *baseTo0thLevel* the scope effect is nullified.

The synchronisation of the operations across the multiple scoped objects may be requested to be atomic, in which case either all operations should succeed or none should be performed - in the latter case an empty reply is returned. Synchronisation is only meaningful when more than one objects have been scoped. Atomic synchronisation can be requested using *atomic* e.g. ‘-s 1stLevel atomic’. You must bear in mind that atomic synchronisation is not supported by most agents, in which case a *synchronisationNotSupported* error will be returned. The default synchronisation is *bestEffort*.

The ‘-f’ option selects the filter to be applied to the scoped objects. This enables to perform the operation only on these objects for which the filter expression evaluates to true. Note that a filter may be also used without scoping. A filter expression may contain assertions on the value of managed object attributes. See

the FILTER EXPRESSIONS section for details on their grammar/construction.

The '-a' option may be used to specify the attribute names for which values should be retrieved. If no attributes are specified, all the attributes of each object are requested. By specifying '-a none', no attributes are requested i.e. empty objects are received - this is may be used to check which objects are present.

If one or more of the requested attributes does not exist for a selected object instance, a *noSuchAttribute* error is returned. Also if the user has no read access rights for a particular attribute/object, a *accessDenied* error is returned. These are partial errors (getListError) as any correctly specified attributes will be received. The fact that inexistent attributes for an object class may be requested and still receive a partial result can be exploited to combine attributes of different classes in one get operation.

The following errors may occur with respect to the base object in which case no result is returned:

noSuchObjectClass when the specified object class is unknown by the agent

noSuchObjectInstance when the specified object instance is unknown by the agent

classInstanceConflict when the specified object instance does not belong to the specified class

syncNotSupported when the agent does not support atomic operations

processingFailure when a general error has occurred

FILTER EXPRESSIONS

A filter expression is used to construct a CMISFilter value. A CMISFilter contains attribute value assertions (AVAs) that are grouped with the logical operators AND, OR and NOT. The filter expression syntax is as follows : (<cmisfilter>) where <cmisfilter> is one of <notfilter>, <andfilter>, <orfilter> or <filteritem>.

The characters used to represent the logical operators are :

Character	**Operator**
"!"	NOT
"&"	AND
"|"	OR

A <notfilter> has the form : (!(<cmisfilter>)) A <andfilter> has the form : ((<cmisfilter>) & (<cmisfilter>) ...) A <orfilter> has the form : ((<cmisFilter>) | (<cmisfilter>) ...) A <filteritem> has one of the two forms : (<attributename>) for creating a CMISFilter item with the assertion test for "present", or (<attributename> <assertiontype> <attributevalue>) for the other assertion types :

Character	**Assertion type**
"="	equality
":="	substrings
">="	greater or equal
"<="	less or equal
":<"	subset of
":>"	superset of
"><"	non-null intersection

With the substrings operator, the character "*" can be used as a wild card.

Some examples of filter expressions : "((objectClass = eventRecord) & (eventType = linkUpEvent))" "((objectClass = log) & (!(administrativeState = unlocked)))" "((objectClass = log) & ((logId <= 2) | (logId >= 10)))" "((wiseSaying := *hello*) | (!(wiseSaying = bye)))" A "NULL" filter (one that always evaluates to true) can be created using : "(NULL)"

(This is actually an empty AND filter.)

The use of the brackets, "(" and ")", is very important, as they are used to delimit the strings used to represent the components of the filter. Also, please enclose your filter expression in quote marks, as in the examples above, so that the your UNIX shell does not interpret special characters such as "!", "|", "&", ">", "<", "(" and ")".

Please note that the string representation of attribute values is determined by the print and parse methods for the particular ASN.1 syntax. The OSIMIS convention for set valued attributes is that they should be enclosed in angular brackets and items should be separated by a "%". Note that set-valued attributes should also be enclosed in double quotes to avoid the special interpretation of those characters by the UNIX shell. Examples of attribute values are:

foobar - string
"Its always easy if you’re told how" - string
"{ Low: 5 Switch: On High: 7 Switch: On }" - gaugeThreshold
"{ Level:10 Offset:5 Switch:On %
Level:20 Offset:2 Switch:On }" - counterThreshold

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the correct OSIMIS ETC directory before starting this program.

FILES

$(ETC)/oidtable.[gen,at] - object identifier and syntax information.

$(ETC)/isoentities - PSAP address information of management agents.

DIAGNOSTICS

Should be obvious.

SEE ALSO

mset(1C), maction(1C), mcreate(1C), mdelete(1C)

NOTES

The abstract syntax of a CMISFilter is given in *ISO 9596 : "Information Technology - Open Systems Interconnection - Common Management Information Protocol specification"*.

There is currently no access control implemented in OSIMIS which means that all management information is visible. In the future, user authentication information may be needed in addition to the other arguments and the accessDenied error will be returned for attributes/objects for which the user has no read access rights.

BUGS

The filter expression parser is a bit shaky! You must have the correct number of brackets, as it keeps a count! For instance the filter expression below is missing the outermost set of brackets : "(eventType = linkUpEvent) | (eventType = linkDownEvent)" but instead of reading it as a badly formed OR filter, the parser reads it as the <filteritem> : "(eventType = linkUpEvent)" Additionally, superfluous brackets, e.g : ((eventType = linkUpEvent)) will cause it to fail.

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

mset – allows to set management attributes in OSI MIBs

SYNOPSIS

mset <agent> <host> -c <class> [-i <instance>]
[-s <scope> [<sync>]] [-f <filter>]
[-w|a|r|d] <attrType[=<attrValue>] ...

DESCRIPTION

mset is a program that enables to connect to a a remote OSI management agent and set management information (managed object attributes). It establishes first a management association, requests the set operation to be performed as specified through the command line arguments (using a M-SET CMIS request), pretty prints the results and/or errors and then releases the association and exits.

OPTIONS

The <agent> argument is mandatory and expresses the remote logical management application "qualifier". In OSIMIS there are two such application qualifiers, SMA for the ISO Transport and the example UNIX MIB agent and OIM-SMA for the OSI Internet MIB agent (OSI view of the SNMP MIB-II for TCP/IP, as described in RFC 1214).

The <host> argument specifies the name of the host where the agent runs - the management application "designator". It should be the name with which the host has been registered in the isoentities database or the global Directory. Note that for Internet hosts, this is usually the last component of its address e.g. athena and NOT athena.cs.ucl.ac.uk .

The ‘-c’ option selects the base object class. It should be used in conjunction with ‘-i’ for the instance (i.e. the object’s name). If it is set ‘-c system’, the -i option may be omitted as there is always only one instance of class system.

The ‘-i’ option selects the base object instance. This is a distinguished name in the ISODE string representation form and may be either a global or a local name, the latter omitting the relative name for the system object. For example,
systemId=athena@subsystemId=transport@entityId=isode and
subsystemId=transport@entityId=isode
are the global and local distinguished names respectively for a transport protocol entity managing the ISODE implementation of the latter. Local names are shorter and useful as there is no need to know in advance the relative name of the system object.

Having selected the base object, the operation may be applied to many objects using scoping/synchronisation and filtering. The ‘-s’ option selects the scope and possibly the synchronisation. The scope may be an individual level e.g. *1stLevel*, *2stLevel* etc., all the objects until a particular level e.g. baseTo3rdLevel or the whole subtree e.g. *wholeSubtree*. By specifying *baseObject*, *0thLevel* or *baseTo0thLevel* the scope effect is nullified.

The synchronisation of the operations across the multiple scoped objects may be requested to be atomic, in which case either all operations should succeed or none should be performed - in the latter case an empty reply is returned. Synchronisation is only meaningful when more than one objects have been scoped. Atomic synchronisation can be requested using *atomic* e.g. ‘-s 1stLevel atomic’. You must bear in mind that atomic synchronisation is not supported by most agents, in which case a *synchronisationNotSupported* error will be returned. The default synchronisation is *bestEffort*.

The ‘-f’ option selects the filter to be applied to the scoped objects. This enables to perform the operation only on these objects for which the filter expression evaluates to true. Note that a filter may be also used without scoping. A filter expression may contain assertions on the value of managed object attributes. See the FILTER EXPRESSIONS section for details on their grammar/construction.

The ‘-w’, ‘-a’, ‘-r’ and ‘-d’ options should be used to specify the attribute names and the corresponding values to be set. The difference is the set mode, which is the following for each option:

Option	Mode	Comments
-w	w-rite	(replace/set)
-a	a-dd value	(for set-valued attributes)
-r	r-emove value	(for set-valued attributes)
-d	d-efault set	(no value required)

After each of these options, there should be an attribute value assertion of the form <attrType>=<attrValue> with the exception of -d after which there should simply be an attribute type (value not required for set to default). The attribute value format is determined by the print and parse methods for a particular ASN.1 syntax. The OSIMIS convention for set valued attributes is that they should be enclosed in angular brackets and items should be separated by a "%". Note that set-valued attributes should also be enclosed in double quotes to avoid the special interpretation of those characters by the UNIX shell. Note that the convention for the value part is the same in filtering expressions (see below). Some examples of attribute value assertions:

```
wiseSaying=foobar
wiseSaying      = "Its always easy if you're told how"
nUsersThreshold = "{ Low: 5 Switch: On High: 7 Switch: On }"
pdusResentThld  = "{ Level:10 Offset:5 Switch:On %
                   Level:20 Offset:2 Switch:On }"
```

If one or more of the requested attributes does not exist for a selected object instance, a *noSuchAttribute* error is returned. If the user has no write access rights for a particular attribute/object or if an attribute is not settable a *accessDenied* error is returned. If the operation is meaningless, e.g. an add or remove operation to a non set-valued attribute, a *invalidOperation* error is returned. Finally if the attribute value is malformed or out of range, a *invalidAttributeValue* is returned. All these are partial errors in the sense that correctly specified operations will be performed. The fact that inexistent attributes for an object class may be requested to be set and still receive a partial result can be exploited to combine attributes of different classes in one operation.

The following errors may occur with respect to the base object in which case no result is returned:

noSuchObjectClass when the specified object class is unknown by the agent

noSuchObjectInstance when the specified object instance is unknown by the agent

classInstanceConflict when the specified object instance does not belong to the specified class

syncNotSupported when the agent does not support atomic operations

processingFailure when a general error has occurred

FILTER EXPRESSIONS

A filter expression is used to construct a CMISFilter value. A CMISFilter contains attribute value assertions (AVAs) that are grouped with the logical operators AND, OR and NOT. The filter expression syntax is as follows : (<cmisfilter>) where <cmisfilter> is one of <notfilter>, <andfilter>, <orfilter> or <filteritem>.

The characters used to represent the logical operators are :

Character	Operator
"!"	NOT
"&"	AND
"|"	OR

A <notfilter> has the form : (!(<cmisfilter>)) A <andfilter> has the form : ((<cmisfilter>) & (<cmisfilter>) ...) A <orfilter> has the form : ((<cmisFilter>) | (<cmisfilter>) ...) A <filteritem> has one of the two forms : (<attributename>) for creating a CMISFilter item with the assertion test for "present", or (<attributename> <assertiontype> <attributevalue>) for the other assertion types :

Character	Assertion type
"="	equality
":="	substrings

">="	greater or equal
"<="	less or equal
":<"	subset of
":>"	superset of
"><"	non-null intersection

With the substrings operator, the character "*" can be used as a wild card.

Some examples of filter expressions : "((objectClass = eventRecord) & (eventType = linkUpEvent))" "((objectClass = log) & (!(administrativeState = unlocked)))" "((objectClass = log) & ((logId <= 2) | (logId >= 10)))" "((wiseSaying := *hello*) | (!(wiseSaying = bye)))" A "NULL" filter (one that always evaluates to true) can be created using : "(NULL)"

(This is actually an empty AND filter.)

The use of the brackets, "(" and ")", is very important, as they are used to delimit the strings used to represent the components of the filter. Also, please enclose your filter expression in quote marks, as in the examples above, so that the your UNIX shell does not interpret special characters such as "!", "|", "&", ">", "<", "(" and ")".

For the OSIMIS convention on the string representation of attribute values, see above the discussion in the '-w|a|r' option regarding the value assertions.

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the correct OSIMIS ETC directory before starting this program.

FILES

$(ETC)/oidtable.[gen,at] - object identifier and syntax information.

$(ETC)/isoentities - PSAP address information of management agents.

DIAGNOSTICS

Should be obvious.

SEE ALSO

mibdump(1C), maction(1C), mcreate(1C), mdelete(1C), sma(8C), oimsma(8C)

NOTES

The abstract syntax of a CMISFilter is given in *ISO 9596 : "Information Technology - Open Systems Interconnection - Common Management Information Protocol specification"*.

NOTE WELL

There is currently no access control implemented in OSIMIS. Set operations are allowed to management control managed objects such as event discriminators and logs to control the level of event reporting and logging. In the OIM-SMA, sets are also allowed to routing tables when the remote agent runs as root. Note that this is very dangerous but the facility was added to show how intrusive management may be performed. If you allow the OIM-SMA to run as root, know you are taking risks!

In the future when access control is implemented, user authentication information will be needed in addition to the other arguments and the accessDenied error will be returned for attributes/objects for which the user has no write access rights.

BUGS

The filter expression parser is a bit shaky! You must have the correct number of brackets, as it keeps a count! For instance the filter expression below is missing the outermost set of brackets : "(eventType = linkUpEvent) | (eventType = linkDownEvent)" but instead of reading it as a badly formed OR filter, the parser reads it as the <filteritem> : "(eventType = linkUpEvent)" Additionally, superfluous brackets, e.g : ((eventType = linkUpEvent)) will cause it to fail.

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

maction – allows to perform a management action on OSI MIBs

SYNOPSIS

maction <agent> <host> -c <class> [-i <instance>]
[-s <scope> [<sync>]] [-f <filter>]
-a <actionType[=<actionValue>]

DESCRIPTION

maction is a program that enables to connect to a a remote OSI management agent and perform an action on managed objects. It establishes first a management association, requests the action operation to be performed as specified through the command line arguments (using a M-ACTION CMIS request), pretty prints the results and/or errors and then releases the association and exits.

OPTIONS

The <agent> argument is mandatory and expresses the remote logical management application "qualifier". In OSIMIS there are two such application qualifiers, SMA for the ISO Transport and the example UNIX MIB agent and OIM-SMA for the OSI Internet MIB agent (OSI view of the SNMP MIB-II for TCP/IP, as described in RFC 1214).

The <host> argument specifies the name of the host where the agent runs - the management application "designator". It should be the name with which the host has been registered in the isoentities database or the global Directory. Note that for Internet hosts, this is usually the last component of its address e.g. athena and NOT athena.cs.ucl.ac.uk .

The ‘-c’ option selects the base object class. It should be used in conjunction with ‘-i’ for the instance (i.e. the object’s name). If it is set ‘-c system’, the -i option may be omitted as there is always only one instance of class system.

The ‘-i’ option selects the base object instance. This is a distinguished name in the ISODE string representation form and may be either a global or a local name, the latter omitting the relative name for the system object. For example,
systemId=athena@subsystemId=transport@entityId=isode and
subsystemId=transport@entityId=isode
are the global and local distinguished names respectively for a transport protocol entity managing the ISODE implementation of the latter. Local names are shorter and useful as there is no need to know in advance the relative name of the system object.

Having selected the base object, the operation may be applied to many objects using scoping/synchronisation and filtering. The ‘-s’ option selects the scope and possibly the synchronisation. The scope may be an individual level e.g. *1stLevel*, *2stLevel* etc., all the objects until a particular level e.g. baseTo3rdLevel or the whole subtree e.g. *wholeSubtree*. By specifying *baseObject*, *0thLevel* or *baseTo0thLevel* the scope effect is nullified.

The synchronisation of the operations across the multiple scoped objects may be requested to be atomic, in which case either all operations should succeed or none should be performed - in the latter case an empty reply is returned. Synchronisation is only meaningful when more than one objects have been scoped. Atomic synchronisation can be requested using *atomic* e.g. ‘-s 1stLevel atomic’. You must bear in mind that atomic synchronisation is not supported by most agents, in which case a *synchronisationNotSupported* error will be returned. The default synchronisation is *bestEffort*.

The ‘-f’ option selects the filter to be applied to the scoped objects. This enables to perform the operation only on these objects for which the filter expression evaluates to true. Note that a filter may be also used without scoping. A filter expression may contain assertions on the value of managed object attributes. See the FILTER EXPRESSIONS section for details on their grammar/construction.

The ‘-a’ option should be used to specify the action type and value, the latter being optional. In the latter case, only the attribute type needs to be specified. When both the attribute type and value are needed, an attribute value assertion of the form <actionType>=<actionValue> should be given. The action value format is determined by the print and parse methods for a particular ASN.1 syntax. The OSIMIS convention

for set valued actions is that they should be enclosed in angular brackets. In the latter case they should be also enclosed in double quotes to avoid the special interpretation of those characters by the UNIX shell. Though the following are attribute rather than action value assertions (no OSIMIS MIB has any actions defined yet), they are given as examples. Note that the convention for the value part is the same for filtering expressions (see below):

```
wiseSaying=foobar
wiseSaying      = "Its always easy if you're told how"
nUsersThreshold = "{ Low: 5 Switch: On High: 7 Switch: On }"
pdusResentThld  = "{ Level:10 Offset:5 Switch:On %
                     Level:20 Offset:2 Switch:On }"
```

If the action does not exist for a selected object instance, a *noSuchAction* error is returned. If the action value is malformed or otherwise inappropriate, *invalidArgumentValue is returned.* The following errors may occur with respect to the base object in which case no result is returned:

noSuchObjectClass when the specified object class is unknown by the agent

noSuchObjectInstance when the specified object instance is unknown by the agent

classInstanceConflict when the specified object instance does not belong to the specified class

syncNotSupported when the agent does not support atomic operations

processingFailure when a general error has occurred

FILTER EXPRESSIONS

A filter expression is used to construct a CMISFilter value. A CMISFilter contains attribute value assertions (AVAs) that are grouped with the logical operators AND, OR and NOT. The filter expression syntax is as follows : (<cmisfilter>) where <cmisfilter> is one of <notfilter>, <andfilter>, <orfilter> or <filteritem>.

The characters used to represent the logical operators are :

Character	**Operator**
"!"	NOT
"&"	AND
"|"	OR

A <notfilter> has the form : (!(<cmisfilter>)) A <andfilter> has the form : ((<cmisfilter>) & (<cmisfilter>) ...) A <orfilter> has the form : ((<cmisFilter>) | (<cmisfilter>) ...) A <filteritem> has one of the two forms : (<attributename>) for creating a CMISFilter item with the assertion test for "present", or (<attributename> <assertiontype> <attributevalue>) for the other assertion types :

Character	**Assertion type**
"="	equality
":="	substrings
">="	greater or equal
"<="	less or equal
":<"	subset of
":>"	superset of
"><"	non-null intersection

With the substrings operator, the character "*" can be used as a wild card.

Some examples of filter expressions : "((objectClass = eventRecord) & (eventType = linkUpEvent))" "((objectClass = log) & (!(administrativeState = unlocked)))" "((objectClass = log) & ((logId <= 2) | (logId >= 10)))" "((wiseSaying := *hello*) | (!(wiseSaying = bye)))" A "NULL" filter (one that always evaluates to true) can be created using : "(NULL)"

(This is actually an empty AND filter.)

The use of the brackets, "(" and ")", is very important, as they are used to delimit the strings used to represent the components of the filter. Also, please enclose your filter expression in quote marks, as in the examples above, so that the your UNIX shell does not interpret special characters such as "!", "|", "&", ">", "<", "(" and ")".

For the OSIMIS convention on the string representation of attribute values, see above the discussion in the '-a' option regarding the value assertions.

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the correct OSIMIS ETC directory before starting this program.

FILES

$(ETC)/oidtable.[gen,at] - object identifier and syntax information.

$(ETC)/isoentities - PSAP address information of management agents.

DIAGNOSTICS

Should be obvious.

SEE ALSO

mibdump(1C), maction(1C), mcreate(1C), mdelete(1C), sma(8C), oimsma(8C)

NOTES

The abstract syntax of a CMISFilter is given in *ISO 9596 : "Information Technology - Open Systems Interconnection - Common Management Information Protocol specification"*.

There is currently no access control implemented in OSIMIS which means that all management actions will be attempted. In the future, user authentication information will be needed in addition to the other arguments and the accessDenied error will be returned for objects for which the user has no action (write) access rights.

BUGS

The filter expression parser is a bit shaky! You must have the correct number of brackets, as it keeps a count! For instance the filter expression below is missing the outermost set of brackets : "(eventType = linkUpEvent) | (eventType = linkDownEvent)" but instead of reading it as a badly formed OR filter, the parser reads it as the <filteritem> : "(eventType = linkUpEvent)" Additionally, superfluous brackets, e.g : ((eventType = linkUpEvent)) will cause it to fail.

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

mcreate – allows to create managed objects in OSI MIBs

SYNOPSIS

mcreate <agent> <host> -c <class>
[-i <instance> | -s <superiorInst>] [-r referenceInst]
[-a <attrType=<attrValue>] ...

DESCRIPTION

mcreate is a program that enables to connect to a a remote OSI management agent and create a managed object in its MIB. It establishes first a management association, requests the create operation to be performed as specified through the command line arguments (using a M-CREATE CMIS request), pretty prints the result and/or error and then releases the association and exits.

OPTIONS

The <agent> argument is mandatory and expresses the remote logical management application "qualifier". In OSIMIS there are two such application qualifiers, SMA for the ISO Transport and the example UNIX MIB agent and OIM-SMA for the OSI Internet MIB agent (OSI view of the SNMP MIB-II for TCP/IP, as described in RFC 1214).

The <host> argument specifies the name of the host where the agent runs - the management application "designator". It should be the name with which the host has been registered in the isoentities database or the global Directory. Note that for Internet hosts, this is usually the last component of its address e.g. athena and NOT athena.cs.ucl.ac.uk .

The ‘-c’ option specifies the class of the object to be created.

None or one only of the ‘-i’ or ‘-s’ options should be used to specify the object or its the (parent in the containment tree) instance i.e. its name. This is a distinguished name in the ISODE string representation form and may be either a global or a local name, the latter omitting the relative name for the system object. For example,
systemId=athena@subsystemId=transport@entityId=isode and
subsystemId=transport@entityId=isode
are the global and local distinguished names respectively for a transport protocol entity managing the ISODE implementation of the latter. Local names are shorter and useful as there is no need to know in advance the relative name of the system object.

No object or superior instance is needed when automatic instance naming is used by the agent and there is only one possible name binding for that object class e.g. for event discriminators and logs. If automatic instance naming is used but there are more than one name bindings, the superior instance needs to be specified to determine the specific binding to be used. Finally, the actual object instance is needed for non-automatic instance naming as it will contain the relative distinguished name for the object to be created.

The ‘-r’ option may be used specify a reference object instance to be used for initial attribute values to be copied.

The ‘-a’ option should be used to specify initial attribute values. Attribute values can ve specified through attribute value assertions of the form <attrType>=<attrValue>. The attribute value format is determined by the print and parse methods for a particular ASN.1 syntax. The OSIMIS convention for set valued attributes is that they should be enclosed in angular brackets. In the latter case they should be also enclosed in double quotes to avoid the special interpretation of those characters by the UNIX shell. Some examples of attribute value assertions are:

```
wiseSaying=foobar
wiseSaying      = "Its always easy if you’re told how"
nUsersThreshold = "{ Low: 5 Switch: On High: 7 Switch: On }"
pdusResentThld  = "{ Level:10 Offset:5 Switch:On %
                     Level:20 Offset:2 Switch:On }"
```

In case of no error, all the attribute values of the newly created object are displayed. The following errors may be returned:

noSuchObjectClass when the specified object class is unknown by the agent

duplicateManagedObjectInstance when the object instance already exists

noSuchReferenceObject when the reference instance does not exist

accessDenied when an instance of the specified class cannot be created through CMIS or when the user has no access rights for object creation (if access control is in use)

invalidObjectInstance when the superior instance does not exist or if a full instance is specified for a class with automatic instance naming and violates the naming rules

missingAttributeValue when a necessary initial attribute value was not specified e.g. the destination for event discriminators

invalidAttributeValue when a specified initial value was malformed or inappropriate

processingFailure when a general error has occurred.

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the correct OSIMIS ETC directory before starting this program.

FILES

$(ETC)/oidtable.[gen,at] - object identifier and syntax information.

$(ETC)/isoentities - PSAP address information of management agents.

DIAGNOSTICS

Should be obvious.

SEE ALSO

mibdump(1C), mset(1C), maction(1C), mdelete(1C), sma(8C), oimsma(8C)

NOTES

There is currently no access control implemented in OSIMIS which means that all management creation operations will be attempted. In the future, user authentication information will be needed in addition to the other arguments and the accessDenied error will be returned for objects for which the user has no action (write) access rights.

BUGS

The filter expression parser is a bit shaky! You must have the correct number of brackets, as it keeps a count! For instance the filter expression below is missing the outermost set of brackets : "(eventType = linkUpEvent) | (eventType = linkDownEvent)" but instead of reading it as a badly formed OR filter, the parser reads it as the <filteritem> : "(eventType = linkUpEvent)" Additionally, superfluous brackets, e.g : ((eventType = linkUpEvent)) will cause it to fail.

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

mdelete – allows to delete managed objects from OSI MIBs

SYNOPSIS

mdelete <agent> <host> -c <class> [-i <instance>]
[-s <scope> [<sync>]] [-f <filter>]

DESCRIPTION

mdelete is a program that enables to connect to a a remote OSI management agent and delete managed objects in its MIB. It establishes first a management association, requests the action operation to be performed as specified through the command line arguments (using a M-DELETE CMIS request), pretty prints the results and/or errors and then releases the association and exits.

OPTIONS

The <agent> argument is mandatory and expresses the remote logical management application "qualifier". In OSIMIS there are two such application qualifiers, SMA for the ISO Transport and the example UNIX MIB agent and OIM-SMA for the OSI Internet MIB agent (OSI view of the SNMP MIB-II for TCP/IP, as described in RFC 1214).

The <host> argument specifies the name of the host where the agent runs - the management application "designator". It should be the name with which the host has been registered in the isoentities database or the global Directory. Note that for Internet hosts, this is usually the last component of its address e.g. athena and NOT athena.cs.ucl.ac.uk .

The ‘-c’ option selects the base object class. It should be used in conjunction with ‘-i’ for the instance (i.e. the object’s name). If it is set ‘-c system’, the -i option may be omitted as there is always only one instance of class system.

The ‘-i’ option selects the base object instance. This is a distinguished name in the ISODE string representation form and may be either a global or a local name, the latter omitting the relative name for the system object. For example,
systemId=athena@subsystemId=transport@entityId=isode and
subsystemId=transport@entityId=isode
are the global and local distinguished names respectively for a transport protocol entity managing the ISODE implementation of the latter. Local names are shorter and useful as there is no need to know in advance the relative name of the system object.

Having selected the base object, the operation may be applied to many objects using scoping/synchronisation and filtering. The ‘-s’ option selects the scope and possibly the synchronisation. The scope may be an individual level e.g. *1stLevel*, *2stLevel* etc., all the objects until a particular level e.g. baseTo3rdLevel or the whole subtree e.g. *wholeSubtree*. By specifying *baseObject*, *0thLevel* or *baseTo0thLevel* the scope effect is nullified.

The synchronisation of the operations across the multiple scoped objects may be requested to be atomic, in which case either all operations should succeed or none should be performed - in the latter case an empty reply is returned. Synchronisation is only meaningful when more than one objects have been scoped. Atomic synchronisation can be requested using *atomic* e.g. ‘-s 1stLevel atomic’. You must bear in mind that atomic synchronisation is not supported by most agents, in which case a *synchronisationNotSupported* error will be returned. The default synchronisation is *bestEffort*.

The ‘-f’ option selects the filter to be applied to the scoped objects. This enables to perform the operation only on these objects for which the filter expression evaluates to true. Note that a filter may be also used without scoping. A filter expression may contain assertions on the value of managed object attributes. See the FILTER EXPRESSIONS section for details on their grammar/construction.

If the object cannot be deleted, a *accessDenied* error is returned. The following errors may occur with respect to the base object in which case no result is returned:

noSuchObjectClass when the specified object class is unknown by the agent

noSuchObjectInstance when the specified object instance is unknown by the agent

classInstanceConflict when the specified object instance does not belong to the specified class

syncNotSupported when the agent does not support atomic operations

processingFailure when a general error has occurred

FILTER EXPRESSIONS

A filter expression is used to construct a CMISFilter value. A CMISFilter contains attribute value assertions (AVAs) that are grouped with the logical operators AND, OR and NOT. The filter expression syntax is as follows : (<cmisfilter>) where <cmisfilter> is one of <notfilter>, <andfilter>, <orfilter> or <filteritem>.

The characters used to represent the logical operators are :

Character	**Operator**
"!"	NOT
"&"	AND
"|"	OR

A <notfilter> has the form : (!(<cmisfilter>)) A <andfilter> has the form : ((<cmisfilter>) & (<cmisfilter>) ...) A <orfilter> has the form : ((<cmisFilter>) | (<cmisfilter>) ...) A <filteritem> has one of the two forms : (<attributename>) for creating a CMISFilter item with the assertion test for "present", or (<attributename> <assertiontype> <attributevalue>) for the other assertion types :

Character	**Assertion type**
"="	equality
":="	substrings
">="	greater or equal
"<="	less or equal
":<"	subset of
":>"	superset of
"><"	non-null intersection

With the substrings operator, the character "*" can be used as a wild card.

Some examples of filter expressions : "((objectClass = eventRecord) & (eventType = linkUpEvent))" "((objectClass = log) & (!(administrativeState = unlocked)))" "((objectClass = log) & ((logId <= 2) | (logId >= 10)))" "((wiseSaying := *hello*) | (!(wiseSaying = bye)))" A "NULL" filter (one that always evaluates to true) can be created using : "(NULL)"

(This is actually an empty AND filter.)

The use of the brackets, "(" and ")", is very important, as they are used to delimit the strings used to represent the components of the filter. Also, please enclose your filter expression in quote marks, as in the examples above, so that the your UNIX shell does not interpret special characters such as "!", "|", "&", ">", "<", "(" and ")".

Please note that the string representation of attribute values is determined by the print and parse methods for the particular ASN.1 syntax. The OSIMIS convention for set valued attributes is that they should be enclosed in angular brackets and items should be separated by a "%". Note that set-valued attributes should also be enclosed in double quotes to avoid the special interpretation of those characters by the UNIX shell. Examples of attribute values are:

foobar - string
"Its always easy if you’re told how" - string
"{ Low: 5 Switch: On High: 7 Switch: On }" - gaugeThreshold
"{ Level:10 Offset:5 Switch:On %
Level:20 Offset:2 Switch:On }" - counterThreshold

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the correct OSIMIS ETC directory before starting this program.

FILES

$(ETC)/oidtable.[gen,at] - object identifier and syntax information.

$(ETC)/isoentities - PSAP address information of management agents.

DIAGNOSTICS

Should be obvious.

SEE ALSO

mibdump(1C), mset(1C), maction(1C), mcreate(1C), sma(8C), oimsma(8C)

NOTES

The abstract syntax of a CMISFilter is given in *ISO 9596 : "Information Technology - Open Systems Interconnection - Common Management Information Protocol specification"*.

There is currently no access control implemented in OSIMIS which means that all deletions of managed objects will be attempted. In the future, user authentication information will be needed in addition to the other arguments and the accessDenied error will be returned or objects for which the user has no deletion (write) access rights.

BUGS

The filter expression parser is a bit shaky! You must have the correct number of brackets, as it keeps a count! For instance the filter expression below is missing the outermost set of brackets : "(eventType = linkUpEvent) | (eventType = linkDownEvent)" but instead of reading it as a badly formed OR filter, the parser reads it as the <filteritem> : "(eventType = linkUpEvent)" Additionally, superfluous brackets, e.g : ((eventType = linkUpEvent)) will cause it to fail.

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

evsink – requests and receives event report from an OSI MIB

SYNOPSIS

evsink <agent> <host> [<eventType> ...]

evsink <agent> <host> [<filter>]

DESCRIPTION

evsink is a program that enables to connect to a a remote OSI management agent and request and receive event reports. It establishes first a management association, requests the event reports as specified through the command line arguments by creating a event forwarding discriminator management control object (using a M-CREATE CMIS request) and then listens for event reports which it pretty prints to the standard output.

The program terminates by receiving a SIGQUIT signal when in the forground (Control-\ for most keyboards) or a SIGTERM signal when in the background (produced by the Unix program kill(1)). Upon the reception of the termination signal, it deletes the event discriminator it created and releases the management association before it exits.

OPTIONS

The <agent> argument is mandatory and expresses the remote logical management application "qualifier". In OSIMIS there are two such application qualifiers, SMA for the ISO Transport and the example UNIX MIB agent and OIM-SMA for the OSI Internet MIB agent (OSI view of the SNMP MIB-II for TCP/IP, as described in RFC 1214).

The <host> argument specifies the name of the host where the agent runs - the management application "designator". It should be the name with which the host has been registered in the isoentities database or the global Directory. Note that for Internet hosts, this is usually the last component of its address e.g. athena and NOT athena.cs.ucl.ac.uk .

If no optional argument is given, all event reports are requested. Discrimination on the type of event reports is allowed by specifying either the types of events to be received or a CMIS filtering expression which may contain any other assertions in addition to ones regarding event types.

FILTER EXPRESSIONS

A filter expression is used to construct a CMISFilter value. A CMISFilter contains attribute value assertions (AVAs) that are grouped with the logical operators AND, OR and NOT. The filter expression syntax is as follows : (<cmisfilter>) where <cmisfilter> is one of <notfilter>, <andfilter>, <orfilter> or <filteritem>.

The characters used to represent the logical operators are :

Character	**Operator**
"!"	NOT
"&"	AND
"|"	OR

A <notfilter> has the form : (!(<cmisfilter>)) A <andfilter> has the form : ((<cmisfilter>) & (<cmisfilter>) ...) A <orfilter> has the form : ((<cmisFilter>) | (<cmisfilter>) ...) A <filteritem> has one of the two forms : (<attributename>) for creating a CMISFilter item with the assertion test for "present", or (<attributename> <assertiontype> <attributevalue>) for the other assertion types :

Character	**Assertion type**
"="	equality
":="	substrings
">="	greater or equal
"<="	less or equal
":<"	subset of
":>"	superset of
"><"	non-null intersection

With the substrings operator, the character "*" can be used as a wild card.

Some examples of filter expressions : "((objectClass = eventRecord) & (eventType = linkUpEvent))" "((objectClass = log) & (!(administrativeState = unlocked)))" "((objectClass = log) & ((logId <= 2) | (logId >= 10)))" "((wiseSaying := *hello*) | (!(wiseSaying = bye)))" A "NULL" filter (one that always evaluates to true) can be created using : "(NULL)"

(This is actually an empty AND filter.)

The use of the brackets, "(" and ")", is very important, as they are used to delimit the strings used to represent the components of the filter. Also, please enclose your filter expression in quote marks, as in the examples above, so that the your UNIX shell does not interpret special characters such as "!", "|", "&", ">", "<", "(" and ")".

Please note that the string representation of attribute values is determined by the print and parse methods for the particular ASN.1 syntax. The OSIMIS convention for set valued attributes is that they should be enclosed in angular brackets and items should be separated by a "%". Note that set-valued attributes should also be enclosed in double quotes to avoid the special interpretation of those characters by the UNIX shell. Examples of attribute values are:

foobar - string
"Its always easy if you’re told how" - string
"{ Low: 5 Switch: On High: 7 Switch: On }" - gaugeThreshold
"{ Level:10 Offset:5 Switch:On %
Level:20 Offset:2 Switch:On }" - counterThreshold

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the correct OSIMIS ETC directory before starting this program.

FILES

$(ETC)/oidtable.[gen,at] - object identifier and syntax information.

$(ETC)/isoentities - PSAP address information of management agents.

DIAGNOSTICS

Should be obvious.

EXAMPLES

evsink SMA athena tConnectionCreation tConnectionShutdown

evsink SMA athena "((eventType=tConnectionCreation) | (eventType=tConnectionShutdown))

evsink SMA athena "((objectClass=transportEntity) & (entityId=isode))"

Note that the first two examples are exactly equivalent.

SEE ALSO

mset(1C), evlog(1C), sma(8C), oimsma(8C)

NOTES

The abstract syntax of a CMISFilter is given in *ISO 9596 : "Information Technology - Open Systems Interconnection - Common Management Information Protocol specification"*.

BUGS

The filter expression parser is a bit shaky! You must have the correct number of brackets, as it keeps a count! For instance the filter expression below is missing the outermost set of brackets : "(eventType = linkUpEvent) | (eventType = linkDownEvent)" but instead of reading it as a badly formed OR filter, the parser reads it as the <filteritem> : "(eventType = linkUpEvent)" Additionally, superfluous brackets, e.g : ((eventType = linkUpEvent)) will cause it to fail.

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

evlog - OSI eventLog utility.

SYNOPSIS

evlog <agent> <host> [{<filter> | <event> ... }]

evlog <agent> <host> D <n>

evlog <agent> <host> D <n> <m>

evlog <agent> <host> S <n> <attributeName> <attributeValue>

DESCRIPTION

evlog is a program that allows the use of logging activities on an OSI agent. It does this by allowing the creation and deletion of eventLog objects and eventRecord objects, and by allowing certain attribute values of the eventLog object to be set, controlling its behaviour.

It returns the following values on exit :

Exit	Meaning
0	OK
-1	Bad command line syntax
-2	Could not open a connection to the agent specified
-3	The connection was broken prematurely
-4	Invalid/Bad argument values provided
-5	Other error

The <agent> argument is mandatory and expresses the remote logical management application "qualifier". In OSIMIS there are two such application qualifiers, SMA for the ISO Transport and the example UNIX MIB agent and OIM-SMA for the OSI Internet MIB agent (OSI view of the SNMP MIB-II for TCP/IP, as described in RFC 1214).

The <host> argument specifies the name of the host where the application runs - the application "designator". It should be the name with which the host has been registered in the isoentities database or the global Directory. Note that for Internet hosts, this is usually the last component of its address e.g. lemma and NOT lemma.cs.ucl.ac.uk.

In the first form of usage :

evlog <agent> <host> [{<filter> | <event> ...}]

if no optional argument is given, then an attempt will be made to create an eventLog object at the specified agent. If this is completed successfully, then the eventLog object created will log all notifications emitted at that agent. Created logs are identified by a logId (an integer) which will be displayed on the terminal on successful creation.

If the <filter> expression is given, then a CMISilter will be an be sent as the initial value for the discriminatorConstruct for that eventLog object. (The syntax for the filter expression is described below.) All notifications will have to pass through the filter before being logged. If the <event> names are given, then only those events will be logged by that eventLog. (This is achieved by constructing an OR filter specifying the eventTypes as given by the <event> names provided.)

In the second form of usage : evlog <agent> <host> D <n>

evlog attempts to delete an eventLog object with logId=<n> at the specified agent.

In the third form of usage :

evlog <agent> <host> D <n> <m>

evlog attempts to delete an eventRecord object with logRecordId=<m> at the eventLog with logId=<n>.

In the fourth form of usage :

evlog <agent> <host> S <n> <attributeName> <attributeValue>

evlog attempts to set the value of the attribute given by <attributeName> to the value given by <attributeValue>, for the eventLog object with logId=<n>. The values for <attributeName> and <attributeValue> are as follows :

<attributeName>	**<attributeValue>**
discriminatorConstruct	{<filter> | <event> ... }]
administrativeState	{locked | unlocked}
maxLogSize	<k>
logFullAction	{wrap | halt}

<filter> is a filter expression. <event> is an event name. <k> is an integer that is greater than or equal to zero.

The attribute *discriminatorConstruct* controls which notifications are to be logged. The type of this value is CMISFilter, and all notifications to be logged must pass through this filter.

The attribute *administrativeState* effectively switches logging on or off. In the "locked" state, no logging occurs. In the "unlocked" state, logging can occur.

The attribute *maxLogSize* controls the size of the log. The units of <k> are taken to be octets. <k> should not be less than the present value of maxLogSize. <k> = 0 is taken as meaning infinity.

The attribute *logFullAction* controls the behaviour of the log when it is full. If the value is "wrap" then when the size of the log reaches the value of maxLogSize, the oldest records in the log will be deleted until there is enough room to store the current one. If the value is "halt" then no more logging will occur - to restart logging, either reset the value of maxLogSize or change the value of logFullAction to "wrap".

FILTER EXPRESSIONS

A filter expression is used to construct a CMISFilter value. A CMISFilter contains attribute value assertions (AVAs) that are grouped with the logical operators AND, OR and NOT. The filter expression syntax is as follows :

(<cmisfilter>)

where <cmisfilter> is one of <notfilter>, <andfilter>, <orfilter> or <filteritem>.

The characters used to represent the logical operators are :

Character	**Operator**
"!"	NOT
"&"	AND
"|"	OR

A <notfilter> has the form :

(!(<cmisfilter>))

A <andfilter> has the form :

((<cmisfilter>) & (<cmisfilter>) ...)

A <orfilter> has the form :

((<cmisFilter>) | (<cmisfilter>) ...)

A <filteritem> has one of the two forms :

(<attributename>)

for creating a CMISFilter item with the assertion test for "present", or

(<attributename> <assertiontype> <attributevalue>)

for the other assertion types :

Character	Assertion type
"="	equality
":="	substrings
">="	greater or equal
"<="	less or equal
":<"	subset of
":>"	superset of
"><"	non-null intersection

With the substrings operator, the character "*" can be used as a wild card.

Some examples of filter expressions :

"((objectClass = eventRecord) & (eventType = linkUpEvent))"

"((objectClass = log) & (!(administrativeState = unlocked)))"

"((objectClass = log) & ((logId <= 2) | (logId >= 10)))"

"((wiseSaying := *hello*) | (!(wiseSaying = bye)))"

A "NULL" filter (one that always evaluates to true) can be created using :

"(NULL)"

(This is actually an empty AND filter.)

The use of the brackets, "(" and ")", is very important, as they are used to delimit the strings used to represent the components of the filter. Also, please enclose your filter expression in quote marks, as in the examples above, so that the your UNIX shell does not interpret special characters such as "!", "|", "&", ">", "<", "(" and ")".

PLEASE NOTE that for an attribute type to be used in a filter expression, there should be a "parse" function defined for its syntax and the function should be registered in the syntax tables. This parse function reads the attribute's value from a "pretty-printed" form and converts it to a value, i.e. a C structure. There is no methodology applied in OSIMIS (as yet) to the way in which values are "pretty-printed", however a loose convention is :

"scalar" values are represented as single strings e.g.,

```
logId = 1                    /* INTEGER */
objectClass = eventRecord    /* OID */
wiseSaying = Hello World     /* Strings */
administrativeState = unlocked   /* ENUMERATED */
```

"set" values are enclosed in curly brackets. e.g.,

```
availabilityStatus :< {inTest offLine offDuty}
nUsersThld = {Low:7 Switch:On High:10 Switch:On}
```

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the correct OSIMIS ETC directory before starting this program.

FILES

$(ETC)/oidtable.[at,gen] - syntax and OID information.

$(ETC)/isoentities - PSAP address information of agents.

DIAGNOSTICS

Should be obvious.

SEE ALSO

evsink(1C), readevlog(1)

NOTES

The abstract syntax of a CMISFilter is given in *ISO 9596 : "Information Technology - Open Systems Interconnection - Common Management Information Protocol specification"*.

The Log Control function is described in *ISO 10164-6 : "Information Technology - Open Sytems Interconnection - System Management Functions - Part 6: Log Control Function"*.

BUGS

The filter expression parser is a bit shaky! You must have the correct number of brackets, as it keeps a count! For instance the filter expression below is missing the outermost set of brackets :

"(eventType = linkUpEvent) | (eventType = linkDownEvent)"

but instead of reading it as a badly formed OR filter, the parser reads it as the <filteritem> :

"(eventType = linkUpEvent)"

Additionally, superfluous brackets, e.g :

((eventType = linkUpEvent))

will cause it to fail.

Also, the program does not allow all attribute values to be set when creating the eventLog object, just discriminatorConstruct.

AUTHOR

Saleem N. Bhatti, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS, and the RACE projects NEMESYS and ICM.

NAME

cmisbrowser – a generic OSI MIB browser

SYNOPSIS

cmisbrowser [standard X arguments]

cmisbrowser [standard X arguments]

DESCRIPTION

cmisbrowser is an X Windows HCI tool that allows you to "browse" through the objects in a OSI Management Information Base (MIB). The browser is generic in that it can connect to a CMIS agent without having any prior knowledge of the structure of its MIB.

The browser provides three kinds of window, a *Hosts window* that allows a user to connect to various agents, a *Browser window* that displays the Managed Objects in the MIB and a *Monitor window* that allows you to "monitor" a particular Managed Object. Each of the windows has a number of *button* operations.

Host window operations

There are two button options in this window. The *Connect* button allows you to connect to a remote host-agent. After you have clicked on the button, to select a host agent combination you must double click on an entry in the the list displayed. The list of hosts and agents is extracted from the isoentities file in the OSIMIS ETC directory.

The *Quit* button terminates the browser and automatically closes all windows and connections to remote agents.

Browser window operations

The browser window allows you to move up and down the MIB containment hierachy. Only one Managed Object is displayed at a time. To display the contents of many Managed Objects at one time, you must use the Monitor command (see below).

The browser window is sub-divided into three sections, one that displays that the class and distinguished name of the Current Object
eg transportEntity systemId=athena@subsystemId=transport
, one that displays the names and values of the Attributes of the current object and another that displays the names of the subordinate objects in the containment hierarchy.

The *Up* button allows you to move up the containment hierarchy.

The *Down* button allows you to move down the containment hierarchy. When you have clicked on the down button, you must select a name from the list of Subordinate Objects (again by double clicking). If there is only one subordinate, the browser will automatically move down to the next level.

The *Modify* button allows you to change attributes of the current object. When you have clicked on the Modify button, a menu with two options, to *Change* or *Set Default* is displayed. Once you have selected an option, the menu disappears and you must select an attribute by double clicking on one of the entries in the Attributes list. A dialog box will appear that allows you to change the value of the attribute. Use the mouse to position to the cursor within the edit window and the del key to delete letters.

The *Monitor* button will activate a Monitor window (see below).

The *Refresh* button refreshes the Current Object. This allows you to see if the current object in the remote agent has changed without going up and down the hierarchy. If the object has been deleted, the browser will automatically move one level up the hierarchy.

The *Quit* button terminates the browser window and automatically closes all monitor windows and the connection to the remote agent.

Monitor window operations

The monitor window constantly polls the remote agent for the contents of a managed object. You create monitored objects by selecting the Monitor button in the browser window. There are two button operations in the window. The *Interval* button allows you to change the polling interval. The *Quit* button closes the window.

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the correct OSIMIS ETC directory before starting this program.

FILES

$(ETC)/oidtable.[gen,at] - object identifier and syntax information.

$(ETC)/isoentities - PSAP address information of management agents.

DIAGNOSTICS

Should be obvious.

SEE ALSO

mibdump(1C), set(1C), maction(1C), mcreate(1C), mdelete(1C)

OPTIONS

The cmisbrowser will take any of the standard X arguments: eg -display hostname:0.0 -fg white -bg black. This defaults can also be set in

AUTHOR

James Cowan, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

iproute - routing table manager utility for OIM SMA

SYNOPSIS

iproute <agent> <host> [n]

iproute <agent> <host> C

iproute <agent> <host> D

iproute <agent> <host> C <destination> <gateway>

iproute <agent> <host> D <destination>

DESCRIPTION

iproute allows the management of routing tables when using the OIM SMA. It will display the routing table to be displayed, and allow entries in the table to be created and deleted. Please read the WARNING section before using iproute.

The <agent> argument is mandatory and expresses the remote logical management application "qualifier". In OSIMIS there are two such application qualifiers, SMA for the ISO Transport and the example UNIX MIB agent and OIM-SMA for the OSI Internet MIB agent (OSI view of the SNMP MIB-II for TCP/IP, as described in RFC 1214). Normally for iproute, use OIM-SMA.

The <host> argument specifies the name of the host where the application runs - the application "designator". It should be the name with which the host has been registered in the isoentities database or the global Directory. Note that for Internet hosts, this is usually the last component of its address e.g. lemma and NOT lemma.cs.ucl.ac.uk.

In the first form of usage :

iproute <agent> <host> [n]

iproute will print out the routing table for the <agent> specified. The routing tables is displayed in rows, with each row showing the follwoing information :

destination : This is the IP address of the destination. This could be a host or a network entry.

gateway : This is the IP address of the gateway that is used for routing packets to destination.

interface name : The name of the local network interface, e.g. "le0".

interface type : The type of the sub-network technology for an interface, e.g. "ethernet-csmacd".

interface status : This will be "up" if the interface is in operation, or "down" if it is not.

The addresses *destination* and *gateway* will be resolved to names, of which the "last part" will be printed, e.g. "lemma.cs.ucl.ac.uk" will be printed as "lemma". If the [n] option is specified, the addresses will be printed in the familiar "dot notation", e.g. "128.16.8.60".

In the second form of usage :

iproute <agent> <host> C

iproute creates the ipRoutingTable object at the specified host <agent>. Note that this does not mean that a new routing table is created - if an instance of this object already exists then the request will fail.

In the third form of usage :

iproute <agent> <host> D

iproute deletes the ipRoutingTable object at the specified <agent>. Note that this will cause only the ipRoutingTable object to be deleted, not the actual routing table itself.

In the fourth form of usage :

iproute <agent> <host> C <destination> <gateway>

iproute will create a routing table entry (ipRouteEntry object) for <destination> via <gateway> at the agent specified.

In the fifth form of usage :

iproute <agent> <host> D <destination>

iproute will delete the routing table entry (ipRouteEntry object) for <destination> at the <agent> specified.

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the OSIMIS OIM-ETC directory before starting this program.

WARNING

There is currently NO ACCESS CONTROL implemented in OSIMIS. Note that to demonstrate the use of intrusive management, iproute(1c) creates and deletes routing table entries (ipRouteEntry objects) via CMIS. This facility is enabled by having oimsma(1C) running with root permission. Please be aware of the risks you are taking when running the oimsma with root permission!

FILES

$(ETC)/oidtable.[at,gen] - syntax and OID information.
$(ETC)/isoentities - PSAP address information of agents.

DIAGNOSTICS

Should be obvious.

NOTES

The definition of the OIM MIB can be found in *RFC 1214 : "OSI Internet Management: Management Information Base"*, L. Labarre (Editor).

BUGS

THERE IS NO ACCESS CONTROL when using iproute. Allowing the ipRoutingTable object to be created and deleted is a poor attempt at restricting access to the routing information.

AUTHOR

Saleem N. Bhatti, University College London

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS, and the RACE projects NEMESYS and ICM.

NAME

sma – system management agent

SYNOPSIS

sma
(manually or under /etc/rc.local)

DESCRIPTION

The *sma* server is a system management agent that implements a (non-standard) MIB for the ISO Transport Protocol which is used to manage the ISODE implementation of the latter. It also implements an example non-standard UNIX MIB with a trivial managed objects showing the number of users in the system.

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the OSIMIS ETC directory before starting the agent.

FILES

$(ETC)/oidtable.[gen,at] - object identifier and syntax information.

$(ETC)/isoentities - PSAP address information of management agents.

$(ETC)/mib.init - initial Management Information Base configuration.

SEE ALSO

mget(1C), mset(1C), maction(1C), mcreate(1C), mdelete(1C), evsink(1C), evlog(1C)

DIAGNOSTICS

Various diagnostics are printed when the program is compiled with the -DDEBUG flag.

AUTHOR

George Pavlou, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

NAME

oimsma - system management agent using the OIM MIB

SYNOPSIS

· oimsma (manually or under /etc/rc.local)

DESCRIPTION

The oimsma server is a system management agent that implements the OIM MIB described in RFC 1214. If run with root permission, oimsma can be used in conjunction with iproute(1C), to perform routing table management.

CONFIGURATION

oimsma uses two configuration files that will be found in the OSIMIS ETC directory. The two files are called "if.config" and "sys.config" and are used to hold information used to initialise the ifEntry objects and OIM system object, respectively, on a host. These two files must be modified for your site before starting oimsma.

ENVIRONMENT

The *OSIMISETCPATH* environment variable should point to the OSIMIS OIM-ETC directory before starting the agent.

There is currently NO ACCESS CONTROL implemented in OSIMIS. Note that to demonstrate the use of intrusive management, iproute(1C) creates and deletes routing table entries (ipRouteEntry objects) via CMIS. This facility is enabled by having oimsma(1C) running with root permission. Please be aware of the risks you are taking when running oimsma with root permission!

FILES

$(ETC)/if.config - configuration information for ifEntry objects.
$(ETC)/sys.config - configuration information for the OIM system object.
$(ETC)/mib.init - MIB initialisation file.
$(ETC)/oidtable.[gen,at] - object identifier and syntax information.
$(ETC)/isoentities - PSAP address information of management agents.

DIAGNOSTICS

Various diagnostics are printed when the program is compiled with the -DDEBUG flag.

NOTES

The definition of the OIM MIB can be found in *RFC 1214 : "OSI Internet Management: Management Information Base"*, L. Labarre (Editor).

AUTHOR

Saleem Bhatti, University College London.

This work was supported by the ESPRIT projects INCA, PROOF and MIDAS and the RACE projects NEMESYS and ICM.

CONTENTS

1. Introduction 4
   1.1 Fanatics Need Not Read Further 4
   1.2 A Note on the Implementation 4
   1.3 Changes From Previous Releases 5

2. Overview 7

3. Communication Services 8

4. General Management System Support 9
   4.1 Support for Asynchronous Event-Driven Applications 10
   4.2 Support For Transparent ASN.1 Handling 18

5. The Generic Managed System 28
   5.1 A Tutorial Introduction 29
      5.1.1 Implementing Managed Object Classes 29
         5.1.1.1 Deriving New Managed Object Classes 29
         5.1.1.2 Instantiation 31
         5.1.1.3 Knowledge Sources 32
      5.1.2 Communicating With Real Resources - Examples 33
         5.1.2.1 Polling Example - the nUsers Attribute 33
         5.1.2.2 Upon External Request Example - the sysTime Attribute 33
         5.1.2.3 Event Driven Example - the tpEntity Managed Object 35
      5.1.3 The Complete Example UNIX Managed Object 36
         5.1.3.1 Methods 36
         5.1.3.2 Attributes 37
         5.1.3.3 Notifications 37
         5.1.3.4 Creation 39
   5.2 Managed Object Support 41
   5.3 Attribute and Syntax Support 57
      5.3.1 Attribute types in the GMS 57
         5.3.1.1 C++ implementations of attribute types in the GMS 60
            5.3.1.1.1 Integer 60
            5.3.1.1.2 Real 61
            5.3.1.1.3 OctetString 62
            5.3.1.1.4 String 63
            5.3.1.1.5 DName 64
            5.3.1.1.6 ObjId 64
            5.3.1.1.7 CounterThreshold 65
            5.3.1.1.8 GaugeThresholdInt and GaugeThresholdReal 66

5.3.1.1.9 TideMarkMin and TideMarkMax 68
5.3.1.1.10 Counter 70
5.3.1.1.11 GaugeInt and GaugeReal 71
5.3.1.1.12 Time 74
5.3.1.1.13 Filter 75
5.3.1.1.14 ObjIdList 76
5.3.1.1.15 AdministrativeState 77
5.3.1.1.16 OperationalState 78
5.3.1.1.17 DestinationAddress 79
5.3.1.1.18 LogFullAction 80
5.3.1.1.19 AvailabilityStatus 80
5.3.2 Introducing your own syntaxes into the software 82
5.3.2.1 Implementing the syntax routines 82
5.3.2.2 The C++ class definition 83
5.3.2.2.1 Minimal requirements for the C++ attribute implementation 84
5.3.2.2.2 A slightly more complex implementation 86
5.3.2.2.3 The CMIS Interface to the attribute definition 89
6. Generic Manager Support 90
7. Management Applications 91