GENERAL PURPOSE CLASSES

Class List

Inherits from:	None
Classes used:	Link
Classes related:	ListIterator
Interface file:	GenList.h
Implementation file:	GenList.cc
Containing library:	util

Introduction

The List class provides a generic linked-list facility. It may be used as is, for non-ordered lists containing elements with no special destructing needs. It may be also used, more commonly, as the parent class of specific lists that redefine the *compare* and *deleteElem* methods. The ListIterator class may be used to walk-through the elements of a list.

Methods

```
class List
{
    // ...
protected:
    virtual int   compare (void*, void*);

public:
    virtual void  deleteElem (void*);

    void   append (void*);
    void   prepend (void*);
    void*  get ();
    void*  getLast ();
    void*  first ();
    void*  last ();
    int    insert (void*);
    void*  remove (void*);
    void*  find (void*);
    int    getCount ();
    void   clear ();
    void   setDeleteMode (Bool);

    List (Bool = True);
    virtual ~List ();
};

#define NULLIST ((List*) 0)
```

Polymorphic Methods

```
virtual int compare (void* elem1, void* elem2);
```

Ordered lists need a means to compare stored elements so that order can be maintained. In some other lists there is the need to find if a particular item is in the list. These requirements mean that the *insert*, *remove* and *find* methods need to be used. For those lists this method need to be redefined.

Arguments point to elements of the list and should be casted back to the right type. The return value should be 0 or OK if the elements are equal, a negative value if *elem1 < elem2* and a positive value otherwise. If equality but no ordering applies, NOTOK may be returned if *elem1 != elem2*. In such lists, *find* and *remove* may be used but not *insert.*

```
virtual void deleteElem (void* elem);
```

Lists contain elements with allocated storage space. The elements fall into three categories:

- simple types such as e.g. int, double, char* (for the first two, the contained elements are int*, double*)
- C structures that do not point to other allocated space, except from that of the structure itself, or C++ classes with no destructors
- C structures that point to additional allocated space or C++ classes with destructors

The default behaviour of the List class with respect to destruction is to simply release the pointed space (or not to release any space if this is desirable). This is adequate for the first two cases above which means that this method does not need to be redefined. In the third case, the method needs to be redefined to use the specific de-allocation mechanism for the elements in hand. The argument needs thus to be casted back to the right type.

Note that this method is public which means it may be used to destruct an element after the latter has been possibly removed from the list and “inspected”. In this case, the list object serves as an engine with destruction knowledge.

General Manipulation Methods

```
void append (void* elem);
void prepend (void* elem);
```

Appends an element to the end (tail) and the beginning (head) of the list respectively.

```
void* get ();
void* getLast ();
```

Remove and return the head and tail of the list respectively. The first is not called *getFirst* for historical reasons (backwards compatibility). A synonymous *getFirst* method has been added in versions > 4.0 .

```
void* first ();
void* last ();
```

Return the head and tail of the list respectively without removing it (useful for simple inspection).

```
int insert (void* elem);
```

Inserts an element keeping the list sorted assuming that the compare method has been redefined. If this is not the case, NOTOK is returned. OK is return upon success and the element is inserted in place.

```
void* find (void* elemCopy);
void* remove (void* elemCopy);
```

find finds if the same element as its argument exists in the list while *remove* both finds and removes it. They both assume that the *compare* method has been redefined. If the latter is not the case or if such an element does not exist, NULLVD is returned. A (pointer to an) element is returned upon success. It is noted that in the case of *find* the returned element pointed to is STILL in the list, so it should not be free'd.

```
int getCount ();
```

It returns the number of elements in the list.

```
void clear ();
```

It deletes all the elements. It should always be called before deleting a list or before the list object goes out of scope and the *deleteElem* method has been redefined (see general comment on C++ destructors in ?.?).

```
int setDeleteMode (Bool delMode);
```

The default list destruction behaviour is to simply release the pointed element space (see *deleteElem* above). In special circumstances, the elements of a list may not have their own allocated space but they may be simple pointers to space elsewhere. In that case, the default behaviour should be altered not to release any space at all. This method allows to do that by setting the delete mode to *False*. If the usage of the list later on changes, the delete mode may be set again to the default i.e. *True*. See also the constructor below.

Constructor

```
List (Bool delMode);
```

The only instance variable that may be set through the constructor is the destruction mode. The default is *True* which means that the pointed memory is released - see also *deleteElem* and *setDeleteMode* above.

Class ListIterator

Inherits from:	None
Classes used:	Link, List
Interface file:	ListIterator.h
Implementation file:	GenList.cc (the only non-inline method)
Containing library:	util

Introduction

The ListIterator class may be used to walk-through the elements contained by an instance of the List class.

Methods

```
class ListIterator
{
    // ...

public:
    void   setList (List*);
    void*  getNext ();
    void   reset ();

    ListIterator (List* = NULLIST);
};
```

```
ListIterator (List* list);
void setList (List* list);
```

The list to iterate upon can be set either through the constructor (default case no list) or through the special *setList* method. The argument is the address of the list. The latter may be also used to reset a new list to the iterator.

```
void* getNext ();
```

Returns (a pointer to) the next element of the list. The element remains in the list i.e. it is not removed. NULLVD is returned when the end of the list is reached.

```
void reset ();
```

It resets the iterator to the beginning of the list. It may be used for successive iterations over the same list at different points in time i.e. the same iterator and list objects are used.

Class Array

Inherits from:	None
Classes used:	None
Interface file:	GenArray.h
Implementation file:	GenArray.cc
Containing library:	util

Introduction

The Array class is very similar to the List one but provides a generic array facility. Elements of an array are not ordered but may be compared for equality in order to support search facilities. Arrays have less overhead than lists in terms of memory management as array cells are allocated in chunks and not on a per element basis. The size of an array is automatically incremented whenever instantaneously exhausted. They should be preferred to lists for fairly static non-ordered collections of objects.

Methods

```
class Array
{
    // ...
protected:
    virtual int   compare (void*, void*);

public:
    virtual void  deleteElem (void*);

    void*  get (int);
    int    set (int, void*);
    int    add (void*);
    void*  remove (int);
    int    remove (void*);
    int    find (void*);
    int    getCount ();
    int    getSize ();
    void   compact ();
    void   reset (int);
    void   setIncrement (int);
    void   setDeleteMode (Bool);

    Array (Bool = True);
    Array (int, Bool = True);
    virtual ~Array ();
};

#define NULLARRAY ((Array*) 0)
```

Polymorphic Methods

```
virtual int compare (void* elem1, void* elem2);
virtual void deleteElem (void* elem);
```

These are exactly the same as those of the *List* class. The only minor difference is that the *compare* method should only test for equality and not ordering as the latter does not make sense (this class does not support ordering). 0 or OK should be returned if the two elements are equal and a non-zero value or NOTOK otherwise.

General Manipulation Methods

```
void* get (int cellNo);
```

Returns a (pointer to an) element given a cell number (array index). The returned element remains in the array and should not be free'd. If the cellNo is bigger than the array size (maximum index = sz-1), NULLVD is returned. NULLVD may be also returned if that cell is "empty". In general, in a compact array cellNo should not be bigger than the element count (maximum index = cnt-1).

```
int set (int cellNo, void* elem);
```

Adds an element to the array at cellNo position. If another element was stored in that position it will be lost (free'd). If the cellNo is bigger than the array size (>= sz), NOTOK is returned.

```
void add (void* elem);
```

Adds an element to the array at the first free available position. If the element count has reached the array size, the array is resized according to the available increment (see *setIncrement*).

```
void* remove (int cellNo);
int   remove (void* elemCopy);
```

These support two different ways to remove an element. The first takes as parameter the cell number and restrictions similar to those of *get* above apply. The second assumes that the *compare* method has been redefined and takes as argument a copy of the element to be removed (removal *by value*). It returns -1 (NOTOK) if such an element was not found or the cell number if the element was found and removed (free'd).

```
int find (void* elemCopy);
```

This is similar to the second *remove* method above but it simply checks if such an element exists without removing it.

```
int getCount ();
int getSize ();
```

getCount returns the number of the actual elements in the array while *getSize* returns the array size. It always holds that *cnt* <= *sz* .

```
void compact ();
```

An array may end-up with "holes" of empty cells as elements are removed. *compact* moves active cells "downwards" so that it results in a contiguous space of cells with elements, from 0 until cnt-1 .

```
void reset (int newSize);
```

It resets the array size. If the new size is smaller than the current element count, *newSize-count* elements will be lost. The *getCount* method may be used in advance in order to avoid this behaviour.

```
void setIncrement (int incr);
```

Every time the element count exceeds the array size, the latter is immediately incremented. The default increment is **20** but it can be changed through *setIncrement* to reflect the nature of the specific array.

```
void setDeleteMode (Bool delMode);
```

The default array destruction behaviour is to simply release the pointed element space (see *deleteElem* above). In special circumstances, the elements of an array may not have their own allocated space but they may be simple pointers to space elsewhere. In that case, the default behaviour should be altered not to release any space at all. This method allows to do that by setting the delete mode to *False*. If the usage of the array later on changes, the delete mode may be set again to the default i.e. *True*. See also the constructors below.

Constructors

```
Array (Bool delMode);
Array (int size, Bool delMode);
```

The only instance variables that may be set through the constructors are the destruction mode and the size. The default destruction mode is *True* which means that the pointed memory is released - see also *deleteElem* and *setDeleteMode* above. The size may be an initial guess of the array size to avoid re-allocating cells many times. If not supplied, cells are allocated in chunks of *incr* each time - see also *setIncrement* above.

PROCESS COORDINATION SUPPORT CLASSES

Class KS

Inherits from:	None
Classes used:	Coordinator and derived classes
Interface file:	GenericKS.h
Implementation file:	GenericKS.cc
Containing library:	kernel

Introduction

The knowledge source (KS) is an abstraction of a general application object being able to receive data asynchronously in external communication endpoints and to be awaken-up periodically in real-time to perform various tasks. The term has its roots to AI blackboard systems and generally KSs implement an application's intelligence with respect to external communications and periodic real-time activities. The KS is an abstract class.

Methods

```
class KS
{
protected:
    // real time wake-up capabilities
    int         scheduleWakeUp (long, char*, Bool = False);
    int         scheduleWakeUps (long, char*, Bool = False);
    int         cancelWakeUps (char*);

    // listening on external communication endpoints
    int         startListen (int);
    int         stopListen (int);

    // notify process shutdown
    int         notifyShutdown ();

    // protected constructor (abstract class)
    KS ();

public:
    // call-backs for wake-ups, external events, process shutdown
    virtual int wakeUp (char*);
    virtual int readCommEndpoint (int);
    virtual int shutdown (int);

    // destructor
    virtual ~KS ();
};
```

Real-time wake-up requests

```
int scheduleWakeUp  (long period, char* token, Bool onlyForNotification);
int scheduleWakeUps (long period, char* token, Bool onlyForNotification);
int cancelWakeUps   (char* token);
```

The first two methods request the initiation of real-time wake-up(s) while the third requests their cancellation.

The *scheduleWakeUps* method schedules wake-ups to take place every *period* seconds. The *token* argument should be a unique character string that distinguishes this series of wake-ups from any others possibly requested by the same knowledge source. The *token* does not need to have allocated memory as the kernel infrastructure makes a copy e.g. a constant string such as "<token>" will be adequate. NULLCP may be supplied if this distinction is not needed or if only one series of wake-ups is scheduled by that knowledge source.

The *onlyForNotification* parameter supports an optimisation relevant only to applications in OSI agent roles, and as such it is optional with default value *False*. If it is *True*, the knowledge source will be awaken only if event forwarding discriminators or log support managed objects are present in the system. This is useful when the wake-ups are only used to support notifications via real resource polling. The default case (parameter not supplied) is to request wake-ups independently of the notification function. OK is returned upon success, NOTOK if the scheduling period is invalid i.e. less than or equal to zero.

The *scheduleWakeUp* method is exactly the same as above with the difference that only **one** wake-up is scheduled.

The *cancelWakeUps* method will cancel either multiple or single wake-ups identified by *token*. Note that if token is NULLCP only wake-ups with exactly that token will be cancelled i.e. NULLCP does NOT mean "cancel all the pending wake-ups for that knowledge source". OK is returned upon success, NOTOK if the cancel operation failed i.e. there was no scheduled wake-up with this token.

External listening and termination notification requests

```
int startListen (int fd);
int stopListen (int fd);
```

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

stopListen enables to de-register an external point of communication so that no more listening on that endpoint takes place. OK is returned upon success, NOTOK if the operation failed i.e. there was no such endpoint registered.

```
int notifyShutdown ();
```

This method provides the capability to request to be notified at process shutdown time.

Polymorphic callback events

The following methods are callbacks for wake-ups, external communication events and process shutdown as results of the above requests. They should be supplied in derived classes according to the use of the above request methods.

```
virtual int wakeUp  (char* token);
```

This is called as a result of the *scheduleWakeUp(s)* methods. The *token* is the one supplied in the scheduling call. The method should return OK when redefined by a user in a derived class. NOTOK is returned at the KS level i.e. when a wake-up is requested and the method has not been redefined (development time bug).

```
virtual int readCommEndpoint (int fd);
```

This is called every time there is data at the external communication endpoint identified by *fd*. The latter serves to distinguish the endpoint in hand from other endpoints used by the same knowledge source. The method returns OK or NOTOK in the same fashion as *wakeUp* above.

```
virtual int shutdown (int fd);
```

This method is called at process shutdown time as a result of both the *startListen* and *notifyShutdown* methods. In the former case, it is called once for every external communication endpoint identified by *fd* this application is still listening to so that the latter is closed-down gracefully - note that there is no need for a *stopListen* call in this case.

In the latter case the method is called with *fd = -1* as an indication of process termination so that the knowledge source can do any other necessary cleaning up. If both cases hold i.e. both endpoints open and a *notifyShutdown* request, the call with *fd = -1* is the last one after the calls for all the endpoints.

The method returns OK or NOTOK in the same fashion as *wakeUp* above.

Class Coordinator

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

Introduction

An instance of this class coordinates activity in distributed applications, acting as a central point for communication through all the external endpoints and for all the scheduled alarms in real-time. Only one instance of this or any derived classes should be present in an application that is realised as a single operating system process.

In UNIX systems it uses the *select(2)* facility, enhanced in ISODE as *xselect* to be portable across UNIX platforms. A fully event-driven scheme is exercised with respect to all external communications and scheduled real-time alarms v through a *first-come-first-served* policy.

This class is written in such a way to allow integration through inheritance with other packages that have their own coordinating mechanisms. This is necessary for distributed applications with graphical user interfaces that receive asynchronous events from both the keyboard and the network. It may be also necessary in order to integrate OSIMIS with other distributed systems platforms. There exist already extensions of this class to work with X-Windows and the ISODE platform (XCoordinator and ISODECoordinator respectively).

It should be noted that the existence of a coordinator instance is totally transparent to application implementors who can access its services through KS-derived classes. It is only visible at initialisation time as explained below.

Methods

```
class Coordinator
{
    // ...

public:
    // central listening loop

    void        listen ();

    // interface to other coordinating mechanisms

    static void setTerminateSignalHandler ();
    static void setAlarmSignalHandler ();

    // ...
};
```

Central listening

```
void listen ();
```

This method realises the central listening loop of an application implemented as one operating system process. It is the last method call that is part of the *main()* program and it should be called after all the necessary application initialisation has finished. It never returns but the process will terminate upon the receipt of a termination signal. The coordinator will then call the *shutdown* method of knowledge sources as explained in the KS class specification.

It is noted that this method should only be called if the central listening loop is under the full application's control. This is the case for applications that are not graphical user interfaces. The latter use the XCoordinator class, see the relevant specification.

Interface to other coordinating mechanisms

```
static void setTerminateSignalHandler ();
static void setAlarmSignalHandler ();
```

These can be used to define who handles the termination and alarm signals in the case of integration with another coordination mechanism e.g. that of X-Windows. They are only meaningful if the *listen* method above is not called but listening is under the full control of the other mechanism. The former method should be always called in that case while the latter may or may not be called.

If *setAlarmSignalHandler* is not called, alarms will be handled by the other mechanism. This provides the capability to use both the KS API and that of the other mechanism to deal with real-time alarms. If it is called, the latter is not possible as alarms will be handled by OSIMIS but a marginally better performance should be expected.

In general, *setAlarmSignalHandler* should not be called only if OSIMIS code is integrated with that of a user interface which has already been using the GUI's alarm API. In all other cases, it should be better called so that alarm signals are handled by OSIMIS.

Class ISODECoordinator

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

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 PSAP descriptor so that explicit use of the UNIX *select(2)* or any other similar facility is not possible. All this class does is to simply redefine the polymorphic *readCommEndpoints* method to use *iserver_wait* instead of *(x)select*.

It should be noted that in this case OSIMIS has still control of the coordination mechanism and the listening procedure should be initialised through the Coordinator::*listen* method. Since X-Windows based GUIs need their own XCoordinator class, it is not possible to have GUI applications that wish to receive association requests realised as one operating system process (e.g. TMN OSs). This is not a problem if TCL/TK is used as the GUI building mechanism as it encourages a two-process model, with the GUI handler in TCL/TK and the application engine in C/C++ (???).

Class XCoordinator

Inherits from:	Coordinator
Classes used:	None
Interface file:	XCoord.h
Implementation file:	XCoord.cc
Containing library:	kernel

Introduction

The XCoordinator class is a special coordinator for OSIMIS applications that wish to use X-Windows based graphical user interfaces, usually applications in manager roles only. It essentially integrates the OSIMIS coordinating mechanism with that of X-Windows.

Note that it is not possible for such applications to receive association requests e.g. in cases where an agent tries to establish one in order to send an event report (see also the IsodeCoordinator class). They may request and receive event reports though through an existing association they have established to an agent.

An application using the XCoordinator class should not call the Coordinator::*listen* method. It should call the *setTerminateSignalHandler* while it may also call the *setAlarmSignalHandler* (see the Coordinator class specification). You may inspect a trivial example of using the OSIMIS coordination mechanism together with X-Windows in $(TOP)/kernel/example/XExample.cc

HIGH-LEVEL ABSTRACT SYNTAX SUPPORT CLASSES

Class Attr

Inherits from:	None
Classes used:	None
Interface file:	GenericAttr.h
Implementation file:	GenericAttr.cc
Containing library:	kernel

Introduction

The Attr class represents an abstract syntax type, encapsulating both data and manipulation behaviour (encoding, decoding etc.) The ASN.1 abstract syntax language that is used by all OSI applications has been the basis for this abstraction. The latter is nevertheless general enough to cope with other abstract syntax frameworks. In OSIMIS, the Attr class is used to model management attributes, actions and notifications. It should be noted that the name Attr denotes any instance of an abstract syntax, as e.g. in attribute value assertion, and not a management attribute.

Attr contains the actual attribute value as a C data structure corresponding to the ASN.1 type. The value is held as a C structure because the *pepsy* ASN.1 compiler does not support C++: the Attr class provides essentially the C++ wrapper. 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 interface.

This class defines a set of virtual methods which may be redefined in derived classes. Such classes for the generic management 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 DMI types e.g. administrative and operational state, attribute value change etc. are provided by OSIMIS. It is almost certain though that applications introducing new managed objects will need additional types. The documentation of this class together with the general guidelines in the section describing the specific OSIMIS realised provide the framework for introducing these.

Methods

```
class Attr
{
    // ...

public:

    // general syntax manipulation methods

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

    PE                  encode ();
    char*               print ();
    void                ffree ();
    void*               copy ();
    int                 compare (void*);
    void*               find (void*);       // only for multi-valued syntaxes
```

```
    PE                  encode (void*);
    void*               decode (PE);         // no corresponding method above
    char*               print (void*);
    void                ffree (void*);
    void*               copy (void*);
    int                 compare (void*, void*);
    void*               find (void*, void*); // only for multi-valued syntaxes

    void*               getElem (void*);     //  ..
    void*               getNext (void*);     //  ..

    // methods to access and modify the contained attribute value

    void*               getval ();
    void                setval (void*);
    void                replval (void*);
    int                 setstr (char*);

    // methods that may be redefined in derived classes to associate behaviour

    virtual void*       get ();
    virtual int         set (void*);
    virtual int         setDefault (void*);
    virtual int         add (void*);         // only for multi-valued syntaxes
    virtual int         remove (void*);      //  ..

    // destruction

    void                clear ();
    virtual             ~Attr ();

    // ...
};

#define NULLATTR ((Attr*) 0)
```

Syntax manipulation methods

The following set of methods fall into four categories:

- methods that check the syntax type
- methods that manipulate the encapsulated value
- methods that manipulate an externally supplied value or values (similar to the ones of the first category)
- methods that allow one to walk through the values of a multi-valued syntax

All these methods, apart from the first one below (*getSyntax*), use protected polymorphic syntax manipulation methods (*_encode*, *_decode*, etc.) for the syntax in hand. The latter are produced automatically by the object-oriented ASN.1 compiler - see section ?.

```
virtual char* getSyntax ();
Bool  isMultiValued ();
```

getSyntax is automatically produced in derived classe by the object-oriented ASN.1 compiler and returns the syntax type exactly as registered in oidtable.at .

isMultiValued returns *True* if the syntax is multi-valued (ASN.1 SET OF or SEQUENCE OF), *False* otherwise.

```
PE    encode ();
char* print ();
void  ffree ();
void* copy ();
int   compare (void* val);
void* find (void* val);
```

encode encodes the contained value. The presentation element returned is not a copy and should not be free'd.

print pretty-prints the contained value. The returned string has allocated memory and should eventually be free'd.

ffree frees the contained value. It is called ffree as free is a C/C++ reserved word.

copy returns a copy of the contained value. The returned value should eventually be free'd, using possibly the *ffree(void*)* method.

compare compares its argument to the contained value (see *_compare* for the returned values).

find finds if its argument is an element of the encapsulated multi-valued syntax. Only the first element of the argument is examined. Note that the argument should be a pointer to the outermost structure e.g. I*ntegerListVal** instead of the contained *int**. NULLVD is returned is such an element is not found.

```
void* decode (PE pe);
PE    encode (void* val)
char* print (void* val);
void  ffree (void* val);
void* copy (void* val);
int   compare (void* val1, void* val2);
void* find (void* val1, void* val);
```

These are exactly the same as the ones in the group above but they operate on the argument(s) instead of the contained value. They are very useful in treating an instance of this class as a syntax engine. Note that *decode* has not counterpart in the group above while find checks if *val1* is a part of *val*.

```
void* getElem (void* val);
void* getNext (void* val);
```

These two together may be used to walk through the elements of a multi-valued syntax. If the value of getNext is NULLVD, the first element i.e. the encapsulated value is returned. An example of their use:

```
IntegerList intList;
// ...
void* cur = NULLVD;  // signifies the beginning

while (cur = intList.getNext(cur)) {
    int* elem = (int*) intList.getElem(cur);
    printf("int elem is %d\n", *elem);
}
```

Attribute value access and manipulation

`void* getval ();`

It returns the contained (pointer to the) C language datastructure corresponding to the ASN.1 type. The returned value is not a copy and should NOT be free'd.

`void setval (void* newValue);`

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

`void replval (void* newValue);`

It sets replaces the contained value with the supplied one via the argument. 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 acquired via *getval*, altered and replaced. This is particularly useful for complex or multi-valued types where only a particular element needs to be modified.

`int setstr (char* strValue);`

It sets the contained value to the equivalent structure to strValue which is its pretty-printed representation e.g. *setstr("5")* for an Integer type. The *_parse(char*)* method is used for the parsing. The contained value is always free'd. NOTOK is returned upon failure to parse the supplied argument.

Polymorphic attribute value access and manipulation

The following set of methods allow to access and manipulate the contained attribute value and may be redefined to perform additional checks, associate behaviour etc. If the method is redefined to perform additional checks e.g. for the range of values, *add* and *remove* do not need to be redefined for the same purpose as they use *set* at the Attr level.

The *setDefault* method simply uses *set* at the Attr level and needs to be redefined only if special 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 is respectively needed.

Another reason for redefining the *get*, *set*, *add* and *remove* methods is in order to possibly associate real resource behaviour for applications in agent roles. Associating an attribute to a real resource should only be done 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 (see section ?.?).

```
virtual void* get ();
```

It 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);
```

It 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. When not redefined, it is equivalent to *setval*. It returns OK upon success and NOTOK upon failure (invalid value), the latter only if it has been redefined.

```
virtual int add (void* addValue);
```

It may be redefined to associate the attribute to a real resource. It may return NOTOK if the supplied value is invalid.

```
virtual int remove (void* addValue);
```

It may be redefined to associate the attribute to a real resource. It may return NOTOK if the supplied value is invalid.

```
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.

Class AnyType

Inherits from:	Attr
Classes used:	None
Classes related:	other Attr-derived types, AVA
Interface file:	AnyType.h
Implementation file:	AnyType.cc
Containing library:	kernel

Introduction

OSIMIS uses syntax tables to provide high-level abstract syntax support facilities. These are the *oidtable.gen* for general object identifiers and the *oidtable.at* for identifiers with associated syntax e.g. management attributes, actions and notifications. Specific abstract syntaxes can be implemented by the AnyType class through a table look-up every time a new type is instantiated. This should be avoided by agents but could be used by managers or hybrid units 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 Attr which provides those methods through the tables. Though this class can be used as a base class for specific attribute types with the gain of avoiding to redefine those methods, this is discouraged as it will result in making agents bound to the use of tables. Nevertheless, this class is very useful for high-level manager access APIs such as the RMIB.

Methods

```
class AnyType
{
    // ...

public:
    // constructors

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

    static Bool createError ();

    // ...
};
```

Constructors

```
AnyType (char* sntx, void* val);
```

The *sntx* argument can be either the name of the associated identifier as in the first column of *oidtable.at* e.g. logId, pdusSentThld, or the actual syntax name as in the third column of that table e.g. Integer, CounterThreshold. The former is checked first and as such it should be preferred in terms of performance. Note that the resulting object does not keep the associated object identifier or name but it does point to the syntax name in *oidtable.at*.

The *val* argument should be the value as a (pointer to a) C structure corresponding to the ASN.1 syntax as produced by the pepsy ASN.1 compiler or as hand-crafted. Space for that structure should have been allocated before hand. A simple example:

```
int* i = new int; *i = 10;
AnyType intType(“Integer”, i);
char* cp; printf(“%s\n”, cp = intType.print()); free(cp);
```

```
AnyType (char* sntx, char* val);
```

This is the same as above but the value is passed as a string argument. In this case memory does not need to be allocated as an internal representation (C structure) is built. The string value should be exactly according to the printing convention for the type, else construction will fail. The same example as above would be:

```
AnyType intType(“Integer”, “10”);
```

```
AnyType (OID sntx, PE val);
```

This is a special constructor for values coming encoded from the network. *sntx* is the object identifier of an entity with that syntax and *val* is the associated value encoded as a presentation element. Note that the value is decoded at construction and, as such, the *val* memory is not utilised i.e. can be free’d after the construction - the same applies to *sntx* (the pe_free and oid_free ISODE routines may be used).

```
static Bool createError ();
```

When constructing an instance, an error may occur if the syntax and value arguments are not in accordance. This static method provides a means to check for such an error by returning a boolean value denoting if an error took place or not during the last construction. If it returns *True*, the resulting object should be deleted. Note that this should only be a development or testing time error.

Class AVA

Inherits from:	None
Classes used:	Attr and derived classes
Interface file:	AVA.h
Implementation file:	AVA.cc
Containing library:	kernel

Introduction

The AVA (Attribute Value Assertion) class provides an attribute type and value pair, together possibly with a CMIS modify operator and an error. A AVA instance with the first three may be used as argument to CMIS operations in high-level manager access APIs while the fourth may be used in get and set results to denote partial errors. A AVA instance with a CMIS error, a type and possibly a value may be used as the parameter carried in CMIS processing failure errors. The AVA class uses oidtables to map types to values.

In short, all possible uses of the AVA information in high level APIs are:

```
type                              - get attribute
type, value                       - action argument/result, event information,
                                    attr get/set/create result,
                                    initial create attribute value
type, value, modify               - set attribute value

error, type                       - partial get error
error, type, value, modify        - partial set error

error                             - error code only
m_processingFailure, type, value - error parameter with specific error/info
```

Methods

```
class AVA
{
    // ...

public:
    // general access methods

    char*        getName ();
    OID          getOid ();
    Attr*        getValue ();
    CMISErrors   getError ();
    char*        getErrorStr ();
    CMISModifyOp getModifyOp ();
    char*        print ();
    void         clear ();
    int          adjustActionOid ();
```

```
    // constructors

    AVA (char*, char*, CMISModifyOp = m_noModifyOp);
    AVA (char*, void*, CMISModifyOp = m_noModifyOp);
    AVA (char*, Attr*, CMISModifyOp = m_noModifyOp);

    AVA (CMISErrors, char*, Attr* = NULLATTR);
    AVA (CMISErrors, OID,   Attr* = NULLATTR);
    AVA (CMISErrors);

    static Bool createError ();

    // ...
};

#define NULLAVA    ((AVA*)  0)
#define DISCARDAVA ((AVA*) -1)
```

General access methods

`char* getName ();`

Returns the name of the associated type as registered in the first column of oidtable.at. The returned value points to storage used for that table and, as such, it should NOT be free'd.

`OID getOid ();`

Returns the object identifier of the associated type. The returned value is not a copy and should NOT be free'd.

`Attr* getValue ();`

Returns the value of the associated type (if any). The returned value is not a copy and should NOT be free'd.

`CMISErrors getError ();`

Returns the CMIS error associated with that AVA. If there is no error, *m_noError* is returned. CMISErrors is defined in mparm.h (MSAP API).

`char* getErrorStr ();`

Returns the above CMIS error in string form. If there is no error, "noError" is returned. The returned value points to static storage and should NOT be free'd.

`CMISModifyOp getModifyOp ();`

Returns the CMIS modify operator associated with that AVA. If there is no associated modify operator, *m_noModifyOp* is returned. CMISModifyOp is defined in mparm.h (MSAP API).

```
char* print ();
```

It returns a string value with structure:

```
“[error: <error>] [<type>: [<value>]] [<modify>]”
```

The square brackets denote optionality. The returned string has allocated memory and should be free’d.

```
void clear ();
```

It deletes all the contained elements. It may be called explicitly but is also called from the destructor.

```
int adjustActionOid ();
```

This should be used only after creating an action argument through the “Info” type extension and a string form for the value [AVA(char*, char*) constructor]. Its effect is to convert back the action type to the original i.e. without the “Info” extension. The reason for that extension is that a type can have only one associated syntax in oidtable.at while actions may have both an information and a reply syntax (see section ?.? for more information).

The use of that constructor for action arguments subsequently necessitates the use of this method and as such is discouraged. Examples of alternative solutions (the above solution shown last) are:

```
actionArg = new AVA(“calcSqrt”, new Real(4));
actionArg = new AVA(“calcSqrt”, new AnyType(“calcSqrtInfo”, “4”);

actionArg = new AVA(“calcSqrtInfo”, “4”); // should be better avoided
actionArg -> adjustActionOid();
```

Constructors

```
AVA (char* sntx, char* val, CMISModifyOp modify);
AVA (char* sntx, void* val, CMISModifyOp modify);
AVA (char* sntx, Attr* val, CMISModifyOp modify);
AVA (OID   sntx, Attr* val, CMISModifyOp modify);
```

The first three provide different ways to initialise the value while the fourth provides a different way to initialise the type. The fourth argument (*modify*) is optional in all of them and should be used when setting attributes.

The three different ways of setting the value are char*, void* and Attr* with allocated space expected for the last two. In the first two the validity of the type/value matching is checked while in the third (Attr*) it is NOT, so be careful.

In the last one the type is set through an object identifier which must have allocated space. You may use the *oid_cpy* ISODE routine to copy an existing one. The advantage of the last constructor is that it does not incur a table search.

```
AVA (CMISErrors err, char* sntx, Attr* val);
AVA (CMISErrors err, OID   sntx, Attr* val);
AVA (CMISErrors err);
```

Similar to the above but with emphasis on the error. They may be used in high-level agent APIs. No checking is done for the validity of the type/value matching, so be careful. Allocated memory is expected for the OID and Attr* arguments, see above.

```
static Bool createError ();
```

Exactly as the same method of the AnyType class, it provides a means for checking construction time errors. It is meaningful only after the (char*, char*) and (char*, void*) constructors - see above.

Class AVAArray

Inherits from:	Array
Classes used:	AVA
Interface file:	AVA.h
Implementation file:	inline

Introduction

The AVAArray class implements a specialised array of AVA instances. There is nothing special about it but it is mentioned here because it is used in high-level manager access APIs. Its inline implementation shown below may serve as a simple example of a specialised array.

Methods

```
class AVAArray
{
protected:
    void         deleteElem (void*);

public:
    AVAArray (int);
};

#define NULLAVAARRAY ((AVAArray*) 0)


inline void AVAArray::deleteElem (void* e)
{ delete (AVA*) v; }

inline AVAArray::AVAArray (int n) : Array (n, True)
{ }
```

GENERIC MANAGED SYSTEM CLASSES

Class MOClassInfo

Inherits from:	None
Classes used:	NameBinding, Attr and derived classes
Interface file:	MOClassInfo.h
Implementation file:	MOClassInfo.cc
Containing library:	gms

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 instances of the class, such as attribute, group, event and action identifiers, name bindings etc. Instances of this class are linked hierarchically in a tree mirroring the GDMO inheritance hierarchy.

Such meta-class information is vital for applications in agent roles and may also be used by applications in manager roles. In OSIMIS, it is only used by applications in agent roles. The code that initialises the meta-class objects is produced by the GDMO compiler as a static method of every managed object class (*initialiseClass*). Ideally, a database representation should be produced (e.g. a flat-file) so that the same information could be parsed and stored internally by applications in manager roles.

Most of the methods of this class are only used by the GMS to access information that is needed to verify requested name bindings at creation and related behaviour at deletion, the existence and validity of attributes and values accessed in get, set and create operations and the existence and validity of actions and their arguments. The only methods that may be of interest to agent application implementors are those that provide access to the template objects for the action information and reply (see also the MO::action method). The *getParent* method allows one to ascend an inheritance branch while the *getFirst* method provides access to the first instance of that class, from which subsequent instances may be accessed. A number of other methods provide access to class information but are not described here - the declaration header file may be consulted if needed.

Methods

```
class MOClassInfo
{
    // ...

public:
    Attr*        getActionInfoTemplate (int);
    Attr*        getActionReplyTemplate (int);

    MOClassInfo* getParent ();
    MO*          getFirst ();    // deprecated

    // ...
};
```

Action template access methods

```
Attr* getActionInfoTemplate (actionId);
Attr* getActionReplyTemplate (actionId);
```

Action argument and reply information is passed to the polymorphic *MO::action* method interface as C data structures produced by the pepsy ASN.1 compiler (void*). The specific Attr-derived classes produced by the O-O ASN.1 compiler that encapsulates pepsy that correspond to those types are stored in the class object. The user code of a specific action method may resort to these in order to manipulate the action parameter and reply information e.g. to print, parse, free those values - see the Attr class specification.

The parameter to both methods is the actionId (an integer tag), as produced by the GDMO compiler for a specific managed object class e.g. I_calcSqrt for the calcSqrt action of the simpleStats class. The Attr value returned may be used as a syntax engine and should NOT be free'd.

Accessing the class hierarchy

```
MOClassInfo* getParent ();
```

Every specific MO class produced by the GDMO compiler contains a static instance of MOClassInfo which is shared by all the instances of that class. The convention is that this meta-class instance can be accessed by a private instance variable named _classInfo or by a public method getClassInfo(). All the meta-class instances are linked in a tree mirroring the inheritance hierarchy and the getParent method allows one to ascend this hierarchy. For example, eventForwardingDiscriminator::getClassInfo->getParent() will yield the meta-class object for discriminator.

Walking through all the instances of a class

```
MO* getFirst ();
```

This method was initially provided as a convenience to be able to access all the instances of a particular class by having access to the meta-class object. It is obviously meaningful only in agent applications. Though it is still available, it is *DEPRECATED* which means that it may NOT be provided in future major versions. The reason for that is that it provides access to all instances of a particular actual class, independently of their position in the Management Information Tree (MIT). It even provides access to instances in different MITs if there are many logical agent applications present in the same physical block (e.g. TMN OSFs in an OS) and this is illegal and dangerous. The MO class provides a comprehensive set of other methods that allow one to search the MIT and can serve the same purpose. Please change your software and avoid using this method for future compatibility.

It is worth mentioning its use: it provides access to the first instance of a particular actual class where first refers to creation in time. The MO::getClassNext() method may then be used to walk through all the instances of that class - this will be also eventually eclipsed. Note that e.g. eventForwardingDiscriminator::getClassInfo()->getFirst() will give a handle to all eventForwardingDiscriminators but NOT to any instances of derived classes. It is only the ACTUAL class instances that are linked together.

Class MOClassArray

Inherits from:	Array
Classes used:	MOClassInfo
Interface file:	Create.h (will be moved eventually to MOClassInfo.h)
Implementation file:	inline implementation
Containing library:	inline implementation

Introduction

The MOClassArray class is a container class containing all the meta-class objects in a particular application. There is always exactly one instance of this class in every (agent) application. It provides search facilities to enable access to a particular meta-class object. The MOClassInfo::getParent method may be subsequently used to traverse (ascend) an inheritance branch.

Methods

```
class MOClassArray : public Array
{
    // ...

public:
    static       MOClassArray* getInstance();

    MOClassInfo* findClass (char*);
    MOClassInfo* findClass (OID);

    // ...
};
```

Instance access

```
MOClassArray* getInstance ();
```

As there is always one instance of this class in a particular application, this static method provides access to its single instance e.g. MOClassArray::getInstance() .

Searching for a particular meta-class

```
MOClassInfo* findClass (char* className);
MOClassInfo* findClass (OID   classOid);
```

The above methods return the desired meta-class object. The first takes as parameter the class name while the second uses the Object Identifier as it registers this class in the GDMO module. User code should need only the former.

Class MO

Inherits from:	None
Classes used:	MOClassInfo, Attr
Interface file:	GenericMO.h
Implementation file:	GenericMO.cc
Containing library:	gms

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 tree and handles to all the meta-class objects for that instance. It provides a set of polymorphic methods that may be re-defined in derived classes to achieve the desired functionality. MO is the parent of Top which is subsequently the parent of specific derived classes produced by the GDMO compiler. Those contain standard template generic code and may be augmented with code implementing the polymorphic methods mentioned above. Special methods produced by the GDMO compiler for derived classes are mentioned last.

The documentation of this class is not yet complete in terms of the details associated with every method call. It will be completed in the next version of this manual.

Methods

```
class MO
{
    // ...

    // polymorphic methods (to be possibly provided in derived classes)
public:
    /* static RDN makeRdn (MO*);  // produced for derived classes */
    virtual int  createRR (AVA*&, void* = NULLVD, int = -1);
    virtual int  deleteRR (AVA*&, void* = NULLVD, Bool = False, int = -1);
protected:
    virtual int  get (int, int, AVA*&, Bool = False, int = -1);
    virtual int  set (CMISModifyOp, int, int, void*, AVA*&,
                      Bool = False, int = -1);
    virtual int  action (int, int, void*, void*&, Bool&, AVA*&,
                      Bool = False, int = -1);
    virtual int  buildReport (int, int, void*&, Bool&);
    virtual int  refreshSubordinates (AVA*&, int = -1);
    virtual int  refreshSubordinate (RDN, AVA*&, int = -1);

    // asynchronous callbacks (resulting from the above calls)
public:
    int          createRRes (int, AVA*);
    int          deleteRRes (int, AVA*);
    int          getRes (int, AVA*);
    int          setRes (int, AVA*);
    int          actionRes (int, void*, Bool, Bool, AVA*);
    int          refreshSubordinatesRes (int, AVA*);
```

```
    // General MIT access methods
public:
    static int  initialiseMIB (char*);
    static MO*  getRoot ();
    static DN   mn2localdn (MN);
    static DN   getAgentNameDomainDN ();

    void       deleteWholeSubtree (DeletionType = dt_undefined, Bool = False);

    int         update (AVA*&);
    Attr*       getUpdatedAttr (OID, AVA*&);    // forces a CMIS GET
    MO*         getMO (DN, Bool, AVA*&);
    MO*         getMO (char*, Bool, AVA*&);
    MO*         getSubordinate (RDN, Bool, AVA*&);
    MO*         getSubordinate (char*, Bool, AVA*&);
    MO**        getSubordinates (Bool, AVA*&);
    MO**        getWholeSubtree (Bool, AVA*&, Bool = False);

    MO*         getSubordinate ();      // first subordinate in binary MIT
    MO*         getSuperior ();
    MO*         getPeer ();

    MOClassInfo* getClassInfo ();
    MO*          getClassNext ();       // deprecated

    OID         getClass ();
    char*       getClassName ();
    int         checkClass (char*, Bool* = NULL);
    int         checkClass (OID,   Bool* = NULL);
    RDN         getRDN ();
    char*       getRDNString ();
    DN          getDN (Bool = False);
    char*       getDNString (Bool = False);

    // ...
};
```

Polymorphic methods and related asynchronous callbacks

These methods may be all redefined in specific derived class to implement the associated behaviour. The makeRdn method is actually produced by the GDMO compiler for classes with the “automatic-instance-naming” property, so it is not as such a polymorphic method and is presented in a commented-out fashion in this class.

The polymorphic methods and properties are related to CMIS operations, namely Get, Set, Action, Create and Delete. The buildReport method is only a remnant of the past, the triggerEvent method of the Top class could be enhanced to support the information passing needed (see Top class). The refreshSubordinate(s) methods relate to object addressing and scoping when a “fetch-upon-request” policy is exercised between managed objects and associated resources.

The philosophy behind the API of the createRR, deleteRR, get, set and action methods is that a class can be augmented with behaviour in such a way to make possible the total re-usability of that class by future derived classes. As such, each of those calls should be passed from the leaf to the parent of the inheritance branch in any implementation. In the case of createRR, though the calls should have the same “upwards” direction, the actual behavioural code in those calls should operate in a “downwards” fashion, as during the construction of objects in typical object-oriented languages. This is possible by organising the contents of createRR suitably.

In the case of get and set, each attribute is requested / set separately and a “no-more-attributes” indication is passed at the end. If all the attributes are requested , only one “get-all” call is passed to the object. In the case of the get method, the behavioural code has to update the requested attributes (or all of them according to the update policy). If a periodic “cache-ahead” policy is exercised, then the get method does not need to be redefined at all. In the set method, the newAttrValue argument should be used only for any real-resource specific interaction: it is the GMS that updates the attribute with the new value. If there is no such interaction, the set method does not need to be redefined.

In the case of get, set, action and deleteRR, the remote managing system may have requested the atomicity of operations, in which case the calls take place twice: first with the checkOnly flag set to True and then without it. In the first pass, the specific class code should check if the requested operation can be performed. If at least one of the involved objects can not perform it, the whole series of operations is aborted. A “no-more” call then follows.

As OSIMIS is designed to operate over a single-threaded paradigm, those operations may happen asynchronously if they involve access to a remote entity. In this case, the methods should return a suitable value that signifies they will perform the operation or check asynchronously. They should then get hold of the operId argument which will serve as the “voucher” for the asynchronous result later. In the case of an action, a series of asynchronous results is possible and as such a “more” field is present in the asynchronous call back to indicate this.

Finally all of these calls may fail for any reasons related to managed object and real resource interaction: the errorInfo in that case should contain the error code and may possibly contain an additional type/value pair for a processing failure error.

It is noted that OSIMIS does not yet implement the full asynchronous API. An implementation of asynchronous actions will be supported in the next version. Finally, more detailed description for all these calls will be supplied in the next version of this manual.

```
// polymorphic methods and their arguments

RDN makeRdn (MO* superior) // GDMO compiler produced in derived classes

int  createRR (AVA*& errorInfo, void* createInfo,
                int operId);
int  deleteRR (AVA*& errorInfo, void* deleteInfo,
                Bool checkOnly, int operId);
int  get (int attrId, int classLevel, AVA*& errorInfo,
                Bool checkOnly, int operId);
int  set (CMISModifyOp setMode, int attrId, int classLevel,
                void* newAttrValue, AVA*& errorInfo,
                Bool checkOnly, int operId);
int  action (int actionId, int classLevel,
                void* actionInfo, void*& actionReply,
                Bool& freeFlag, AVA*& errorInfo,
                Bool checkOnly, int operId);
int  buildReport (int eventId, int classLevel,
                void*& eventInfo, Bool& freeFlag);
int  refreshSubordinates (AVA*& errorInfo,
                int operId);
int  refreshSubordinate (RDN rdn, AVA*& errorInfo,
                int operId);


// resulting asynchronous callbacks

int  createRRes (int operId, AVA* errorInfo);
int  deleteRRes (int operId, AVA* errorInfo);
int  getRes (int operId, AVA* errorInfo);
int  setRes (int operId, AVA* errorInfo);
int  actionRes (int operId, void* actionReply, Bool freeFlag,
             Bool moreFlag, AVA* errorInfo);
int  refreshSubordinatesRes (int operId, AVA* errorInfo);
```

General management information access methods

The management information tree is represented internally as a binary tree. A number of methods allow to walk through it and access information. In may of these, one may explicitly request to refresh the MIT before returning particular information. In that case, if the refreshing or updating involves remote communication things may go wrong. This is the reason a errorInfo parameter may be supplied by the user of those calls to be filled-in. If the user is not interested in this information, the DISCARDAVA value can be passed. The calls that do not involve MIT refreshing cannot go wrong.

If asynchronous communications are implemented by particular object classes for MIT refreshing or updating, then a special error will be returned which will mean “you cannot get this information in a synchronous fashion”. Since though it is the user’s code that will invoke those calls, it should know the semantics of the involved classes (after all, this is his/her application) and such errors should only take place during the development period.

The full set of methods that the user may use are presented below. A more detailed description on a per method basis will be supplied in the next version of this manual.

```
// General MIT access methods

int  initialiseMIB (char* mibInitFileName);
MO*  getRoot ();
DN   mn2localdn (MN mname);
DN   getAgentNameDomainDN ();

void  deleteWholeSubtree (DeletionType deleteType, Bool includingBase);

int   update (AVA*& errorInfo);
Attr* getUpdatedAttr (OID attrOid, AVA*& errorInfo);
MO*   getMO (DN dn, Bool refresh, AVA*& errorInfo);
MO*   getMO (char* dnStr, Bool refresh, AVA*& errorInfo);
MO*   getSubordinate (RDN rdn, Bool refresh, AVA*& errorInfo);
MO*   getSubordinate (char* rdn, Bool refresh, AVA*& errorInfo);
MO**  getSubordinates (Bool refresh, AVA*& errorInfo);
MO**  getWholeSubtree (Bool refresh, AVA*& errorInfo, Bool includingBase);

MO*   getSubordinate (); // first subordinate in binary internal MIT
MO*   getSuperior ();
MO*  getPeer ();

MOClassInfo* getClassInfo ();
MO*          getClassNext ();       // deprecated
OID   getClass ();
char* getClassName ();
int   checkClass (char* className, Bool* onlyActualAndAllomorphs);
int   checkClass (OID   className, Bool* onlyActualAndAllomorphs);
RDN   getRDN ();
char* getRDNString ();
DN    getDN (Bool global);
char* getDNString (Bool global);
```

Methods produced by the GDMO compiler in derived classes

A number of methods are produced by the GDMO compiler and may be used by application implementors. If a class has the “create-with-automatic-instance-naming” property, a stub for the makeRdn method is produced. If an action is present in the GDMO, the relevant stub is also produced. This is not the case regarding the set method as attributes may be settable but without associated real-resource behaviour,. The same applies to the get method as it may only be needed according to the update policy. Finally, the buildReport method is not produced as it is not needed for the object, state and relationship management notifications and will be eventually eclipsed.

The static getClassInfo method provides access to the meta-class object for a particular class while a static create method enables to create an instance. The latter will be eventually augmented with a parameter denoting the conditional packages to be created as the latter are not yet properly supported.

Finally, attributes of a particular instance may be accessed through inline methods produced by the GDMO compiler which have exactly the same name as the attribute e.g. OperationalState* operationalState() .

```
MOClassInfo*  getClassInfo ();
MO*           create (RDN rdn, MO* superior);
```

Class Top

Inherits from:	MO
Classes used:	MOClassInfo, discriminator, eventForwardingDiscriminator, log, logRecord, eventLogRecord, AttrValue (for notifications), ObjectClass, ObjectClassList, ObjId, ObjIdList (attributes)
Interface file:	Top.h
Implementation file:	Top.cc
Containing library:	gms

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 discovery of the capabilities of a MIT in a generic fashion. These attributes are the objectClass, nameBinding, packages and allomorphs.

OSIMIS supports allomorphism but packages are not yet properly supported through the GDMO compiler. The latter recognises conditional packages but does not produce (yet) the necessary code to support their dynamic creation. An interim approach that has been used is to make them mandatory. This may not always be easy as some times conditional packages are used to configure a particular instance in conflicting ways. In such cases, non-standard mechanisms should be used to control that e.g. additional “switch” attributes etc.

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).

Methods

```
class Top : public MO
{
    // ...

protected:
    int          addAllomorphicClass (OID);

public:
    Bool         hasPackage (char*);
    Bool         hasPackage (int, int);

    // interface to the general notification function

    int          triggerEvent (char*);
    int          triggerEvent (int, int);

    static int   relayEventReport (OID, MN, char*, OID, PE);
```

```
    // interface for object and state management notifications

    int          triggerObjectCreation (
                 int = int_SMI_SourceIndicator_resourceOperation,
                 char* = NULLCP);
    int          triggerObjectDeletion (
                 int = int_SMI_SourceIndicator_resourceOperation,
                 char* = NULLCP);

    int          triggerAttributeValueChange (AttrValue*,
                 int = int_SMI_SourceIndicator_resourceOperation,
                 char* = NULLCP);
    int          triggerAttributeValueChange (List*, // of AttrValue
                 int = int_SMI_SourceIndicator_resourceOperation,
                 char* = NULLCP);

    int          triggerStateChange (AttrValue*,
                 int = int_SMI_SourceIndicator_resourceOperation,
                 char* = NULLCP);
    int          triggerStateChange (List*, // of AttrValue
                 int = int_SMI_SourceIndicator_resourceOperation,
                 char* = NULLCP);

    // ...
};
```

Allomorphic behaviour

```
int  addAllomorphicClass (OID allomorph);
```

As there is no GDMO construct to specify that a class is allomorphic to another class, the implementor of the former should provide this information through this call. This method is usually called in the polymorphic MO::createRR() method for a particular class. OSIMIS accepts only parent classes of a class to be specified as allomorphs. The argument is the Object Identifier of tha allomorphic class as registered in the GDMO module. A typical way of invoking this method is e.g. in the uxObj2::createRR method through addAllomorphicClass(name2oid(“uxObj1”));

OK is returned upon success while NOTOK is returned upon failure e.g. invalid allomorph (not parent).

Package information

```
Bool  hasPackage (char* packageName);
Bool  hasPackage (int packageId, int classLevel)
```

These two methods enable to interogate a MO instance about the existence of a particular *conditional* package. The first accepts the package name while the second accepts the *packageId* label produced by the GDMO compiler for every conditional package and the class level as in many other MO class methods. Note that these are not yet implemented. When implemented, they may be used by behavioural code to tailor an instance’s behaviour according to the presence of conditional packages (if any).

General notification interface

```
int          triggerEvent (char* eventName);
int          triggerEvent (int eventId, int classLevel);
```

These methods offer an interface to trigger notifications according to the object's specification. The latter may be converted to event reports and/or specific log records according to the presence of even forwarding discriminator and log managed objects in the local MIT. This procedure is totally transparent to managed object class implementors.

They are overloaded methods that offer the same interface through the notification name or the notification id / class level. The second is mostly used by derived class behavioural code while the first may be used to trigger a notification from outside a MO instance.

As a result of calling these, the polymorphic MO::buildReport method will be called which will have to be filled with the notification information. This separation is due to historical reasons, the notification information could have simply being a void* additional parameter.

OK is returned upon success and NOTOK upon failure (invalid notification for this object).

```
static int   relayEventReport (OID moClass, MN moName, char* eventTime,
                               OID eventType, PE eventInfo);
```

Hybrid units which act in both agent and manager roles 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 MIT. 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. The parameters are as in the OSIMIS MSAP CMIS API.

Object and state management notification interface

The objectCreation, objectDeletion, attributeValueChange, stateChange and relationshipChange notifications are extremely important and as such are treated specially (the relationshipChange not yet). Special methods are provided to trigger those and they all have two common arguments: the sourceIndicator and additional text. The sourceIndicator indicates if it was a resourceOperation that triggered the notification (default) or a managementOperation. Any space for the additionalText parameter is not free'd by those calls (it is the caller's responsibility to do it, if needed). There is a number of other common optional arguments for these notifications that are not yet supported.

```
int triggerObjectCreation (int sourceIndicator, char* additionalText);
int triggerObjectDeletion (int sourceIndicator, char* additionalText);
```

These do not take any additional arguments. The full list of instance attributes will be sent with the notification.

```
int triggerAttributeValueChange (int attrId, int classLevel, void* prevValue,
                                 int sourceIndicator, char* additionalText);
int triggerAttributeValueChange (AttrValue* attrValue,
                                 int sourceIndicator, char* additionalText);
int triggerAttributeValueChange (List*, // of AttrValue
                                 int sourceIndicator, char* additionalText);
```

These take information that specifies the attribute(s) that changed and possibly the previous value. The first two support one only attribute while the third passes a list of attributes that changed. The AttrValue class is very simple and its declaration can be found in Top.h. In the first two, only the prevValue is free'd by the GMS while in the third case the whole list should have allocated space and is free'd.

The previous value should be saved before changing it to the new one through the Attr::copy() method.

```
int triggerStateChange (int attrId, int classLevel, void* prevValue,
                                 int sourceIndicator, char* additionalText);
int triggerStateChange (AttrValue* attrValue,
                                 int sourceIndicator, char* additionalText);
int triggerStateChange (List*, // of AttrValue
                                 int sourceIndicator, char* additionalText);
```

These are exactly the same as the previous ones but for state changes.

REMOTE MIB ACCESS CLASSES

Class CMISObject

Inherits from:	None
Classes/Types used:	OID, DN, AVA, AVAArray, CMISErrors
Interface file:	RMIBresult.h
Implementation file:	RMIBresult.cc
Containing library:	rmib

Introduction

The CMISObject class is a general purpose result containing class used in the RMIB API. It is used to hold a single managed object result. This class has parameters which are common to all management operations (i.e. the invoke identifier, the class name, the OID value, the instance name, the DN value, the time at which operation was performed, and the error). This class also has parameters which are specific to the type of operation (i.e. attribute value assertions in the case of a Get, Set or Create operation, and an action result in the case of an Action operation).

Methods

```
class CMISObject
{
    // ...

public:

    char*       getClass ();
    OID         getClassOid ();
    char*       getInstance ();
    DN          getInstanceDn ();
    char*       getTime ();
    AVAArray*   getAttrs ();
    AVA*        getActionRes ();
    CMISErrors  getError ();
    char*       getErrorStr ();
    AVA*        getErrorInfo ();
    int         getInvoke ();
    int         getOperation ();
    void        print ();

    CMISObject (OID, DN, char*, AVAArray*, AVA*, CMISErrors, int = 0, int = 0);
    CMISObject (OID, DN, char*, AVAArray*, AVA*, AVA*, int = 0, int = 0);
    ~CMISObject ();
};

#define NULLCMISOBJECT ((CMISObject*) 0)
```

```
char* getClass ();
```
getClass returns the class name of the managed object. The returned value is in static storage and should not be freed.

```
OID getClassOid ();
```
getClassOid returns the OID of the managed object. The returned value has memory allocated and should be freed using *oid_free*.

```
char* getInstance ();
```
getInstance returns the distinguished name of the managed object. The returned value is a copy obtained using *dn2str* and should be freed using *free*.

```
DN getInstanceDn ();
```
getInstanceDn returns the DN of the managed object. The returned value has memory allocated and should be freed using *dn_free*.

```
char* getTime ();
```
getTime returns the time (in the ASN.1 GeneralizedTime format for the OSIMIS agents) at which the operation was applied to the managed object. This parameter is optional and in cases where the agent does not support time NULLCP is expected. The returned value, if non-null, has memory allocated and should be freed using *free*.

```
AVAArray* getAttrs ();
```
getAttrs returns a pointer to AVAArray which contains the attribute value assertions. If not applicable, this method returns a NULLAVAARRAY. If the returned value is non-null, the member elements should be freed using *clear* and the array finally disposed using *delete*.

```
AVA* getActionRes ();
```
getActionRes returns a pointer to AVA which contains the action result in the type/value combination. If not valid, this method returns a NULLAVA. If the returned value is non-null, it is a copy and should be freed using *delete*.

```
CMISErrors getError ();
```
getError returns the error identifier (an enumerated type) of the error occurred when the operation was applied to the managed object. If no error, m_noError is returned.

```
char* getErrorStr ();
```
getErrorStr returns the human-readable string version of above, pointing to a static storage area and should not be freed. If no error, “noError” is returned.

```
AVA* getErrorInfo ();
```
getErrorInfo returns a pointer to AVA which contains the error code and the error information (in some cases) using the type/value combination. The error information is available *only* when the error code is either m_noSuchAttribute, m_invalidAttributeValue, m_missingAttributeValue or m_processingFailure, otherwise the error code by itself should be retrieved using either *getError* or *getErrorStr* described above. If the returned value is non-null, it is a copy

and should be freed using *delete*.

```
int getInvoke ();
```

getInvoke returns the invoke identifier used in the operation.

```
int getOperation ();
```

getOperation returns one of M_GET, M_SET, M_ACTION, M_CREATE or M_DELETE indicating the type of operation. If not available, 0 (zero) is returned.

```
void print ();
```

print can be used to pretty print the managed object's information to the standard output.

Constructors

```
CMISObject (OID oid, DN dn, char* time, AVAArray* attrs, AVA* actionRes,
     CMISErrors error, int id = 0, int op = 0);
CMISObject (OID oid, DN dn, char* time, AVAArray* attrs, AVA* actionRes,
     AVA* error, int id = 0, int op = 0);
```

Class CMISObjectList

Inherits from:	List
Classes/Types used:	CMISObject, CMISErrors
Classes related:	ListIterator
Interface file:	RMIBresult.h
Implementation file:	RMIBresult.cc
Containing library:	rmib

Introduction

The CMISObjectList class is derived from the generic List class. This class is designed to contain instances of CMISObject and used in the RMIB API. Typically, this class is used with the Get, Set, Action, and Delete operations where the result can consist of a number of managed objects. As such, CMISObjectList provides the means to hold a number of CMISObject results collectively.

Methods

```
class CMISObjectList : public List
{
    // ...

public:

    CMISErrors  getError ();
    char*       getErrorStr ();
    CMISObject* getErrorObj ();
    int         getInvoke ();
    int         getOperation ();
    void        print ();

    CMISObjectList (int = 0, int = 0);
    ~CMISObjectList ();
};

#define NULLCMISOBJECTLIST ((CMISObjectList*) 0)
```

```
CMISErrors getError ();
```

getError returns the error identifier (an enumerated type) of the first CMISObject with an error. If no error found, m_noError is returned.

```
char* getErrorStr ();
```

getErrorStr returns the human-readable string version of above, pointing to a static storage area and should not be freed. If no error found, “noError” is returned.

```
CMISObject* getErrorObj ();
```

getErrorObj returns the first CMISObject with an error. If none, NULLCMISOBJECT is returned. If the returned value is non-null, it is a copy and should be freed using *delete*.

```
int getInvoke ();
```

getInvoke returns the invoke identifier used in the operation.

```
int getOperation ();
```

getOperation returns one of M_GET, M_SET, M_ACTION or M_DELETE indicating the type of operation. If not available, 0 (zero) is returned.

```
void print ();
```

print can be used to pretty print the CMISObject elements to the standard output.

Constructors

```
CMISObjectList (int id = 0, int op = 0);
```

Class RMIBAgent

Inherits from:	KS
Classes/Types used:	OID, MN, CMISScope, CMISFilter, External, CMISSync, AVA, AVAArray, CMISObject, CMISObjectList, RMIBManager
Interface file:	RMIBAgent.h
Implementation file:	RMIBAgent.cc
Containing library:	rmib

Introduction

The RMIB Access API provides an object-oriented abstraction of OSI MIBs using the notion of an “association object”. The RMIBAgent class implements the association object, providing the high-level means of association control, string-based management operation and event reporting interfaces, and the provision of interaction in the synchronous RPC-like fashion and in the asynchronous fashion using callback facilities. The latter facilities require the use of the RMIBManager class. For the full explanation of the RMIB concept please see section ?.?.

Methods

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

public:

    // management association

    int    connect ();
    int    connect (char*, char*);
    int    disconnect ();
    int    reset (char* = NULLCP, char* = NULLCP);
    int    notifyBrokenAssociation (RMIBManager*);

    // event reporting interface

    int    receiveEvent (char*, char*, char*, char*, RMIBManager*);
    int    stopReceiveEvent (char*, char*, char*, char*, RMIBManager*);
    int    receiveEvent (char*, RMIBManager*);
    int    stopReceiveEvent (char*, RMIBManager*);
    int    receiveEvent (CMISFilter*, RMIBManager*);
    int    stopReceiveEvent (CMISFilter*, RMIBManager*);
    int    cancelTimeAssertion (RMIBManager*);

    // for the simulated time manager

    CMISEventReportArg* getCMISEventReportArg ();
```

```
// low-level management opreations interface

int    CmisGet (OID, MN, CMISScope*, CMISFilter*, External*, CMISSync,
            int, OID[], CMISObjectList*&, RMIBManager* = NULLRMIBMANAGER,
            Bool = False);

int    CmisSet (OID, MN, CMISScope*, CMISFilter*, External*, CMISSync,
            AVAArray*, CMISObjectList*&, RMIBManager* = NULLRMIBMANAGER,
            Bool = False);

int    CmisAction (OID, MN, CMISScope*, CMISFilter*, External*, CMISSync,
            AVA*, CMISObjectList*&, RMIBManager* = NULLRMIBMANAGER,
            Bool = False);

int    CmisCreate (OID, MN, int, MN, External*, AVAArray*, CMISObject*&,
            RMIBManager* = NULLRMIBMANAGER);

int    CmisDelete (OID, MN, CMISScope*, CMISFilter*, External*, CMISSync,
            CMISObjectList*&, RMIBManager* = NULLRMIBMANAGER);

// high-level management operations interface

int    Get (char*, char*, int, int, char*, External*, CMISSync, char**,
          CMISObjectList*&, RMIBManager* = NULLRMIBMANAGER, Bool = False);

int    CancelGet (int);

int    Set (char*, char*, int, int, char*, External*, CMISSync, AVAArray*,
          CMISObjectList*&, RMIBManager* = NULLRMIBMANAGER, Bool = False);

int    Action (char*, char*, int, int, char*, External*, CMISSync, AVA*,
          CMISObjectList*&, RMIBManager* = NULLRMIBMANAGER, Bool = False);

int    Create (char*, char*, int, char*, External*, AVAArray*,
            CMISObject*&, RMIBManager* = NULLRMIBMANAGER);

int    Delete (char*, char*, int, int, char*, External*, CMISSync,
            CMISObjectList*&, RMIBManager* = NULLRMIBMANAGER);

// other member functions

int    getMsd ();
Bool   isListening ();
int    getTimeOut ();
void   setTimeOut (int);
char*  getAgentName ();
void   setAgentName (char*);
char*  getHostName ();
```

```
    void   setHostName (char*);
    int    getnAll ();
    int    getnFlt ();
    int    cancelCallbacks (RMIBManager*);
    void   print ();
    static int getInvoke ();

    // provide KS callbacks to the Coordinator
    // (not to be used by the top-level application)

    int    readCommEndpoint (int);
    int    shutdown (int);

    // constructors/destructor

    RMIBAgent (char* = NULLCP, char* = NULLCP);
    ~RMIBAgent ();
};
```

Management association

`int connect ();`

This method should be used to establish association *only* if the agent and host names are available from the instance variables. OK is returned if established and NOTOK otherwise. This method should also be used to re-establish an association if the remote agent aborted in which case the agent and host names will be kept.

`int connect (char* agent, char* host);`

This method should be used to establish association to `agent` at `host` *only* if the agent and host information are not available from the instance variables. OK is returned if established and NOTOK otherwise.

`int disconnect ();`

This method should be used to release the association gracefully. All instance variables will then be restored to the default values. If an event forwarding discriminator object exists then it will be deleted. Any pending asynchronous results will be lost. OK is returned upon success and NOTOK otherwise. Once the association is cleanly terminated the object can be reused in a future association using *connect(char* agent, char* host).*

`int reset (char* agent = NULLCP, char* host = NULLCP);`

This method can be used to reset the object providing that it is not in an association. Agent and host names may be optionally supplied. This method is useful if the object had become redundant due to the remote service provider rejection or the remote agent abortion and is needed to be reused elsewhere in the application with a different set of agent/host parameters. OK is returned if successful and NOTOK otherwise.

`int notifyBrokenAssociation (RMIBManager* mgr);`

This method is part of the asynchronous service interface, and should be used to register `mgr` with the agent object such that `mgr` can be informed when the remote agent terminates or aborts. The callback required in this case is *bro-*

kenAssociation and as such this method must be redefined in the RMIBManager-derived class. OK is returned if the registration is accepted and NOTOK otherwise.

Event reporting interface

```
int receiveEvent (char* objClass, char* objInst,
     char* eventType, char* eventTime, RMIBManager* mgr);
```

This method allows the manager instance `mgr` to register to receive events of type `eventType` from the emitting managed object whose class and distinguished names are `objClass` and `objInst` respectively. The event time is in the ASN.1 GeneralizedTime format for the OSIMIS agents, and `eventTime` may be used in an expression to discriminate on this parameter. E.g., "((eventTime>=199401010000000Z) & (eventTime<=199401312359592))" to accept only those event reports generated during January 1994. Supplying NULLCP to the first four parameters is equivalent to receive all event reports *providing* that no manager is registered with a filter expression. Event reports are notified through the *eventNotification* callback of the RMIBManager-derived class. This method returns OK if the registration is accepted, otherwise NOTOK if an identical one exists or if `mgr` is already registered to receive all event reports.

```
int stopReceiveEvent (char* objClass, char* objInst,
     char* eventType, char* eventTime, RMIBManager* mgr);
```

This method should be used to cancel the registration done with the above method by providing the same arguments. OK is returned if the registration is cancelled, NOTOK otherwise.

```
int receiveEvent (char* filter, RMIBManager* mgr);
```

This method allows the manager instance `mgr` to register to receive event reports which will be subjected to the filter expression provided in `filter`. In constructing the filter expression, attributes of the eventRecord class (which inherits from eventLogRecord) should be used (i.e. managedObjectClass, managedObjectInstance, eventType and eventTime). For example, "((managedObjectClass = uxObj1) & (wiseSaying := *hello*))". Event reports are notified through the *eventNotification* callback of the RMIBManager-derived class. This method returns OK if the registration is accepted, otherwise NOTOK if there exist a manager registered to receive either using a filter or all event reports.†

```
int stopReceiveEvent (char* filter, RMIBManager* mgr);
```

This method should be used to cancel the registration done with the above method by providing the same arguments. OK is returned if the registration is cancelled, NOTOK otherwise.

```
int cancelTimeAssertion (RMIBManager* mgr);
```

This method should be used to cancel all assertions involving the `eventTime` parameter registered using the first *receiveEvent* method.

---

†. This is due to the limitations of CMIS event reporting where there is no official means of associating the distinguished name of the event forwarding discriminator object with the event report. This prevents us from identifying the correct (filter, manager) pair. To work around this problem, the only way to assert more than one filter expression is to OR the expressions into one big expression and register this using a dedicated event manager. The latter will be an RMIBManager-derived one which takes the responsibility of demultiplexing the event reports.

High-level management operations

```
int Get (char* objClass, char* objInst, int scopeType, int scopeLevel,
     char* filter, External* access, CMISSync sync, char* attrs[],
     CMISObjectList*& objList, RMIBManager* mgr = NULLRMIBMANAGER,
     Bool oneByOne = False);
```

`objClass` and `objInst` are used to specify the class and distinguished names of the base managed object respectively. `scopeType` is used to specify the type of subtree to be scoped below the base managed object (see table ?.? for the possible values). `scopeLevel` is required to indicate the level of subtree if `scopeType` is either Sc_IndividualLevel or Sc_BaseToNthLevel, otherwise 0 (zero) should be supplied. `filter` can be provided to test on the value assertions of attributes of the selected set of managed objects. *MParse*(3N) describes how to construct string filter expressions. `access` is an application defined parameter used for the access control. For the creation and deletion of this parameter please refer the MSAP manual pages. If no access control is exercised NULLMACCESS should be supplied. `sync` is used to specify the type of synchronisation required which can either be s_bestEffort (for best effort) or s_atomic (for atomic).

`attrs` should be used to specify the attributes to be retrieved where NULLCP must be given as the last entry, otherwise supply NULL (or “all” as the first entry) to retrieve all the attributes or “none” as the first entry to retrieve none. `objList` is a reference to an object list pointer that must be used to obtain the managed object results *only* if the operation is synchronous, and should be freed using delete. The `mgr` instance should otherwise be supplied if the operation is asynchronous in which case the result is returned through the *generalResult* callback of the RMIBManager-derived class. Boolean parameter `oneByOne` should be given as True if the managed object results are to be returned on the one-by-one basis through the *singleGetResult* callback of the RMIBManager-derived class. This method returns OK (synchronous case) or the invoke identifier (asynchronous case) if the request is successful, and NOTOK if failure.

Table 1: Scope types

Type of scope	Value
base managed object alone	`Sc_BaseObject`
first level subordinates only	`Sc_FirstLevel`
base managed object and all subordinates	`Sc_WholeSubtree`
individual level subordinates only	`Sc_IndividualLevel`
base managed object to Nth level subordinates	`Sc_BaseToNthLevel`

```
int CancelGet (int getId);
```

This method should be used to cancel an asynchronous Get operation issued with the invoke id `getId`.

```
int Set (char* objClass, char* objInst, int scopeType, int scopeLevel,
     char* filter, External* access, CMISSync sync, AVAArray* attrs,
     CMISObjectList*& objList, RMIBManager* mgr = NULLRMIBMANAGER,
     Bool noConf = False);
```

`objClass` and `objInst` are used to specify the class and distinguished names of the base managed object respectively. `scopeType` is used to specify the type of subtree to be scoped below the base managed object (see table ?.?

for the possible values). `scopeLevel` is required to indicate the level of subtree if `scopeType` is either Sc_IndividualLevel or Sc_BaseToNthLevel, otherwise 0 (zero) should be supplied. `filter` can be provided to test on the value assertions of attributes of the selected set of managed objects. *MParse*(3N) describes how to construct string filter expressions. `access` is an application defined parameter used for the access control. For the creation and deletion of this parameter please refer the MSAP manual pages. If no access control is exercised NULLMACCESS should be supplied. `sync` is used to specify the type of synchronisation required which can either be s_bestEffort (for best effort) or s_atomic (for atomic).

`attrs` should be used to provide the attribute value assertions which require modification (see table ?.? for the possible modify operators to be used in the AVA creations). `objList` is a reference to an object list pointer that must be used to obtain the managed object results *only* if the operation is synchronous, and should be freed using delete. The `mgr` instance should otherwise be supplied if the operation is asynchronous in which case the result is returned through the *generalResult* callback of the RMIBManager-derived class. If a non-confirmed operation is required, the optional parameter `noConf` should be supplied as True where `objList` and `mgr` should be NULLCMISOBJECTLIST and NULLRMIBMANAGER respectively. This method returns OK (synchronous case) or the invoke identifier (asynchronous case) if the request is successful, and NOTOK if failure.

Table 2: Modify operators

Type of modification	Value
set to default	`m_setToDefault`
replace current value	`m_replace`
add value (set-valued attributes only)	`m_addValue`
remove value (set-valued attributes only)	`m_removeValue`

```
int Action (char* objClass, char* objInst, int scopeType, int scopeLevel,
     char* filter, External* access, CMISSync sync, AVA* action,
     CMISObjectList*& objList, RMIBManager* mgr = NULLRMIBMANAGER,
     Bool noConf = False);
```

`objClass` and `objInst` are used to specify the class and distinguished names of the base managed object respectively. `scopeType` is used to specify the type of subtree to be scoped below the base managed object (see table ?.? for the possible values). `scopeLevel` is required to indicate the level of subtree if `scopeType` is either Sc_IndividualLevel or Sc_BaseToNthLevel, otherwise 0 (zero) should be supplied. `filter` can be provided to test on the value assertions of attributes of the selected set of managed objects. *MParse*(3N) describes how to construct string filter expressions. `access` is an application defined parameter used for the access control. For the creation and deletion of this parameter please refer the MSAP manual pages. If no access control is exercised NULLMACCESS should be supplied. `sync` is used to specify the type of synchronisation required which can either be s_bestEffort (for best effort) or s_atomic (for atomic).

`action` is used to specify the type of action requested. `objList` is a reference to an object list pointer that must be used to obtain the managed object results *only* if the operation is synchronous, and should be freed using delete. The `mgr` instance should otherwise be supplied if the operation is asynchronous in which case the result is returned through the *generalResult* callback of the RMIBManager-derived class. If a non-confirmed operation is required, the optional parameter `noConf` should be supplied as True where `objList` and `mgr` should be NULLCMISOBJECTLIST and NULLRMIBMANAGER respectively. This method returns OK (synchronous case) or the invoke identifier (asynchronous case) if the request is successful, and NOTOK if failure.

```
int Create (char* objClass, char* objInst, int instType, char* refInst,
     External* access, AVAArray* attrs, CMISObject*& obj,
     RMIBManager* mgr = NULLRMIBMANAGER);
```

`objClass` and `objInst` are used to specify the class and distinguished names of the managed object respectively. `instType` 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_SUPERIOR_INST, specifying the name of the parent object under which the object will be created (the remote agent will assign the relative object's name). If `objInst` is not specified, NULL may be used for `instType`. `refInst` is a managed object instance of the same class as the object to be created, which may be used to determine the initial attribute values, depending on the managed object class specification. It may be composed in the same way as the `objInst` distinguished name and it is an optional parameter, in which case NULLCP should be used. `access` is an application defined parameter used for the access control. For the creation and deletion of this parameter please refer the MSAP manual pages. If no access control is exercised NULLMACCESS should be supplied.

`attrs` should be used to provide the initial attribute value assertions in the creation, otherwise NULLAVAARRAY should be used for the default initialisation. `obj` is a reference to an object pointer that must be used to obtain the managed object result only if the operation is synchronous and should be freed using delete. The `mgr` instance should otherwise be supplied if the operation is asynchronous in which case the result is returned through the *createResult* callback of the RMIBManager-derived class. This method returns OK (synchronous case) or the invoke identifier (asynchronous case) if the request is successful, and NOTOK if failure.

```
int Delete (char* objClass, char* objInst, int scopeType, int scopeLevel,
     char* filter, External* access, CMISSync sync, CMISObjectList*& objList,
     RMIBManager* mgr = NULLRMIBMANAGER);
```

`objClass` and `objInst` are used to specify the class and distinguished names of the base managed object respectively. `scopeType` is used to specify the type of subtree to be scoped below the base managed object (see table ?.? for the possible values). `scopeLevel` is required to indicate the level of subtree if `scopeType` is either Sc_IndividualLevel or Sc_BaseToNthLevel, otherwise 0 (zero) should be supplied. `filter` can be provided to test on the value assertions of attributes of the selected set of managed objects. *MParse*(3N) describes how to construct string filter expressions. `access` is an application defined parameter used for the access control. For the creation and deletion of this parameter please refer the MSAP manual pages. If no access control is exercised NULLMACCESS should be supplied. `sync` is used to specify the type of synchronisation required which can either be s_bestEffort (for best effort) or s_atomic (for atomic).

`objList` is a reference to an object list pointer that must be used to obtain the managed object results *only* if the operation is synchronous, and should be freed using delete. The `mgr` instance should otherwise be supplied if the operation is asynchronous in which case the result is returned through the *generalResult* callback of the RMIBManager-derived class. This method returns OK (synchronous case) or the invoke identifier (asynchronous case) if the request is successful, and NOTOK if failure.

Other member functions

```
int getMsd ();
```

getMsd returns the Unix file descriptor used as the external communication endpoint. The returned value is positive if the object is in association and UNCONNECTED (or -1) if not.

```
Bool isListening ();
```

isListening returns True if the object is listening through the application's coordinator object which is needed for the asynchronous service interface. False is returned if not registered to listen or if the coordinator object is not instantiated.

```
int getTimeOut ();
```

getTimeOut returns the timeout interval in seconds.

```
void setTimeOut (int intvl);
```

setTimeOut should be used to change the timeout interval to `intvl` seconds which must not be less than the initial default value of thirty seconds.

```
char* getAgentName ();
```

getAgentName returns the agent name. A valid string returned does not imply that the object is in association. NULLCP is returned if the instance variable containing the agent name is not set.

```
void setAgentName (char* agent);
```

setAgentName should be used to overwrite the instance variable containing the agent name with `agent`.

```
char* getHostName ();
```

getHostName returns the host name. A valid string returned does not imply that the object is in association. NULLCP is returned if the instance variable containing the host name is not set.

```
void setHostName (char* host);
```

setHostName should be used to overwrite the instance variable containing the host name with `host`.

```
int getnAll ();
```

getnAll returns the number of RMIBManager objects registered to receive all event reports.

```
int getnFlt ();
```

getnFlt returns the number of RMIBManager objects registered to receive filtered event reports.

```
int cancelCallbacks (RMIBManager* mgr);
```

cancelCallbacks should be used to cancel all registered callbacks of `mgr`. See also *cancelTimeAssertion* above.

```
void print ();
```

print can be used to pretty print the RMIBAgent instance to the standard output.

```
static int getInvoke ();
```

getInvoke returns the invoke identifier used in the last operation.

Constructors

```
RMIBAgent (char* agent = NULLCP, char* host = NULLCP);
```

The only instance variables which may be set during instantiation are the logical application name (`agent`) and the host name (`host`). If these values are supplied then association can later be established using *connect()*. Otherwise, *connect(char* agent, char* host)* must be used.

Class RMIBManager

Inherits from:	None
Classes used:	Attr, RMIBAgent, CMISObject, CMISObjectList
Interface file:	RMIBManager.h
Implementation file:	None
Containing library:	rmib

Introduction

The RMIBAgent class provides a number of asynchronous services such as the notification of event reports, broken association, and management operation results. These services can *only* be used by the management application through the callback methods of the RMIBManager class. The latter is an abstract class which must be specialised to get the required manager class in order to use one or more of the asynchronous services provided by the RMIBAgent.

Methods

```
class RMIBManager
{
public:

    virtual int eventNotification (char*, char*, char*,
                        char*, Attr*, RMIBAgent*);
    virtual int generalResult (CMISObjectList*, RMIBAgent*);
    virtual int singleGetResult (CMISObject*, RMIBAgent*);
    virtual int createResult (CMISObject*, RMIBAgent*);
    virtual int noTimeAssertion (RMIBAgent*);
    virtual int brokenAssociation (RMIBAgent*);

    virtual ~RMIBManager ();
};

#define NULLRMIBMANAGER ((RMIBManager*) 0)
```

Polymorphic callback events

```
virtual int eventNotification (char* objClass, char* objInst,
     char* eventType, char* eventTime, Attr* eventInfo, RMIBAgent* agent);
```

eventNotification should be redefined to receive event report notifications from the remote agent represented by `agent`. `objClass` and `objInst` are the class and distinguished names of the emitting managed object from which an event of type `eventType` is notified at `eventTime` (in the ASN.1 GeneralizedTime format for OSIMIS agents). `eventInfo` contains the event report in the type/value combination and should be decoded accordingly. The return value should be either OK or NOTOK.

```
virtual int generalResult (CMISObjectList* objList, RMIBAgent* agent);
```

generalResult should be redefined to receive asynchronous results of the Get, Set, Action, and Delete operations issued to the remote agent represented by `agent`. `objList` contains the result managed objects and should be freed using *delete*. The return value should be either OK or NOTOK.

```
virtual int singleGetResult (CMISObject* obj, RMIBAgent* agent);
```

singleGetResult should be redefined to receive asynchronous results of a Get operation issued with the one-by-one delivery option to the remote agent represented by `agent`. `obj` contains the managed object result and should be freed using *delete*. The return value should be either OK or NOTOK.

```
virtual int createResult (CMISObject* obj, RMIBAgent* agent);
```

createResult should be redefined to receive the asynchronous result of a Create operation issued to the remote agent represented by `agent`. `obj` contains the result and should be freed using *delete*. The return value should be either OK or NOTOK.

```
virtual int noTimeAssertion (RMIBAgent* agent);
```

noTimeAssertion should be redefined with an appropriate behaviour to allow `agent` to inform that the remote agent which it represents does not support time. For instance, in event reporting, the CMIS event time field is optional and in such cases filters with event time discrimination, if any, will not evaluate correctly and should therefore be cancelled. The return value should be either OK or NOTOK.

```
virtual int brokenAssociation (RMIBAgent* agent);
```

brokenAssociation should be redefined with an appropriate behaviour to take action if the remote agent represented by `agent` terminates or aborts. One appropriate action is to reset `agent` and reuse the object elsewhere in the application. The return value should be either OK or NOTOK.