The MGA library

Arpad Bakay
ISIS, Vanderbilt University
September 2000

Table of contents

Introduction

The purpose of this work is to document the concepts, the architecture and the functionality of the MGA library, which is a key component of the GME 2000 modeling infrastructure [1].

The MGA library (**MgaLib**) is used to build hierarchical networks of objects that represent the modeling concepts described in the GME Tool documentation [2]. The relevant part of this object model is also described below (1.2). Most of the objects, their methods, and their properties closely correspond to those concepts, thus working with them will probably be straightforward to people with some experience in modeling. There are, however, some additional library objects that are not part of those concepts, but rather control the operation of the whole library, in particular the consistent multi-user access to the modeling database.

1. General information on the MGA library

1.1 Components of the GME 2000 architecture

The MGA library is a component of the Generic Modeling Infrastructure (fig. 1), surrounded by other important modules:

- As modeling relies on a domain-dependent modeling paradigm (metamodel), the MGA library works in close cooperation with the **MetaLib** module, which provides (in this case read-only) access to the metamodeling information. In most cases the user of the MGA library will need to access the Meta library directly as well.
- Both MgaLib and MetaLib use the object oriented database services provided by the **CoreLib** module. Among other things (like transactions and undo/redo functions), the Core completely and uniformly wraps the physical database format actually used for persistent storage, thus protecting the upper layers from having to deal with issues related to different databases.

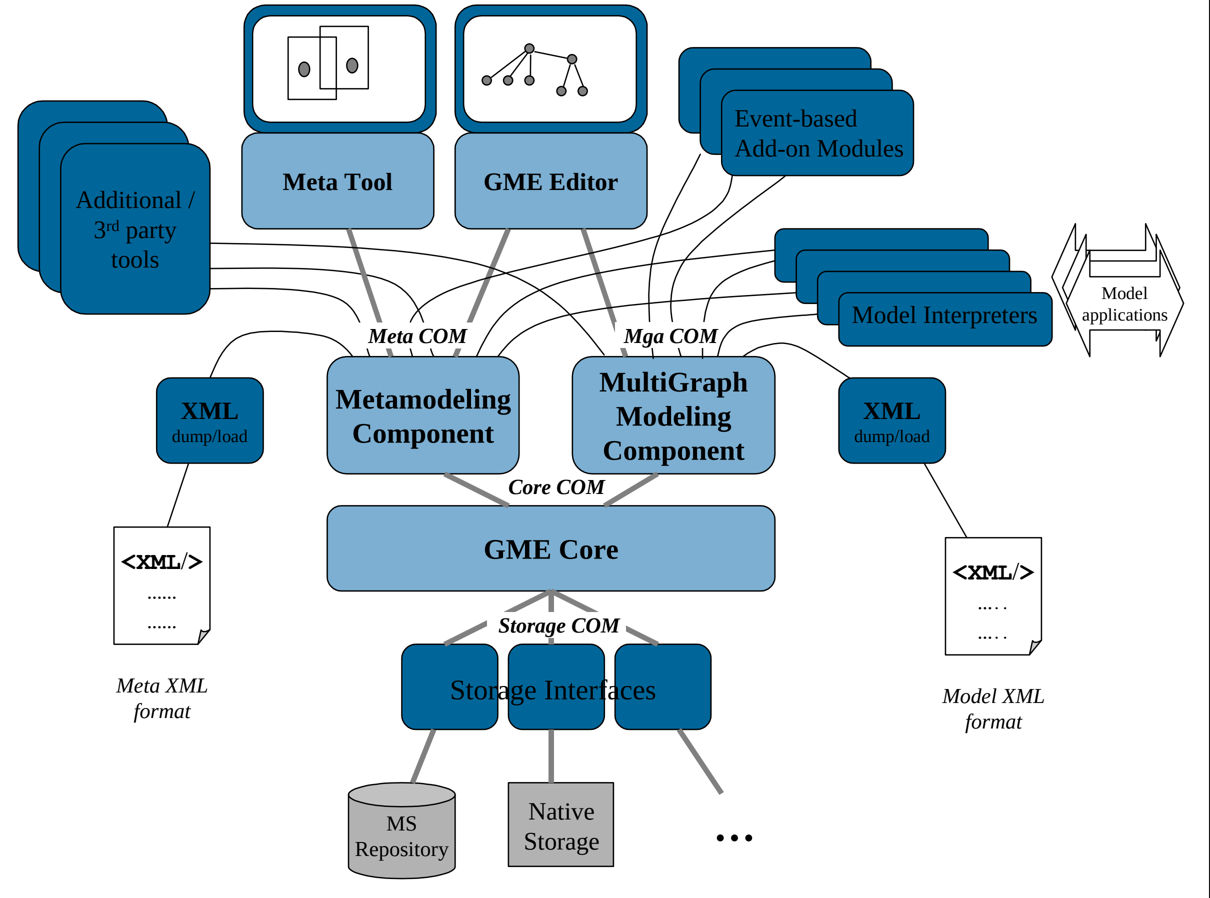

Fig. 1. GME 2000 Component Architecture

- The fourth basic GME module, the **GME GUI** (which also includes the GME browser) uses the services of the MGA and Meta libraries. It is important to recognize that the GUI component uses the exact same interfaces that are available for the other modules.
- The most typical user-implemented modules in the architecture are **Interpreters** that perform some (either read or read-write) operations on the MGA library. They may be paradigm-specific or paradigm-independent tools. Especially in the later case, the interpreter usually needs to access the Meta library as well.
- **Add-ons** are the other type of user-implemented modules in the architecture. Add-ons are executed in an event-based fashion, i.e. they are triggered by changes to the database. Apart from this special event-based execution mechanism, the access to the library by add-ons and interpreters is very similar.
- The architecture includes **XML dump and load facilities** for both the modeling and paradigm information.

The COM interface is the primary interface to the MGA library. Others (like the Builder Object Network) are implemented above the COM layer. The library is not only COM-compatible, but also closely follows the current COM technology-related guidelines (e.g., interface and method naming, parameter types, error handling). Consequently, it is easy to work with from Visual Basic, and it also appears familiar to the more experienced COM programmer.

1.2 The generic MGA object model

1.2.1 Core concepts

The primary function of an MGA database is to model different real-world scenarios for different problem domains with a simple uniform set of concepts. The basic modeling element types are called *atoms* and *models*. The rules for combining these elements are defined by *metamodel* (paradigm), a set of domain-specific concepts and rules that is considered invariant at model-building time.

Elements have some predefined (built-in) *properties.* The two most important properties are the *kind* (what kind of object this element represents) and the *name* (which may be empty in case of ‘nameless’ objects). In addition to elements, both element types can have a set of *attributes*. The number, name, and datatype (e.g. string, number) of the attributes defined for an element of a given kind is defined by the metamodel.

Atoms represent concepts that only have attributes, but do not have their internal structure modeled in further detail. Models do have structure; to represent this, models may *contain* atoms and further models. This way a *hierarchy* of elements is built where subsequent layers provide a more and more detailed representation of the modeled system.

In addition to its own type, a contained object always has a defined role in its container (e.g. an element of type 'room' can function as a 'livingroom', a 'bedroom' or a 'dining room' within a house). The rules of combining different parts to form composite models are also defined by the metamodel.

Elements may be *related* to each other. MgaLib defines three types of relations:

- *Connections* connect 2 or more objects where each of the participating objects has a defined role (e.g. husband and wife).
- *References* allow referrals (pointers or aliases) from a model to another distant element.
- *Sets* refer to arbitrary number of objects, thus forming a collection of members.

The rules for relations (what can relate to what) are defined in the metamodel. Like atoms and models, relations are also contained (owned) by models, although some of them (references and connections) can refer to elements outside of the scope of their parent. For maximal simplicity, the MGA object model defines relations as new types of elements. Like atoms and models, they can have attributes, and they are assigned roles in their parent models. Furthermore, relation elements can be targets of other relations as well. Again, all of these capabilities are controlled by rules set forth in the metamodel.

In summary, the final set of element types contains models, atoms, references, connections and sets. These are also the principal objects of the MgaLib interface, usually called FCO's (*first-class objects).*

1.2.2 Other concepts of the MGA object model

- As real-world objects are often represented and analyzed in different ways (e.g. multiple views), a model can also have multiple *aspects*, where each aspect reveals a different subset of the child elements. An element as seen in an aspect of its parent model is also called *part.*

- Although the metamodel defines the basic rules of building a model hierarchy, models and other elements may also be associated with additional *constraints* that safeguard the consistency of the model. Constraints are usually expressed in the form of predicate expressions. Valid models must simultaneously satisfy all of their constraints.

- The MGA object model makes it possible to associate a hierarchical structure of free-form information with any of the element. This is the object *registry*, described in detail later (3.5).

- The model of a real-world system is not necessarily a single top-level entity, but may consist of several disjoint parts. These are called root modeling objects or *rootFCO's*. If their number is large, or if they are conceptually different, it may be useful to organize them into different groups. This is made possible by the use of a hierarchical system of *folders.* Folders can contain folders and rootFCO's. The rules of the folder structure are also partially defined by the metamodel. Still, folders must not be mistaken for models or other modeling elements: their hierarchy is not used for modeling any real-world system, but they are simply containers for conveniently organizing parts of a model.

1.3 General notes on the MGA library

The primary source of information on the library is the MGA IDL file. This file defines all the interfaces used between the library and the client module (but not the interfaces used internally between the MGA and the Meta and Core libraries). There are a total of 25 interfaces defined. The IDL also defines data types and constants used by some of the interface methods.

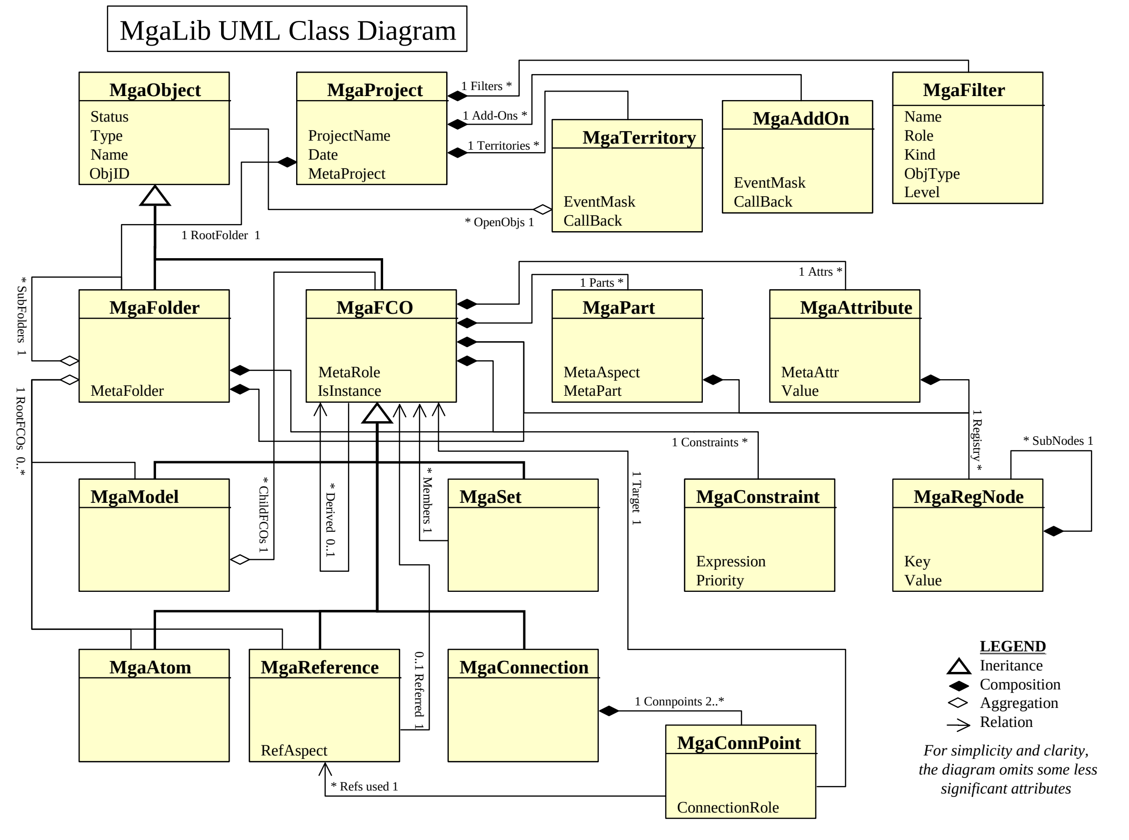

Fig. 2.

The UML class diagram on Fig. 2 gives a first look at 18 of the COM interfaces provided by the library. This is practically a complete picture, though, since most of the other interfaces are simple collection interfaces to the ones in the figure (described at 2.5), and the single remaining one, IMgaEventSink, is only defined by the IDL, but needs to be implemented by the client as a callback interface.

2 Basic MgaLib operations

2.1 Starting an MgaLib session

MgaProject is the fundamental point of access to the MGA library. This is the only creatable object; that is, all the other objects are accessed as return values of methods of MgaProject or other objects retrieved earlier. Most applications will need to use a single MgaProject object only, unless they need to work with several databases simultaneously.

The MgaProject object is first obtained through an operating system call (`CoCreateInstance)`. For this call to succeed, the MGA library must be properly registered in the operating system (e.g. by the `regsvr32` command).

Once the project object is available, the next step is either to Create or to Open a database using the corresponding methods of IMgaProject. Both methods use *datasource identifiers*: a datasource type (‘`MGA=’` for binary, `’DBQ=’ ’DNS=’` for MS repository storage), and a datasource name. In case of native and DBQ storage, the datasource name is a regular filename (e.g.: “DBQ=\gme\data.mdb”).

MgaProject.Open takes a single parameter, the datasource ID. MgaProject.Create also requires a second parameter to specify the Meta datasource for the newly created modeling project.

VB example

```
Dim project As New MgaProject
project.Create(”MGA=C:\GME\Proba.bin”,”MGA=C:\GME\Meta.bin”)

...

project.Close();
```

C/C++ example (without ATL)

```
#include <windows.h>
#include <tchar.h>
#include <objbase.h>
#include “mga.h”     // or “gme.h”


IMgaProject *project;
CoInitializeEx(NULL,COINIT_APARTMENTTHREADED);
HRESULT hr;
hr  =  CoCreateInstance(CLSID_MgaProject,  NULL,  CLSCTX_ALL,  IID_IMgaProject,
                                   &(void *)project);
if(hr != S_OK) { ... }  // handle error
BSTR data = SysAllocString(OLESTR(”MGA=C:\\GME\\Proba.bin”));
BSTR meta = SysAllocString(OLESTR(”MGA=C:\\GME\\Meta.bin”));
if(!meta || !data) {  ...  }  // handle error
hr = project->Create(data, meta);
if(hr != S_OK) { ... }   // handle error
```

```
SysFreeString(data);
SysFreeString(meta);

...       // work on database (create territory and transactions)

hr = project->Close();
if(hr != S_OK) { ... }   // handle error
project->Release();
```

C++ example (with ATL)

```
#include <windows.h>
#include <tchar.h>
#include <objbase.h>
#include <atlbase.h>
#include "mga.h"     // or "gme.h"

#define HR_CHECK(hrop) if((hrop) != S_OK) throw "MGALib returned error"

...

try {
  CComPtr<IMgaProject> project;
  CoInitializeEx(NULL,COINIT_APARTMENTTHREADED);
  HRESULT hr;
  HR_CHECK(project.CoCreateInstance( OLESTR("Mga.MgaProject")) );
  HR_CHECK(project->Open( CComBSTR("MGA=C:\\GME\\Proba.bin"));

  ...       // work on database (create territory and transactions)
  ...       // preferrably within another try block

  hr = project->Close();
  if(hr != S_OK) { ... }   // handle the very unusual close error differently
}  // 'project' is automatically released at this point!!!
catch(…) {
   // most errors can be handled here
}
```

Note that the first C/C++ example shown uses normal pointers instead of smart pointer classes defined in the Advanced Template Library (ATL). As the next example demonstrates, using smart pointers is definitely easier, safer, and thus the recommended way of COM programming. For this reason, apart from this first baseline example (which is meant to be readable without any ATL knowledge), all the following examples will use ATL smart pointers only. ATL is not available for 'C' programming.

Once these steps are executed successfully, the MgaProject object refers to an open project. The project remains open until MgaProject.Close is executed, which returns the project to an unopened state. Then it is ready again for opening another database, or for releasing the MgaProject object. (Releasing the project object in its open state also results in closing the database, but this practice does not allow for error checking in the close operation).

2.2 Transactions

The session continues in the form of *transactions:* a transaction is the invocation of the BeginTransaction() method followed by a series of operations and finally closed by a call to either the CommitTransaction() or AbortTransaction() methods. Commit finalizes all changes made during the transaction, while Abort returns to the initial state valid at the beginning of the transaction. In any case, the atomicity of the operation is preserved. Each transaction must be closed by a successful Commit or Abort before the next one is initiated. In practice, this means that if Commit fails, an Abort still needs to be executed. Aborts usually succeed: their failure indicates deep and severe run-time (e.g. communication) error conditions, which do not leave many options to protect the database anyway. After a successful commit or abort, the project is ready for another transaction to be executed.

MgaLib allows several clients to work on the same MgaProject object simultaneously. However, the current implementation allows only one transaction to be active at a time. Consequently, it may happen that a BeginTransaction call fails (E_MGA_ALREADY_IN_TRANSACTION) because the project is already in a transaction initiated by another client. The client then must wait until the previous transaction is closed.

Clients are identified by their *territories* (MgaTerritory objects), and BeginTransaction() requires an MgaTerritory parameter to be specified. Although some complex clients (like the GME GUI) use several territories, most clients usually create a single territory and keep using it throughout their execution. Territories are created by the MgaProject.CreateTerritory() method. Concurrency and territories are described in more detail later (see Section 4).

There are two types of transactions: *general*, which gives read-write access to the database, and *read-only*. In the current MgaLib version, neither transaction type is allowed to overlap, so read-only is really a limited access version of the general one. The requested transaction-type is specified as an optional second parameter to MgaProject.BeginTransaction().

The following example shows the use of territories, transactions, and some basic operations. The latter are described in the next section (2.4).

VB example

```
'. . . project is already open
Dim terr as IMgaTerritory
project.CreateTerritory Nothing, terr
project.BeginTransaction terr
Dim ff As MgaFolder
Set ff = project.RootFolder
Dim str As String
str = ff.Name
project.CommitTransaction
```

```
MgsBox str
```

C++ example (with ATL)

```
// . . . proj is already open
#define HR_CHECK(hrop) if((hrop) != S_OK) throw "MGALib returned error"
try {
    CComPtr<ImgaTeritory> t;
    // create the territory
    HR_CHECK(proj->CreateTerritory(NULL, t));
    // start first transaction
    HR_CHECK(proj->BeginTransaction(t));
    try {
       CComPtr<IMgaFolder> ff;
       HR_CHECK(proj->get_RootFolder(&ff));
       CComBSTR name;
       HR_CHECK(ff->get_Name(&name));
       name.Append("-changed");
       HR_CHECK(ff->put_Name(name));
       HR_CHECK(proj->CommitTransaction());
    }
    catch(…) {
       HR_CHECK(proj->AbortTransaction());
    }
    // ready to begin another transaction:
    HR_CHECK(proj->BeginTransaction(t));
    try {
       // other MgaLib operations
    }
    catch(…) {
       HR_CHECK(proj->AbortTransaction());
    }
    // territory t is automatically destructed here
}
catch(…) {
}
// proj is still open, ready for further operations or close
```

2.3 Error handling

As seen in the examples above, all COM calls are checked for error. Indeed, practically every library call can go wrong.

Failures can originate at different levels:

- In the COM interface between the client and the MGA library. In these cases, MgaLib usually does not even get called before an error is returned.
- In the MGA library. This may involve either an invalid operation (e.g. an object was to be created where it is not allowed), temporal unavailability (transactions, locks), or an error in the consistency of the underlying database.

- Below the MGA library. This means either that one of the underlying modules failed, or that the communication between MgaLib and one of those modules had an error.

All of these errors are signaled in the HRESULT return value of the COM calls. Some of them are set by the operating system, some by the MgaLib, and others by the underlying modules. The categories are distinguishable by error codes: codes below 0x80000000 are from the operating system (use Microsoft Error Lookup for error descriptions), those between 0x87650000 and 0x8765FFFF are returned by MgaLib, and those between 0x80730000 and 0x8073FFFF come from other GME modules.

Under normal conditions (if there are no network or OS problems and the underlying database has no defects), the only errors signaled will be those coming from the MGA library. There are two techniques for decoding error codes:

- For each of the error codes, the 'mgaerrors' enum definition in the MGA.IDL file lists a mnemonic (like E_MGA_ALREADY_IN_TRANSACTION, E_MGA_TARGET_DESTROYED) and a short description text. (If an error is not found in this section, then it comes from another component).
- The library also supports runtime error information. Visual Basic provides built-in support to signal this information in the normal error-handling mechanism (e.g. error message window). To access it from C++, the application must request the ISupportErrorInfo interface of the object and test its InterfaceSupportsErrorInfo method. If both of these steps succeed, then the current error information is accessible from the client thread by calling the ::GetErrorInfo() global function.

If MgaLib returns an error, it means that the operation had no effect on either the state of the MgaLib or the contents of the database. The only exceptions are MgaProject.Close() and MgaProject.AbortTransaction(), where error returns leave the library in an undefined state.
After an error, the client can decide to

- repeat the operation (especially if MgaProject.BeginTransaction() returns E_MGA_ALREADY_IN_TRANSACTION),
- do some other operations (e.g. when E_MGA_META_VIOLATION is returned),
- commit or abort the current transaction if the failed operation was executed in a transactional context.

The example above also illustrates a fairly reliable method to handle errors in C++ through the *throw* mechanism. This structure can be further improved by defining and by using smart transaction objects, which can control commit or abort automatically (no example provided in this documentation).

In case of lower-level errors, it is hard to say what to do. Aborting the current operation and immediately closing the database is usually the safest (although not always feasible) solution. If the library works over a repository database, successful commits are guaranteed to leave the database in a consistent state until a following commit begins. If the MGA native database is used, however, data is not written to the disk until the database is closed.

2.4 Navigating the object tree

As seen earlier in this chapter, the MgaProject object is obtained from the operating system (or from another MgaLib client that shares its own copy). This is a unique MgaProject feature, since all the other objects are accessed by invoking methods of already existing ones. This chain of access starts with the single MgaProject available initially.

The set of MgaObjects in the library form a tree-like hierarchy: the hierarchy of Folders spans the top (the tree is usually drawn upside down, so ‘top’ means ‘closer to the root’) levels, followed some layers of models, ending with other (non-container) objects at the terminal nodes. The relation between adjacent layers is called *containment* or *parent-child* relationship. While all other types of relationships are optional, containment is guaranteed to connect the whole set of objects into a tree. This makes it the primary technique for navigating among the objects.

The root node of the tree (by definition, always a folder) is accessed by the MgaProject.RootFolder() method. To navigate ‘down’, either of the two child collection access properties (ChildFolders or ChildFCOs) is used. To move up, the parent can be obtained through the MgaFolder.ParentFolder property or the more general MgaObject.GetParent method, which returns a MgaFolder or a MgaModel depending on the location of the object. Besides these, there are other, more specialized and/or child access methods as well (e.g. the ones that use MgaFilter). Of course it is also possible to navigate the object tree along the other types of relations (e.g. references, inheritance, etc.).

2.5 Working with MgaLib COM collections

Some of the functions described above (like MgaFolder.ChildFolders and MgaFolder.ChildFCOs) return collections of object interfaces (MgaFCOs and MgaFolders). There are numerous other MgaLib functions that use collections, either as input or as output parameters. To make working with collections easy, MgaLib (and all the other GME2000 components) provides a uniform programming interface to all the collection interfaces.

Collections are easily recognizable by their consistent naming: IMga*xxxx*s is a collection of IMga*xxxx* interfaces. Collection interfaces are somewhat hidden in the IDL file, since their declarations are generated by a preprocessor macro, which is included from InterfaceColl.h. The member functions supported by these macro-base interfaces are as follows:

Count() property:	returns the size of the collection
Item(nnn) property:	returns the 'nnn'-th item. The base of the indexing is 1!
NewEnum() property:	this function is supported for Visual Basic compatibility.
GetAll() method:	this method makes it possible to access all members of a collection by a single COM call. The two input parameters are the requested count of objects, and a memory buffer large enough for storing this number of pointers. The output parameter is the number of objects actually returned, which normally matches the requested number.

Accessing collections from Visual Basic is possible through the built-in 'For Each' enumeration construct. Alternatively, the Item() and Count() properties are also accessible.

The GetAll() function, however, is not compatible with the general VB interface conventions. Its function is to provide an efficient access method from C/C++, as shown in the following example.

While the Visual Basic example is fairly simple, the C++ code is rather lengthy and complex. Unfortunately this is about the simplest way in C++ for working with collections in a reliable way. The smart array pointer implemented by the template makes it sure that references are released even in case of errors – a problem for which no other simple solution is available. Fortunately it is possible to encapsulate most of the enumeration code into a few macros, which will eventually make the code for C++ collection iterations almost as simple as it is seen with Visual Basic.

VB example

```
'. . . transaction is already open
Dim ff As MgaFolder
Set ff = project.RootFolder
Dim childfolders As MgaFCOs
Set childfolders = ff.ChildFolders
Dim sf As MgaFolder
' traverse using 'For Each'
For Each sf In childfolders
     Dim childfcos As MgaFCOs
     Set childfcos = sf.ChildFCOs
' use Count and Item() here:
     If childfcos.Count > 0 Then
        MsgBox  childfcos.Item(1).Name    ' 1-based access!!!!
     End If
Next
```

C++ example (with ATL)

```
template <class TYPE>
class COMPtrArray {
public:
    CComPtr<TYPE> *f;
    COMPtrArray(int length) { f = new CComPtr<TYPE>[cnt]; }
    ~ COMPtrArray() { delete []f;   }
}

// . . . transaction is already open
#define HR_CHECK(hrop) if((hrop) != S_OK) throw "MGALib returned error"

try {
   CComPtr<IMgaFolder> ff;
   HR_CHECK(proj->get_RootFolder(&ff));
   CComPtr<IMgaFolders> subfolders;
   HR_CHECK(ff->get_ChildFolders(&subfolders));

   long count;
   HR_CHECK(subfolders->get_Count(&count));
   COMPtrArray<IMgaFolder> folderarray(count);
   unsigned long result = 0;
   if(count) HR_CHECK(subfolders->GetAll(count, folderarray.f, result));
   if((unsigned long)count != result) throw("Error");
   // access members of array by iterating through it
   for(int i = 0; i < count; i++) {
        CComBSTR name;
        HR_CHECK(folderarray.f[i]->get_Name(&name));
        MessageBox(name);
   }  // folderarray is destructed, COM pointers are released at this point
}
catch(…) {
}
```

2.6 <u>Creating, querying, and modifying objects</u>

Objects are created one at a time by calling methods of their would-be parents. The object type is specified by a reference to a meta object, the type of which depends on where the object is created:

- MgaMetaFolder for MgaFolder.CreateFolder
- MgaMetaFCO for MgaFolder.CreateFCO
- MgaMetaRole for MgaModel.CreateChildFCO (roles uniquely specify the object FCO kind as well)

After a successful Create*xxx* operation, the object is ready for read-write operations. The most basic ones are as follows:

- Setting and reading the object name through the **Name** property.
- Setting and querying attributes of FCO's by any of the attribute access functions. The simplest way to read or write attributes is through the **AttributeByName** property. Attributes are also accessible through MgaAttribute subobjects, described in detail at 3.4.
- Setting and reading the Registry of the object (see 3.5 for details)
- Creating further objects in models and folders using the methods described.
- Deleting an object by the **DestroyObject** method. If the object has children, all contained objects are also destroyed.

- Creating reference, connection, or set relations. These operations are described in the next section (2.7).

Note: Most MgaLib interfaces define a mixed set of properties and methods. The difference is obvious in Visual Basic, where properties are somewhat more convenient to use, since they can be directly used within expressions and as the left-hand side of assignment statements (depending on whether the property is readable, writable or both). In C++, the IDL compiler translates both of them to class member functions: methods retain their original name, but property reads and writes are prefixed with ‘get_’ and ‘put_’ respectively (e.g.: MgaObject.Name translates to IMgaObject->get_Name and IMgaObject->put_Name).

2.7 Relations: references, connections and sets

References, connections, and sets define some additional methods over the ones available for FCO’s. These methods make it possible to access the relations that these objects represent. Relation objects may be created empty (using the ImgaModel.CreateChildFCO() function), but two further methods (MgaModel.CreateReference and MgaModel.CreateSimpleConn) allow creation of initialized relations as well.

All relations are subject to extensive checks on creation or modification. Objects are checked to conform to the specifications described in the meta, which include the pointer specifications assigned to the corresponding metaobjects. In some cases (e.g. non-simple connections), it is inevitable to have inconsistent states while an object is not yet fully built. To handle this situation, MgaProject allows the temporary suppression of pointerspec checks through the MgaProject.CheckSupress(VARIANT_TRUE) call. The suppressed tests are not completely canceled, but postponed until either MgaProject.CheckSupress(VARIANT_FALSE) is called or the current transaction is committed. If the test fails, this later operation will return an error. The user then must either correct the offending relation, or abort the entire transaction.

Queries and subsequent modifications are available through the following functions:

References:

The MgaReference.Referred property sets or retrieves the object referenced (or NULL for empty references). If the reference relation is used by connections (i.e. there are connections going through it), its target cannot be changed. The corresponding points of these connections (see below for explanation on ConnectionPoints) are available through the UsedByConns property.
The MgaReference.RefAspect property stores an MgaMetaAspect value for model references. It is normally NULL, which means that the aspect of the reference target is determined automatically.

Sets:

Sets are queried by the Members and IsMember properties and modified by the AddMember, RemoveMember and RemoveAll methods. Each object is only included once in the set, no matter how many times it was added by AddMember(). RemoveMember, however, checks for the membership of the object to be deleted and returns E_MGA_OBJECT_NOT_MEMBER if the check fails.

Connections:

All connections implement the IMgaConnection and IMgaSimpleConnection interfaces. These interfaces support query and modification through MgaConnectionPoint subobjects.

IMgaModel::CreateSimpleConnDisp is the preferred way to create connections. Srcref and Dstref define the refport parent, if non-NULL.

For complex cases, new connection points are established through the IMgaConnection::AddConnPoint method. Parameters of this method include the endpoint role name (“src” or “dst”), the target object, and, if the connection end is to be a refport, a list of references. This reference list is ordered: the first reference contains the refport, the second is the referee of the first, the third is the referee of the second, and so on. The maxinrole argument to AddConnPoint specifies a limit of already existing endpoints with the same name. If this limit is exceeded, an error (E_MGA_CONNROLE_USED) is returned. Connection points provide read-only access to their target, references, and their owner connection object, and read/write access to the connection role.

Endpoints can be queried and set directly through the IMgaSimpleConnection interface (Src, Dst, SrcReferences and DstReferences properties and SetSrc, SetDst methods).

Reverse queries:

FCO’s may be interested which relations refer to them. This information is available through the IMgaFCO.ReferencedBy, IMgaFCO.MemberOfSets, and IMgaFCO.PartOfConns properties, which return a collection of related FCOs or ConnPoints. The UsedByConns property of MgaReference, which returns the ConnectionPoints going through a reference relation, also belongs to this category.

2.8 The lifecycle of object and subobject interfaces

Although the library returns pointers to existing objects/subobjects only, the life of those may be shorter than the time a client keeps the references to them. This often happens unexpectedly, especially in the following cases:

- An object (or attribute) is created and its COM pointers are acquired (AddRef-d), but the transaction finally aborts, or the transaction is later undone.
- An object is destroyed by a client in a transaction while other clients still have interface references to it.

The consequence to all this is that nonexistent (destroyed or not yet created) objects may have references at some point in time. Moreover, if the deletion is undone later, the same pointers are expected to be available to access the undestroyed objects as if nothing had happened.

To represent this situation, objects and subobjects can have so-called ‘deleted’ and 'zombie' states. These exist while there are references to a deleted object. If the deletion of an object is undone (or the transaction including the delete is aborted) the object is reborn. If the delete operation cannot be undone, the object enters the ‘zombie’ state.

The status of an object can be queried by IMgaObject.Status(), which returns OBJECT_EXISTS, OBJECT_DELETED or OBJECT_ZOMBIE respectively. In addition to that, ‘deleted’ state provides a limited access to the object: it can be checked for its name, ID, Meta, MetaRole and MetaBase, but all other types of access return an error. If an object is deleted, the corresponding subobject pointers also enter the ‘deleted’ state. During this time, they can still return their owner object (Object property), but all other operations return E_MGA_OBJECT_DELETED error. Once the object becomes a zombie, only IMgaObject.Status() is available, all other calls (including calls to subobjects) return E_MGA_OBJECT_ZOMBIE.

While the MGA interface supports deleted and zombie states for objects and subobjects alike, it is generally an arguable programming practice to keep pointers to subobjects between transactions. In other words, subobject references are best used as short-term references. These considerations do not apply for Object interface references, where keeping object references between compound operations is generally acceptable, and the notification mechanism (described at 4.4) is a reliable way to detect deletions.

3 Complex operations

3.1 Inheritance

The inheritance mechanism provided by the MGA library is designed to support the efficient composition and subsequent modifications of models. It allows the definition of *types*, from which *instances* or *subtypes* can be derived. A type can be a model hierarchy (i.e. a model with all its children, grandchildren, etc.) of any size and complexity. There are some obvious limitations, however:

- a type must be a root object (i.e. contained directly by any folder)
- a type cannot contain an instance of itself or any of its subtypes.

In fact, when a hierarchy is created, it does not need to be explicitly changed to become a type: every root FCO can by default be used as a type unless it is derived as an instance.

Instances are derived from types by a single library call. They are deep copies of their types, i.e. each contained object is replicated. They can both be root or non-root objects. Inside the instances, no new objects may be added to the structure defined by a type, but relations defined by references, connections and sets may be modified. Thanks to the defaulting mechanism detailed at the description of the attributes and the registry (3.4 and 3.5), instances inherit the attributes and registry data of their types, but this can be overridden any time later.

Subtypes are similar to instances, but they are less restricted in that their structure can be extended by adding new objects. Objects inherited from the type (which is called *basetype* in case of subtypes) cannot be deleted, however. If a subtype is a root object, it can be used to derive further subtypes as instances, thus forming an *inheritance chain*. Although they cannot be used as (base)types in Derive*xxx* commands, non-root object subtypes (and also instances!) may also have objects derived from them if they become part of a larger hierarchy that is used as a type.

This way, any single object may have a chain of objects that serve as types for each other finally producing the object itself. However, this *inheritance chain* is linear and multiple inheritance is not possible, i.e. each object can only have a single object it is immediately derived from.

Comparable to the containment tree, inheritance defines another not so obvious, but very important tree structure among the modeling objects. The MGA library provides a set of methods to discover and change these relations.

Creating subtypes and instances

DeriveChildObject (for models) and DeriveRootObject for (folders) are the principal methods for creating derived objects. If the basetype is a model, all of its descendants are derived. Moreover, if the target model is a type itself, the objects derived from the target will also contain a derived copy of the newly derived objects.

Relations are preserved in the derived types unless the project operations mask disables this feature (this is described in detail at 3.3). Internal relations will be redirected to the derived child of the target. Relations targeting external objects refer to the same object as the corresponding relation in the basetype.

Accessing the (base)type hierarchy.

DerivedFrom: the object from which the current object is immediately derived. NULL for non-derived objects.
Type: the type object from which the current instance object is derived (possibly through other instances). Error is returned if the object is not an instance.
BaseType: the object from which the current type object is immediately derived. NULL for non-derived objects, error returned for instances.
ArcheType: the object that is at the farthest position within the chain of base objects (i.e. the one which is not derived from anything). NULL if the object is not derived.
IsInstance: returns VARIANT_TRUE, or VARIANT_FALSE. It also returns VARIANT_FALSE if the object is not derived at all.
IsPrimaryDerived: returns VARIANT_TRUE if the immediate base object is a type (i.e. root FCO), or VARIANT_FALSE if it is a descendant within a root FCO type, or if the object is not derived.

Getting the subtypes of an object

DerivedObjects: the list of objects (types, instances) derived from this object.

Changing the inheritance hierarchy.

AttachToType: set an existing object to be derived from a type
DetachFromType: break the inheritance link between an object and its base.

Comparing a relation object to its basetype

CompareToBase: Determine if the relations of the derived object follow the relations of the base object.
RevertToBase: Modify the derived object to exactly follow the relations in its base object.

Finding the equivalent object in the subtype/instance

ChildDerivedFrom: Find the child of a model that is immediately derived from the argument FCO. The parent of this argument FCO is the immediate base of the model on which this method is invoked.

3.2 Move and Copy

MoveFCOs and CopyFCOs operations (available for MgaModel and MgaFolder) can affect a large number of objects in one call, especially if they involve moving from or moving/copying to types (described later). Both operations are passed a collection of objects to be moved/copied. These objects do not need to be siblings originally, but their common new parent will be the object (model or folder) whose MoveFCOs or CopyFCOs method was called.

If an object in the collection is a model, all of its descendants are copied or moved.

If the new parent is a model, the roles of the objects in the new parent must be determined. This happens either manually or automatically. The roles parameter accepts an IMgaMetaRoles collection, which must either be null, or must have a length exactly matching that of the objects collection. The role specified at the N-th position will determine the role of the N-th object. The roles collection may contain NULL's, which implies automatic role assignment. If the entire roles parameter is NULL, all roles are assigned automatically. A role assigned automatically is selected according to the original role of the object. Matching is based on rolenames, and if the new container does not have a role that corresponds (by name) to the original one (or the object was a rootobject and has no role at all), the entire operation fails with an E_MGA_NO_ROLE error.

3.3 Relations involved in complex operations

Objects copied, moved, derived, or deleted may represent relations (references, sets and connections) to other objects, or may be related by others. Depending on the relative position of the relation and its target, three relevant cases are identifiable:

MM_INTERNAL:	Relation between parts of the original copied/moved/deleted/ derived objects, i.e. relations where both the relation and its target are among the affected objects.
MM_OUTWARD:	The relation is among the affected objects, but the related object is somewhere outside.
MM_INWARD:	The relation not part of the affected objects, but the target is one of them (this later case is only relevant for delete and move operations).

(Note, that **MM_*xxx*** mnemonics are all defined in Mga.idl.)

The action to take when doing the operations on relations is determined by the MgaProject.OperationsMask() property. This is 32-bit bitmask, where an octet is assigned to each of the relations

MM_REF:	references (octet 0, i.e. the least significant byte/LSB),
MM_CONN:	connections (octet 1) and
MM_SET:	sets (octet 2).

Within each octet, separate groups of 2 bits define instructions for the following relations:

MM_INTERNAL	(bit 0-1, mask value 0x3),
MM_OUTWARD	(bits 2-3, mask value 0xC) and
MM_INWARD	(bit 4-5, mask value 0x30)

The bit value defines the instructions to follow:

00 / MM_ERR:	Signal error and abort the entire operation.
01 / MM_CLEAR:	Clear the relation (set the reference to NULL, delete the connection point, or remove the set member) and proceed with the operation. If the **MM_FULLDELETE** (0x40) bit is also set in the corresponding octet, destroy the entire connection object as well.
10 / MM_DO:	Transform the relation to be a logical equivalent of the original. (For delete operations, MM_DO and MM_CLEAR have the same effect.)

By setting the different operationsmask values, the relation-preserving behavior of the complex operations can be controlled in fine detail.

3.4 Attributes

First class objects (MgaFCO's) can carry a set of attributes. The types and number of these attributes are defined by the meta. The supported attribute types are as follows:

ATTVAL_STRING: a string of max. 256 bytes length
ATTVAL_INTEGER: 32-bit signed integer
ATTVAL_DOUBLE: 64-bit float number
ATTVAL_BOOLEAN: one of the values true or false.
ATTVAL_REFERENCE: reference to another FCO
ATTVAL_ENUM: a string value corresponding to one of the possible choices defined in the meta. The input method for such attributes is usually a menu of fixed choices.
ATTVAL_DYNAMIC: a string value corresponding to one of the possible choices defined in the object itself. The input method for such attributes is usually a menu of user-defined choices.

Attributes implement a *defaulting mechanism*. If an object has an unset attribute, it will assume the value defined in the object from which it is derived. If that object does not have a value for this attribute defined, the inheritance chain is traversed along its length. If none of the predecessors has the corresponding value set, the system resorts to the default defined by the meta of the attribute. Consequently, the value exposed by an attribute effectively reflects the corresponding attribute value of the closest object where it is defined. If a value is assigned, however, it never modifies any of the (base)types, but only the object itself. The location of the actual value of an attribute is indicated by the *attribute status* (described below).

Attributes may need to contain variable information in addition to their value. A good example for this is a dynamic choice attribute, where the choices must also be recorded within the attribute. Compared to the attributes themselves, this information is less strictly structured, and its interpretation is much more application-specific. The *attribute registry* is a data structure designated for these types of information. The registry can be accessed through the MgaAttribute object. See the registry section below for further instruction on using it.

There are several techniques to work with attributes, which differ in the way of identifying the attribute and in the data type used for accessing its value:

Direct name-based access
This is the simpler way to read and write attributes. The attribute (identified by its name) is accessed as a property of the owner object in a single command.

- The value may be transferred as a Variant through **AttributeByName**, or through one of the ***xxx*AttrByName** properties as String, Integer, Double, Bool, or an IMgaFCO reference to an object <u>*in the same project*</u>. For setting the attribute through a Variant, normal variant conversions are provided, but with the other methods the property used must match the type of the attribute. Enumeration and dynamic attributes are also to be transferred as strings.
- Name-based access can also be used to Clear an attribute (**ClearAttrByName)**. This of course may expose a default value as described in the defaulting mechanism.

Access through an MgaAttribute object
This is a more complicated process of at least two-steps. First an MgaAttribute object is retrieved, identified by its MgaMetaAttribute (which is possibly obtained from the attribute name through another call to MgaMetaFCO), and then properties and methods of this subobject are used. The most notable advantage to this technique is that it provides access to the status and registry of the attribute.

- **Value, *xxx*Value**, and **Clear** are the equivalent of the direct name-based access properties and methods described above. The same rules of using value types apply here as well.
- The status of an attribute provides information on where exactly the current value of an attribute is defined. The **Status** property returns one of these codes:
  - ATTSTATUS_HERE: the value is defined the current object.
  - status is > 0: the number is the distance between the current (sub)object and the (sub)object that provided the current value measured along the inheritance chain.
  - ATTSTATUS_METADEFAULT: the value is defined by the meta.
  - ATTSTATUS_UNDEFINED: neither the object, nor its (base)types or the meta contains a value.
  - ATTSTATUS_INVALID: database error condition; normally never happens.
- The per-attribute registry can be accessed through the **Registryxxx** functions. See the description of the registry (3.5) for details.

Access through the Attributes collection: the **Attributes** FCO property returns an MgaAttributes collection of all the attributes defined for an object (i.e. also including the ones without assigned values in the current object). The MgaAttribute objects retrieved from the collection (using its Item() property or GetAll() method) are used as described above.

3.5 The Registry

Attributes are useful for storing strongly typed information on objects. While it is usually a good place to record most relevant pieces of information, the same fixed structure often becomes impractical if other, less strictly structured information needs to be associated with the object. Visualization data is a typical example for this, especially if part of this information is considered to be relevant for a single application only. Another example could be the case when the amount of information to be stored by an object is potentially infinite or cannot be determined during the metaprogramming phase.

The registry is a recursive, tree-like data structure of theoretically unlimited size and depth. The basic element of this structure is a *node*, which may have a string *value* assigned to it and may contain any number of *subnodes*. Nodes also have a string type *key* that uniquely identifies them within their immediate parent node. Keys are unique only in their parent's subnode collection, so, to unambiguously identify a node in the registry, a full ('/'-separated) enumeration of all the keys of its parents is prefixed to the key of the node itself. (This is very similar to the identification mechanisms used in file systems, directory systems, and the Windows registry.)

The MgaLib defines several interfaces that can own registries: FCO's, folders, and also subobjects of FCO's represented by MgaPart and MgaAttribute. All these registries work identically, although subobject registries exist only *virtually*. This means that they are physically contained within the registry of their parent FCO. The only noticeable implication is that subobject registries are also locatable as part of the parent's registry. (Specifically, part and attribute registries are found under the **/_PartRegs_/<*aspectname*>/** and **/_AttrRegs_/<*attributename*>/** subnodes in the parent object.)

Registries implement a defaulting mechanism similar to the one seen with attributes. Values defined by the (base)types of an object are visible in the object itself unless they have been overridden. MgaMetaFCOs and MgaMetaFolders may also provide default values which may be overridden in both the object itself and in any of its (base)types.

The visibility of nodes is conceptually defined by 'merging' the object's registry tree with the registries of its (base)types and of its metaobject, using the following rules of conflict resolution:

- The effective value of a node is the closest value-bearing node with the corresponding pathname in the chain of 1) the object itself 2) the predecessor object(s), 3) the metaobject. Nodes may exist without a value (e.g. after Clear()); these objects expose the settings in their (base)type or meta.

- Nodes do not automatically mask out the subnodes of their equivalents in the predecessor objects. The set of subnodes is thus effectively extended by the subnodes of the predecessor nodes.
- Setting the node to be *opaque* can change the behavior described above. In this case, neither the node's value nor the set of its subnodes will be affected by the corresponding nodes in the predecessors. In this way, a single opaque node can effectively mask out the corresponding subtree of nodes in all the predecessor objects.

There are two ways to work with registry nodes:

Access through registry owner objects:
MgaFCO, MgaFolder, MgaAttribute and MgaPart all define properties to access (read and set) registry node values through absolute pathstrings. This limited functionality is sufficient for most applications.

Access through MgaRegNode objects:
Registry owner interfaces also provide access to the collection of the locally defined first-level nodes (**Registry**), or any node identified by absolute pathstrings (**RegistryNode**). It is important to note that, while **RegistryNode** always returns an **MgaRegNode** object, the collection returned by the **Registry** property returns only the locally defined registry nodes, but not the inherited ones.
The resulting **MgaRegNode** object provides full access to the registry:

- Setting, reading and clearing node values: the **Value** property is used to set/retrieve the value. Registry values are always of string type. The **Clear** method removes the value in the current node, if there is any. Clear, in effect, reverts the node's value to that of the corresponding node in the nearest (base)type or metaobject. Neither of these operations have any effect on the subnodes.
- Getting the status of a node. The **Status** property exactly where the value of a registry node is defined:
    - ATTSTATUS_HERE: the value is defined in the current (sub)object.
    - status is > 0: the number is the distance between the current (sub)object and the (sub)object that provided the current value measured along the inheritance chain.
    - ATTSTATUS_METADEFAULT: the value is defined by the meta. (The metaregistry is always expected to define a (possibly empty) value, so this status never returns ATTSTATUS_UNDEFINED.)
    - ATTSTATUS_INVALID: database error condition; normally never happens.
- Navigation among their subnodes: the **SubNodes** and **SubNodeByName** can both be used to access subnodes of the current node. As with the (sub)object methods, while **SubNodeByName** always returns a **MgaRegNode** object, the collection returned by the **SubNodes** property only returns the locally defined registry nodes, but not the inherited ones.
- Accessing the opacity attributes through the **Opacity** property.
- Removing a node along with the subtree supported by it. Once a node is created in an object's registry, it remains there no matter what value or how many subnodes it has.

Thus, **RemoveTree** is the only method to remove nodes. Of course, RemoveTree recursively removes all the child nodes as well.

3.6 Searching the object tree using filters.

Instead of providing a number of functions for searching the object hierarchy by different criteria, MgaLib offers only a small number of functions to execute searches based on a configurable search filter. This filter is MgaFilter, and the methods to use it are MgaProject.AllFCOs, MgaFolder.GetDescendantFCOs, and MgaModel.GetDescendantFCOs. As the name implies, these methods search parts of the object tree, and return a list of objects that match the conditions set in the filter.

MgaFilter objects are created through MgaModel.CreateFilter. Before being used in the search methods, a filter is usually initialized by setting any of its criteria properties:

- **Name**: a single name or a space-separated list of names. An object will match if its name equals to any of the names in the list.
- **Kind**: a space-separated list of kindnames of metaref numbers. An object will match if its kindname equals to any of the names in the list, or if its kind metaref matches any of the metarefs on the list.
- **Role**: a space-separated list of rolenames of metaref numbers. An object will match if its rolename equals to any of the names in the list, or if its role metaref matches any of the metarefs on the list. Root objects do not have roles, so they never match a non-empty role field.
- **ObjType**: a space-separated list of objtype numbers or objtype mnemonics as defined by MGA.IDL, like "OBJTYPE_MODEL" or "3" (the latter is equivalent to "OBJTYPE_REFERENCE"). An object matches if its objtype or mnemonic is included in the list.
- **Level**: a space-separated list of numbers or dash-separated number pairs. For example '-2 8 11-14 23-' will match all objects which are at level 0, 1, 2, 8, 11, 12, 13, 14, 23 or any number above. In case of the GetDescendantFCOs methods, 'Level' is defined as relative distance measured from the object itself (0: the object itself, 1: its children, etc.). For MgaProject.AllFCOs, the folder hierarchy is invisible and 'Level' is the distance measured from the RootFCO level, that is 0 specifies the RootFCOs, 1 specifies their children, etc. The maximum number allowed in a Level specification is 30, all deeper levels are included if and only if the string contains an open-ended specification (e.g "28-").

If any of the specifications is the empty string, the corresponding test is skipped. As an extreme, an MgaFilter without any initialization will return all of the candidate objects.

As it is the case with other return values, the resulting collection reflects the state of the database and the filter at the time when the search was executed. Later modifications on the filter or the data do not effect the selected set.

4 Multiclient operation and event notifications.

4.1 Territories

An MgaProject object may have several simultaneous clients (even several clients on the same machine). To let them all have a consistent view of the state of the database, there is a notification mechanism built into the library.

The design goals for this mechanism were efficiency and simplicity -- two rather contradictory requirements. Efficiency requires that clients receive notification only for those changes that are relevant for them (and also receive detailed context information at the same time), while a single general notification ('something has changed') and the subsequent full refresh by all the clients is probably by far the simplest solution.

As a core concept in the MGA library, each GUI, Interpereter or other MGA client maintains a so-called 'Territory of interest' ('Territory' for short) object. The only possible MGA clients without Territories are event-based add-ons (described below), which run on event notifications and are executed in a new Territory context every time.

The Territory is a semi-automatic set of FCO or Folder objects: objects can be added with the OpenObj, OpenFCO, and OpenFCOs methods of the MgaTerritory interface, but any object reference returned by a library function call automatically opens the object in the territory. (In practice, OpenObj is only used when interface pointers are passed between territories).

Objects are discarded from the territory

- if the last reference to an object is released.
- by the MgaTerritory.Flush and MgaTerritory.Destroy operations or the COM destruction of the MgaTerritory object, which immediately discard everything.

Note that the Territory contains MgaObjects (FCO’s and folders) only, i.e. changes to subobjects generate notifications for their parent objects. Non-object interfaces do not have notification (apart from global events associated with changes of the project object).

All access to the database is executed in a Territory context:

- All client access is opened by a BeginTransaction call, which receives a territory as a parameter.
- Before notifications are delivered to each territory, the receiving territory is made active. Although the notified client is not working in its own transaction, all database access will affect its own territory only.
- Add-ons are executed in an empty territory every time. This territory is flushed after the add-on is done.

Besides receiving notifications, the presence of an object in a territory also involves a lock acquired on the object that prevents other projects (projects running on other machines) from modifying the object in the database. The lock automatically acquired is a read lock, which is overridable by the Lock() and UnLock() methods of the object interface.

4.2 <u>Add-ons</u>

Add-ons (event-based add-ons) are client modules that do not initiate transactions but only execute on notifications. Consequently add-ons do not have territories, but they receive notification on all object change events for which they have their event mask set. While most clients (the ones that do not use read-write notification) need read-only access when responding to events (since they usually modify the database within their own transactions), add-ons are expected to actively change the database when notified. Since these changes usually provide some higher-level consistency for the database (e.g. keep two unrelated object attributes in synch), it is reasonable that they belong to the original transaction.

Add-ons are created by the MgaProject.CreateAddOn method and remain active until their Destroy() method is called or their last COM reference is released.

4.3 <u>Event handlers</u>

When a territory or add-on is created (MgaProject.CreateTerritory, MgaProject.CreateTerritory), an event handler is specified as a parameter. These are callback objects (implemented by the client) that support the IMgaEventHandler interface. The event handler will receive notification on all global events, and on per-object events if any of the objects in the territory has changed. The IMgaTerritory and IMgaAddOn interfaces allow a mask to be specified (EventMask()) to exclude events irrelevant for the client.

Territories allow an optional second event handler (rwhandler) to be specified. This works similarly to the primary one, but executes in read-write mode. Add-on event handlers are always executed in read-write mode.

4.4 <u>Events an event delivery</u>

There are two types of notifications: global events referring to global changes, like undo/redo and transaction operations, and per-object events to signal that an object or any of its sub-objects have changed. If there is a change to a relation between two objects (e.g. an endpoint is set for a connection), both objects (i.e. the connection and its target) receive notifications.

Object events are much more frequent, since global notifications mostly reflect less common, somewhat exceptional events: transaction abort, redo, undo, project property changes, etc. (Two notable exceptions, GLOBALEVENT_COMMIT_TRANSACTION and GLOBALEVENT_NOTIFICATION_READY, are mentioned later). Global notifications are immediately delivered whenever such an event occurs.

Object notifications are not delivered immediately, but in batches, when the transaction owner decides to do so (invoking the MgaProject.Notify() method) or commits the transaction (MgaProject.CommitTransaction()). Notifications are then delivered on all changed and still unnotified objects to all the territories that contain the object and indicate interest in at least one of the changes that occurred.

After all object notifications are delivered, each notified territory and add-on receives a GLOBALEVENT_NOTIFICATION_READY global notification and also another GLOBALEVENT_COMMIT_TRANSACTION notification if the transaction is committed. These signal to the client that there are no more pending notifications.

Global notifications arrive by a callback to the GlobalEvent function. Global notifications are to be observed, since some project-level modifications (AbortTransaction, Undo/Redo) do not send per-object notifications for the objects affected, but instead generate a single global event only. (As a general rule, clients are expected to perform a complete database refresh in these rather exceptional cases.)

Object notifications are received through a call to ObjectEvent. The parameters to this function are the Object pointer, the event mask (multiple bits may be set), and an additional user-defined value. The latter serves as a convenience to the client: before the event, the client can Associate() a single VARIANT (e.g. a pointer to one of his data structures) with any object in its territory, and this value (returned during notification) will be used in the client event handling routine to locate the appropriate client data. This VARIANT value is per-object and per-territory, so others cannot interfere with it. If no data has been set, (or it has been reset to NULL) the notification will contain the NULL VARIANT in this parameter.

4.5 The sequence of multi-client operations

Most clients will access the MGA database in two situations:

- As a *standalone* operation initiated by the client itself. Such operations must be bracketed by BeginTransaction/CommitTransaction calls.
- In response to an *event notification*. This is also the only access mode for event-based add-ons.

The typical sequence of steps in a transaction are as follows.

1. Apart from creating Territories, Add-Ons and setting the MgaProject.OperationsMask, clients are not allowed to access the database without beginning a transaction first, even if another client has opened a transaction. Such operations result in either a E_MGA_NOT_IN_TRANSACTION, or a E_MGA_FOREIGN_TRANSACTION error.
   Accordingly, standalone operations by the clients are fully synchronized. Such an operation begins when a client calls BeginTransaction with its own territory, and ends when it calls CommitTransaction or AbortTransaction. If another client tries to begin a transaction, it will be refused until the first concludes (E_MGA_ALREADY_IN_TRANSACTION).
2. If the database is changed during an operation, events will be generated at some later time. By default, it will happen during the execution of the IMgaProject::CommitTransaction call, but the client can choose to deliver all pending events at any time by calling IMgaProject::Notify().
3. When events are sent out, all interested clients receive them, one after the other.
4. First, the event-based add-on clients and territory read-write handlers are executed. These are allowed to both read from and write to the database. If an object is changed, this change is again delivered to all of the interested add-on and read-write handler clients (care must be taken when writing modifying handlers to avoid infinite loops). When there are no more notifications to be delivered, handlers are delivered globalevent messages (GLOBALEVENT_NOTIFICATION_READY). This may again cause changes and per-object notifications. The process repeats itself until a quiescent state is reached.
5. The last handlers to be notified are the traditional territory-based read-only handlers, i.e. those having their own territories. They will receive notification on both the original changes and the ones by the handlers described at 3). These clients are not allowed to change the database while handling events. The pure purpose of these notifications is to refresh client copies of object data and to visually reflect changes in user interfaces. If a client wants to respond to events by writing to the database, it should either implement a read-write handler, or schedule a client-initiated operation for itself at the next available time. When there are no more events to be delivered, handlers receive a global notification (GLOBALEVENT_NOTIFICATION_READY).
6. If the notification was the result of a transaction commit, changes are committed to the database, and the MgaProject returns to transaction-free state. Any client is now free to initiate a new transaction.

Note: Since transactions are to be executed in a synchronized way, they are expected to be short, like handling a message in a windowing system. If client is expected to do a long series of calls, it should either break them into several transactions, or (at least) issue MgaProject.Notify() calls regularly to enable refresh by the other clients.

4.6 <u>Operation of multiple hosts working on the same database.</u>

The previous section describes the rules when multiple clients use the same MgaProject object simultaneously. This is a very common situation, since the GME GUI alone uses the MgaLib through several independent client threads. Interpreters and other GUI objects further increase this parallelism.

Another form of shared access is established when multiple MgaProject objects share access to a single SQL-server based repository database. Since these projects usually run on different machines, synchronization does not rely on notifications but on database locks instead. Locking is automatic, i.e. database integrity is guaranteed no matter what the user does. On the other hand, it may occur any time that the MgaLib library refuses to execute an operation because another client locks the requested part of the database. The error code for this situation is E_LOCK_VIOLATION (0x80732001). In case of a locking error, the client is unable to proceed until the other client on the other machine releases the lock.

Multiple clients are allowed to access any part of the database in read-only mode. To change any part, however, exclusive access is required. In practice it means that if several hosts work on the same MGA database, they can modify only disjoint, and unconnected (no inheritance, relations, etc.) parts of the database. Other clients are not allowed to read these parts. For closer cooperation, clients must synchronize their operations by releasing locks for each other.

Locks can be released by flushing all territories that have accessed the locked object and executing the MgaProject.FlushUndoQueue() function. Locks on objects are also released if there are no territories holding the object and the transaction of the last access shifts out from the undo queue.

References

[1] A.Ledeczi, et al., **Metaprogrammable Toolkit for Model-Integrated Computing**
Proceedings of the IEEE ECBS'99 Conference, 1999.
[2] **GME2000 Users Manual, Version 1.0**
ISIS, Vanderbilt University, October 2000