An Object-Oriented Approach to the Implementation of OSI Management

Luca Deri, Eugenio Mattei

Despite the advantages offered by the generality of its model, the effort put into its definition by the standardization bodies, and the support of government organizations, OSI management is still far from reaching a predominant stand in the market of network and systems management. Proprietary architectures, products and, especially in the United States, SNMP are still the preferred solutions for many users. One of the main obstacles to the wide adoption of OSI management is the supposed difficulty of its implementation.
This article will attempt to show that such complexity can be resolved if the proper tools are chosen and if the intrinsic object-oriented features of OSI management are exploited. The design and implementation of an OSI management library is described. It will be shown that a library implementation is feasible and can suitably exploit the object-oriented structure of the management information: the definition of automatic tools for the implementation of new managed object classes is also covered. Finally, implications related to the handling of extension of the managed object class behaviour are identified.
Familiarity with OSI management, object-oriented terminology and C++, though not strictly required, is certainly useful.

Keywords: Network Management, OSI Management, GDMO, ASN.1, C++

Introduction

OSI Management [1] encompasses the definition and implementation of management tools that use the OSI application layer protocols for communication. Base components of OSI management are: the *Management Model*, the *Information Model*, communication protocols used for the actual transfer of management information, and a number of generic resource-independent functionalities collectively called *Systems Management Functions*. The Management Model [5] is defined in terms of management applications that perform management activities in a distributed manner by establishing associations between system management entities (agents and managers). The manager system manages the network resources, according to a defined policy, by issuing remote management requests to one or more agent processes. The agent process manages the real resources by executing the requests issued by the manager.

The OSI Information Model [6, 7] structures the *management information* according to a description of the resources to be managed in the network. Examples of such resources may be a host, a routing table, a network device and an application process. The information model deals with *managed objects* that are abstractions of

real resources for the purpose of management. They embody the management information for management applications. The notation used to describe them is defined in a document entitled “Guidelines for the Definition of Managed Objects (GDMO)” [8]. ISO has defined the GDMO language to provide a common way to define the managed information.

The effective transfer of the management information between agent and manager processes is performed using the CMIP (Common Management Information Protocol) protocol [3, 4].

This article focuses on the design and implementation of an OSI general-purpose management library. It describes a true library implementation that may be used as the kernel of an OSI management application. The guidelines and examples have been drawn from implementation experience of the authors in the context of European research projects and in the course of designing and implementing commercial products.

The library implementation contains a set of functions to be linked to the user modules to obtain the final application. This solution has been mainly chosen because it does not contain any system dependent section and therefore it allows great portability.

1. OSI Management and Object-Oriented Technology: An Evolutionary Approach

Much of the complexity of the OSI management stems from the generality of its model. The complexity arises from the amount of validity checking that an implementation must perform, such as checking if an attribute is member of a particular managed object class, verifying if a particular instance can be created, controlling if the specified value is within the specified range. In order to limit this complexity, an important objective is to define a framework and a set of tools that perform all these oprations in a general and efficient way, therefore allowing the implementor to focus on the part of the implementation that does the “real” work. This goal may be achieved by suitably exploiting the intrinsic object-oriented structure of the OSI management information.

This section shows that the GDMO constructs can be represented by an object-oriented language. The C++ language has been chosen since it adds powerful object-oriented extensions to C, which is by far the most widely used language in network applications.

In GDMO the key construct used to define the structure and behaviour of the managed object classes (MOCs) is the MOC template. This template identifies the inheritance relationship, the contained packages behaviour, the attributes, notifications and operations allowed in the MOC. A GDMO class template is the base of the formal definition of a managed object. It is defined in the following way:

```
<class-label> MANAGED OBJECT CLASS
    [DERIVED FROM <class-label> [,<class-label>]*;]
    [CHARACTERIZED BY <package-label> [,<package-label>]*;]
    [CONDITIONAL PACKAGES <package-label> PRESENT IF condition-definition
                          [, <package-label> PRESENT IF condition-definition]*;]
REGISTERED AS object-identifier;
```

The above management scheme clearly encompasses object-oriented concepts and features. In fact a MOC may be considered as a category of managed objects. The definition of a class is derived from that of another class. The specific peculiarities of the class that characterizes it with respect to its superclass are defined by means of the packages specified after the `CHARACTERIZED BY` construct. The `DERIVED FROM` construct defines that this MOC requires all the characteristics of the superclass(es) and the `CHARACTERIZED BY` construct specializes it by adding new characteristics. The characteristics of the properties of a MOC are named its *attributes*, and each attribute has a *value*.

In order to show how this can be expressed in C++, suppose we have the following GDMO class:

```
logRecord MANAGED OBJECT CLASS
    DERIVED FROM top;
    CHARACTERIZED BY ...
    ...
REGISTERED AS {2, 9, 3, 2, 3, 7};
```

This MOC may be defined in C++ in the following way:

```
typedef char* ObjectId;

class top {
private:
    ObjectId      registeredAs;
    NameBinding   nameBinding;
... };
```

```
class logRecord : public top {
private:
  ...
public:
  logRecord() {
    objectIdCopy(&registeredAs,
    "logRecord"); ...};
... };
```

Please note that the symbolic value of the object identifier, `logRecord` in this case, is stored, rather than a sequence of integers: this is mapped internally into the real value `2.9.3.2.3.7`, following the same approach used in Isode [13] with the `oidtable` files.

With this C++ definition, it is possible to exploit the inheritance between the MOCs thus avoiding the need to redefine the same attribute, `registeredAs`: for each class it is defined once and instantiated according to the GDMO definition of the derived class.

The package template is a combination of behaviour definitions, attributes, attribute groups, operations, notifications and parameters. It may be referenced in a MOC template and is defined in the following way:

```
<package-label> PACKAGE
    [BEHAVIOUR         <behaviour-definition-label> [,<behaviour-definition-label>]*;]
    [ATTRIBUTES        <attribute-label> propertyList [<parameter-label>]*
                       [, <attribute-label> propertyList [<parameter-label>]*]*;]
    [ATTRIBUTE GROUPS  <group-label> [<attribute-label>]*
                       [, <group-label> [<attribute-label>]*]*;]
    [ACTIONS           <actions-label> [<parameter-label>]*
                       [, <actions-label> [<parameter-label>]*]*;]
    [NOTIFICATIONS     <notification-label> [<parameter-label>]*
                       [, <notification-label> [<parameter-label>]*]* ;]
REGISTERED AS object-identifier;
```

where

```
propertyList      ->     [REPLACE-WITH-DEFAULT]
                         [DEFAULT VALUE value-specifier]
                         [INITIAL VALUE value-specifier]
                         [PERMITTED VALUES type-reference]
                         [REQUIRED VALUES type-reference]
                         [get-replace]
                         [add-remove]

value-specifier   ->     value-reference DERIVATION RULE <behaviour-derivation-label>
get-replace->     GET | REPLACE | GET-REPLACE
add-remove        ->     ADD | REMOVE  | ADD-REMOVE
```

The package template can also be modelled using object-oriented design. A package may be defined as a C++ class that contains instances of attributes, attribute groups, actions and notifications. Suppose we have defined the following GDMO package:

```
examplePackage PACKAGE
    BEHAVIOUR            exampleClassBehaviour;
    ATTRIBUTES           objectName    GET,
                         Error-Counter PERMITTED VALUES AttributeModule.CounterRange
                                       REQUIRED  VALUES AttributeModule.CounterRange
                                       GET;
    ATTRIBUTE GROUPS     attributeGroup;
    NOTIFICATIONS        protocolError;
REGISTERED AS {joint-iso-ccitt ms(9) smi(3) part4(4) package(4) examplepackage(1)}
```

This package may be defined in C++ in the following way:

```
class examplePackage : public Package {
private:
    ObjectNameAttribute         objectNameAttribute;
    ErrorCounterAttribute       errorCounterAttribute;
    AttributeGroup              attributeGroup;
    ProtocolErrorNotification   protocolErrorNotification;
public:
    examplePackage() { objectIdCopy(&registeredAs, "examplePackage"); ...};
... };
```

where

```
class Package {
private:
    ObjectId      registeredAs;
... };

class ObjectNameAttribute :public Attribute { ...
class ErrorCounterAttribute: public Attribute { ...
class AttributeGroup: public AttributeGroup { ...
class ProtocolErrorNotification: public Notification { ...
```

and the detailed definition of the Attribute and Notification classes is the one given in 2.1.

In the OSI management environment, the semantics (behaviour definition) of the various components of a MOC is distinguished from the corresponding format (syntax). The syntax is defined by the Abstract Syntax Notation One (ASN.1) [2], which provides a wide variety of types ranging from simple bit strings to complex structures. The GDMO formalism defines a set of templates to link the semantics of the various constructs with the syntax of their values: the most commonly used are the WITH ATTRIBUTE SYNTAX in the ATTRIBUTE template, WITH INFORMATION SYNTAX and WITH REPLY SYNTAX supporting productions in the ACTION and NOTIFICATION templates. The different options for syntax management are described in 2.2. Another way to establish this link is by means of the PARAMETER template. It is defined in the following way:

```
<parameter-label> PARAMETER
    CONTEXT       context-type;
    syntax-or-attribute-choice;
    [BEHAVIOUR <behaviour-definition-label> [, <behaviour-definition-label>]*;]
REGISTERED AS object-identifier;
```

where

```
context-type                  ->    context-keyword | ACTION-INFO | ACTION-REPLY |
                                    EVENT-INFO | EVENT-REPLY | SPECIFIC-ERROR
context-keyword               ->    type-reference.<identifier>
syntax-or-attribute-choice    ->    WITH SYNTAX type-reference |
                                    ATTRIBUTE <attribute-label>
```

Parameters qualify and further define the structures in the syntax of attributes, action requests / responses and notifications; therefore they are most commonly used

to associate user responses with actions, operations on attributes, create and delete. A parameter template not only joins the behaviour with the syntax definition, but also identifies the context where the parameter may be present in a Protocol Data Unit (PDU) as expressed by the `context-type`production. This feature may be exploited not only for checking purposes, but also in the encoding/decoding phase to restrict the contexts where the parameter may be present.

The adoption of an object-oriented approach to model the OSI management information reduces the design of a MOC to an “ad hoc” composition of basic modules. In this way it is also possible to reuse modules as the attributes in different classes, thus reducing the amount of code to be written for each MOC but also making the integration of different modules easier. In fact defining for each basic piece an API, which in C++ is a set of attributes and methods, allows modules to be integrated from various implementations and automatic tools to be produced for their generation and integration.

2. The management framework: library structure

This section shows how to apply the concepts introduced in the previous one. An implementation of the OSI management library has the following objectives:

- to develop a general-purpose library;
- to limit the size of the code needed to add new MOCs by sharing the code among MOCs;
- to define tools that automatically generate the code needed to manage additional MOCs.

2.1 Definition of managed objects

Each MOC may be seen as the integration of the following basic components: packages, name bindings, behaviour characteristics. Distinct MOCs are derived from a common superclass, and they are different with regard to the number and type of their basic components.

```
class ManagedObjectClass {
    protected:
        char* ClassName;                              /* Class name */
        ObjectId ClassId;                             /* Class identifier */
        int NumberOfClassPackages;
        Package* theListOfClassPackages;            /* Package list */
        int NumberOfNameBindings;
        BindingRecord* theListOfClassNameBindings;  /* Name Binding list */
        Allomorphism allomorphs;                    /* List of allomorphic classes */
      ........ };
```

This C++ class models the structure of a MOC. The packages of the class, derived from the `Package` C++ class, are stored in the `theListOfClassPackages` list. The various name bindings, defined using the `BindingRecord` C++ class, are stored in the `theListOfNameBindings` list. The `BindingRecord` class, which contains all the information related to the name binding, may be defined in the following way:

```
class BindingRecord {
  private:
   ObjectIdSuperiorClassId;               /* ObjId of the superior class */
   ObjectIdBindingClassAttributeId;   /* Class binding attrib. identifier */
   Bool           allowManagementCreation; /* True if creation is allowed through
                                              management operations */
   CreateModifier  createModifier;          /* Flags: reference and automatic creat. */
   Bool           allowManagementDeletion; /* True if the deletion is allowed through
                                              management operations */
   DeleteModifier  deleteModifier;          /* Flags: delete-if-no-contained or
                                               delete-contained */
....};
```

The correspondence between these C++ classes and the pertinent GDMO templates is clear.

The `Package` class may be defined in the following way:

```
class Package {
         private:
         ObjectId PackageId;                          /* Package identifier */
         Bool IsAPresentPackage;                      /* True if the pkg. is present */
         int NumberOfAttributes;
         Attribute* theListOfAttributes;              /* Attribute list */
         int NumberOfGroupAttributes;
         AttributeGroup* theListOfGroupAttributes;    /* Attribute Groups list */
         int NumberOfClassNotifications;
         Notification* theListOfClassNotifications;   /* Notification list */
         int NumberOfClassActions;
         Action* theListOfClassActions;               /* Action list */
       ........ };
```

The attributes derived from the `Attribute` C++ class are stored in the `theListOfClassAttributes` list. The notifications derived from the `Notification` C++ class are stored in the `theListOfClassNotifications` list. The actions derived from the `Action` C++ class are stored in the `theListOfClassActions` list.

The Notification class may be defined in the following way:

```
typedef int Error;

class Notification {
    private:
        ObjectId NotificationId;  /* Notification identifier */
    public:
        Error BuildNotification(EventReportArgument*, ManagedObjectClass*);
                                  /* Builds the notification filling the
                                     EventReportArgument for given input MO; an
                                     error code is returned */
      ........ };
```

The BuildNotification method builds the notification for the given ManagedObjectClass instance, filling the EventReportArgument parameter. This method is used in the implementation of the Event Forward Discriminators and Log MOCs [7] as described below.

The Action class may be defined in the following way:

```
class Action {
    private:
        ObjectId ActionId;        /* Action identifier */
    public:
        Bool TryToExecuteAction(ActionInfo*, ManagedObjectClass*);
                                  /* This method returns True if the action can be
                                     executed with success on the input MO, False
                                     otherwise. The ActionInfo parameter, if not
                                     applicable to the action, is set to NIL */
        Error ExecuteAction(ActionInfo*, ActionReply*, ManagedObjectClass*);
                                  /* Executes the action on the input MO and fills the
                                     ActionReply; an error code is returned. The
                                     ActionInfo and ActionReply parameters, if not
                                     applicable to the action, are set to NIL */
      ........ };
```

In the previous class the two methods TryToExecuteAction and ExecuteAction are defined to execute the action. The first method is used in the case of an action request with the synchronization set to atomic: for each filtered MO instance, the library checks whether it is possible to execute the action invoking the TryToExecuteAction method on the selected MO instance. TheExecuteAction method executes the defined action.

The framework described above offers the following advantages:

- the management information is modelled as in GDMO: this approach, aside from its intrinsic coherence, provides extendibility and will allow the implementation of forthcoming standard functionalities such as management knowledge functions;
- each class is modelled by exploiting the Attribute class (used indirectly in the Notification and Action classes): this allows run-time instantiation of the

MOCs as explained below.

The `Attribute` class may be defined in the following way:

```
class Attribute {
private:
     ObjectId AttributeId;       /* Attribute identifier */
     char* theAttributeValue;/* Parsed attribute value */

   public:
     virtual void   Free();
     virtual char*  Copy(Error, Bool);        /* Return a copy of the attr. value */
     virtual Result Compare(void*, Bool);     /* Compares the input value with the
                                                 attribute value */
     virtual char*  Get(Bool);                /* Return the attribute value */
     virtual void*  GetResourceValue(Error*); /* Get the value of the associated
                                                 resource */
     virtual Result CheckSet(AttrModifier*); /* Check if the attribute value may be
                                                 set according to the AttrModifier*/
     virtual Error  Set(void*, Bool,          /* Set the attribute value according to
                        AttrModifier*);          the AttrModifier*/
     virtual Error  SetResourceValue(void*,   /* Set the value of the associated
                                     Bool);      resource */
     virtual Error  SetToDefault();           /* Set the attribute value to the
                                                 default value */
     virtual Error  Add(void*, Bool);         /* Add an (a set of) element to the
                                                 attr. value; defined for set-valued
                                                 attributes only */
     virtual Error  Remove(void*, Bool);     /* Remove an (a set of) element from the
                                                 attr. value; defined for set-valued
                                                 attributes only */
     virtual Result CheckAction(ActionInfo*); /* Check if the action may be executed
                                                 successfully */
     virtual Error  Filter(FilterItem*);     /* Evaluate the filter item */
     virtual Error  GetAny_ty(Any_ty*);       /* Fills the input parameter with the
                                                 any defined by type related to the
                                                 theAttributeValue value */
...};
```

Even though it is not possible to describe the meaning of each method in detail, it is important to note that the principle is to avoid the diversity of the attribute types by casting and storing the attribute value in `theAttributeValue` class attribute. In this way it is possible to define a common `Attribute` class rather than different classes, one for each attribute type; different attribute classes are subclasses of the `Attribute` class. The attribute value is a string that contains the parsed value[1]. Let us consider the following example. Suppose we have this ASN.1 value:

```
SimpleSyntax ::=  CHOICE {
        numberINTEGER,
        stringOCTET STRING,
        objectOBJECT IDENTIFIER,
        empty        NULL }
```

[1]Please note that not all the attributes must be “stored”. In fact some of them are very dynamic and keeping their values wouldn’t make any sense. In this case `theAttributeValue` attribute points to `NULL` and when such attribute is accessed its value is retrieved via the `Get` method.

Possible values are `(number = 25)` or `(string = "Hello world !")`. This approach, similar to the one used by the IBM CmipWorks platform [10] that represents the syntax values using strings, offers several advantages:

- the use of C structures to store attribute values gives rise to the problem of memory management (allocation, copy, freeing), for instance when an element is added to or removed from a 'set/seq. of'; with the use of the string representation is the system that takes care of it simply concatenating or deleting substrings.
- the string representation is closer to the ASN.1 than a C type; this hides the way that the value is stored inside the attribute, which may be the string itself (like in our case), a C type of even an XOM object depending on the user requirements.
- a string is stored in a single adjacent area of memory rather that a C type that may contain sub-pointers; this is extremely useful to implement a persistent attribute because we simply have to store the string value.

To provide the highest flexibility the string and the C type representations are used. The attribute methods have a boolean flag; when this flag is set to FALSE the value, cast to `void*`, contains the corresponding C structure, otherwise it contains the string value cast to `void*`. Because the attribute value is always stored in the string format, in the first case the parsing function is invoked to convert the value.

The complexity of the filters, the addition/deletion of members to/from the attribute, and the filling of any defined by's are moved from the library kernel to each `Attribute` class or subclasses. The kernel of the library handles attributes as if they were of the same type, and each `Attribute` hides the complexity and the peculiarities of such attributes. Another advantage of this approach is that the structure of the library is not complex, thus enabling the possible errors to be confined to the `Attribute` subclasses. It is also possible to define two attributes that share the same syntax but with different methods or vice versa (e.g. same ASN.1 type with different tagging) defining a common superclass and deriving from that class the two subclasses that redefine only the non-common section.

The Attribute class is used in the `EventForwardDiscriminator(EFD)` and `Log` MOCs to handle the notifications. When a potential notification is to be evaluated, the elements of the notification may be seen as a set of `Attribute` instances. The evaluation process is simply a sequence of `Attribute::Compare` method calls. In fact the `DiscriminatorConstruct` attribute, present in `EFDs` and in `LogRecords` instances, is a filter to be evaluated against the set of the attributes present in the potential notifications. This is the same process that is executed to select instances by means of filtering in a CMIS operation.

In order to exploit the inheritance relationship supported by the GDMO, each class should define its own characteristics and allow subclasses to add or redefine characteristics without having to rewrite everything from scratch. For such reason the `ManagedObjectClass` and the `Package` class contain the following methods:

```
class ManagedObjectClass {
  ...
  public:
    Error AddPackage(Package*);
    Error AddNameBind(BindingRecord*);
  ... };
```

```
class Package {
  ...
  public:
    Error AddAttribute(Attribute*);
    Error AddAttrGroup(AttributeGroup*);
    Error AddNotific(Notification*);
    Error AddAction(Action*);
  ... };
```

With this framework each class can handle and manage packages, attribute groups, actions and notifications, and identify them with the respective ObjectIds. Each `AddXXX` method simply adds the input instance to the `theListOfXXX` list of instances. In fact each MOC constructor method contains the methods to add actions, packages, etc., dynamically to the class. This feature is important for handling the run-time definition of the MOCs.

Suppose we have defined the following GDMO class:

```
exampleClass MANAGED OBJECT CLASS
    DERIVED FROM fatherClass1, fatherClass2;
    CHARACTERIZED BY examplePackage1, examplePackage2;
    CONDITIONAL PACKAGES
           examplePackage3 PACKAGE
                  ACTIONS action1, action2;
                  NOTIFICATION notification1;
    REGISTERED AS {joint-iso-ccitt ms(9) smi(3) part4(4) package(3)
                   examplePackage3(3) }
    PRESENT IF !....!
REGISTERED AS {joint-iso-ccitt ms(9) smi(3) part4(4) managedObjectClass(3)
               exampleClass(0) }
```

That class will be handled in the following way:

```
class exampleClass : public fatherClass1, fatherClass2 { .....};

exampleClass::exampleClass()
{
  Package       *examplePackage1, *examplePackage2, *examplePackage3;
  Action        *action1, *action2;
  Notification  *notification1;
  BindingRecord *theNameBinding;

  /* Instanciate and fill every instance */
  ....
  AddPackage(examplePackage1); /* Add the package to theListOfClassPackages list */
  AddPackage(examplePackage2); /* Add the package to theListOfClassPackages list */
```

```
if(...) /* The condition on examplePackage3 is satisfied */
{
  examplePackage3->AddAction(action1);        /* Add action1 to examplePackage3 */
  examplePackage3->AddAction(action2);        /* Add action2 to examplePackage3 */
  examplePackage3->AddNotific(notification1); /* Add notification1 to examplePackage3 */
  AddPackage(examplePackage3); /* Add the package to theListOfClassPackages list */
}
  AddNameBind(theNameBinding); /* Add the NB to theListOfClassNameBindings list */
  ....
};
```

When `exampleClass` is instantiated, the following methods are called in this order:

- `fatherClass1::fatherClass1, fatherClass2::fatherClass2`
- `exampleClass ::exampleClass`

Hence `exampleClass` exploits the inheritance relationship to instanciate its own attributes once the father classes have automatically instantiated their own. Another important feature shown by the example is how to express the GDMO multiple inheritance (`exampleClass` is derived from both `fatherClass1` and `fatherClass2`).

Allomorphism, that is the ability for a managed object to act as if it were a member of another object class, may also be easily expressed with the proposed framework, by exploiting the value of the `allomorphs` attribute, member of the `ManagedObjectClass` class, which contains the list of all the allomorphic classes.

2.2 Syntax management

Syntax management is handled in two different ways, depending on the availability in the OSI stack of an automatic encoding/decoding (enc/dec) utility. Some implementations such as the Finsiel one [11, 12], provide automatic enc/dec; others (eg. Isode [13]) do not. In the former case the syntax must be registered at the initialization time and the PDUs are automatically encoded/decoded; in the latter the application has to encode/decode the PDUs, and no syntax registration is needed.

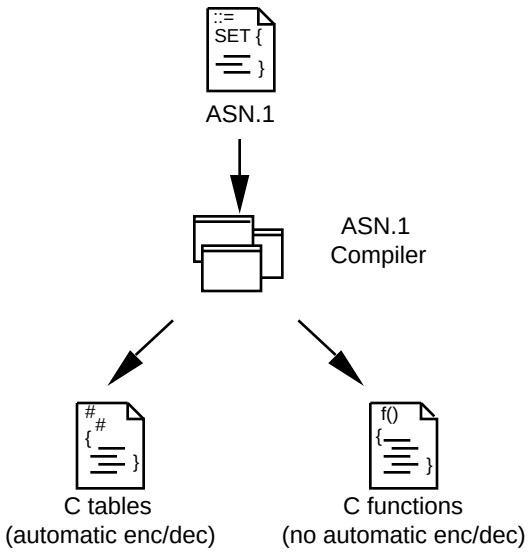

Fig. 1 - ASN.1 compiler

Generally each OSI stack has an off-line compiler that takes as input an ASN.1 file and produces tables or functions for the enc/dec depending on the presence of automatic enc/dec utility.

In both cases the `WITH INFORMATION SYNTAX` and `WITH REPLY SYNTAX` productions (as well as the `Parameter` template) are used to identify exactly which position the ASN.1 data type may occupy in a PDU. This has the following advantages:

- strong checking: the errors during the enc/dec are easily identified by limiting the set of possible data types to analyze;
- enhanced performance: during the enc/dec of specific sections of a PDU only specific data types that may occur in those sections are scanned.

Depending on the presence of automatic enc/dec, two approaches may be adopted to handle the syntax management:

- read the file containing the tables for the enc/dec automatically at run-time and register these syntaxes (automatic enc/dec);
- integrate the functions for the enc/dec in methods of the `Attribute` class that are called when a specific data type is to be managed (no automatic enc/dec).

In the first case a library function is provided:

```
Error RegisterAbstractSyntax(char* syntaxFile, char* contextFile);
```

where the `syntaxFile` is the path name of the file containing the tables for the enc/dec to be registered and `contextFile` is the path name of the file that contains

information related to the context of the attribute. Suppose we have defined the `SimpleAsn1Type` type inside the `SimpleModule` ASN.1 module. The ASN.1 compiler (Figure 1) will produce for this type the `SimpleAsn1TypeTable` table that contains the information to be registered to handle the enc/dec automatically. Supposing that this data type may be present in an action info or in an event reply, an entry of the context file is:

```
SimpleModule.SimpleAsn1Type; {1, 9, 3, 4, 7, 1}; ACTION-INFO, EVENT-REPLY
```

The `RegisterAbstactSyntax` function simply links the information in the two input files and automatically performs the registration of the data types.

If an automatic enc/dec is not present, the `Attribute` class provides the following virtual methods:

```
class Attribute {
 ...
public:
   virtual Any_ty* Encode(Error);    /* Encode the value contained in theAttributeValue
                                        attribute */
   virtual Error   Decode(Any_ty*);  /* Decode the value contained in the input
                                        parameter and stores it into the
                                        theAttributeValue attribute */
   Error   RegisterAttrContext();    /* Register the attribute context */
 ... };
```

The Any_ty type contains the encoded attribute value that will be stored inside the PDU; this is equivalent to the Isode PE. When a new class is registered, all the attributes register their context. When a data type is encoded/decoded in the table corresponding to the context where the data type is present, the attribute matching the ObjectId of the data type is found. When found, the `Encode/Decode` method is called.

The approach described in this section allows a generic framework to handle MOCs to be defined. The inheritance and the containment relationship are handled and the GDMO constructs are implemented in a natural and extensible way. The next section shows how it is possible to automate the implementation of MOCs.

3. Automating code generation

A network management system is not very useful as long as it does not offer tools to automate the addition of new MOCs. Figure 2 depicts the steps needed to add new MOCs to the management library.

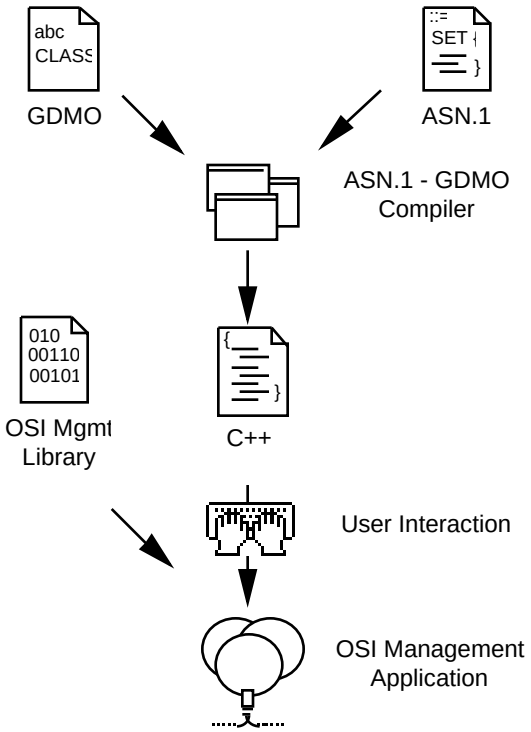

Fig. 2 - Process of building an OSI management application

The user defines his own MOCs in GDMO and their syntax in ASN.1; the GDMO/ASN.1 compiler produces the code that implements the MOCs to be modified by the user in order to define behaviour-dependent functionalities. The last step is the integration of the library with the C++ code to produce the final OSI management application.

This architecture uses the management library as the kernel of the management application and exploits the GDMO/ASN.1 compiler to define the code needed to implement the MOCs. A user interaction is needed in this process for the following reasons:

- the behaviour in GDMO is defined in plain English and therefore the code implementing the behaviour clause cannot be automatically generated;
- the interaction with the real resources still has to be added.

Some protocols exist that define interaction with the real resources, while no standard or general formal approach exists for the definition of the behaviour. All the code implementing a new MOC may be produced by an automatic tool, except for some modules: actions, notification and the behaviour-dependent sections. In fact, for instance, a method to trigger an action for a certain MOC may be automatically generated but this method can't be 'filled' by an automatic tool because the behaviour of this action is described in plain English in the DEFINED AS

section of the action template. Analogously, attributes are often behaviour dependent and a get/set operation on an attribute usually causes an interaction with the corresponding real resources.

The GDMO/ASN.1 tool takes as input an ASN.1 file containing the abstract syntax and a GDMO file containing the MOCs definition and produces the code to handle the syntax and to manage the MOCs. Note that this tool can be split into a GDMO compiler and an ASN.1 compiler. The former generates the code for the MOCs and the latter the data structures (functions) to manage the syntax. This may be useful because an OSI stack generally already has an ASN.1 compiler. The GDMO compiler is simply a tool that 'assembles' attribute, behaviour, action and notification instances driven by the GDMO definitions. All this architecture works if the `Attribute` subclasses are defined for each data type because the basic module of this assembly is the `Attribute` class instance. The code for the `Attribute` subclasses may be generated in two ways: either the existing ASN.1 compiler is not modified and so continues to produce C data structures and enc/dec functions while a new tool produces the attribute subclasses, or a single tool is implemented possibly modifying and integrating an existing ASN.1 compiler. In the former case the Attribute compiler must know the naming rules used by the ASN.1 compiler to define the C types that it takes as input and that have been generated from ASN.1. Note that the attribute compiler may take as input the ASN.1 rather than the C types to generate the `Attribute` subclasses; also in this case the naming rules for the compiler must be the same.

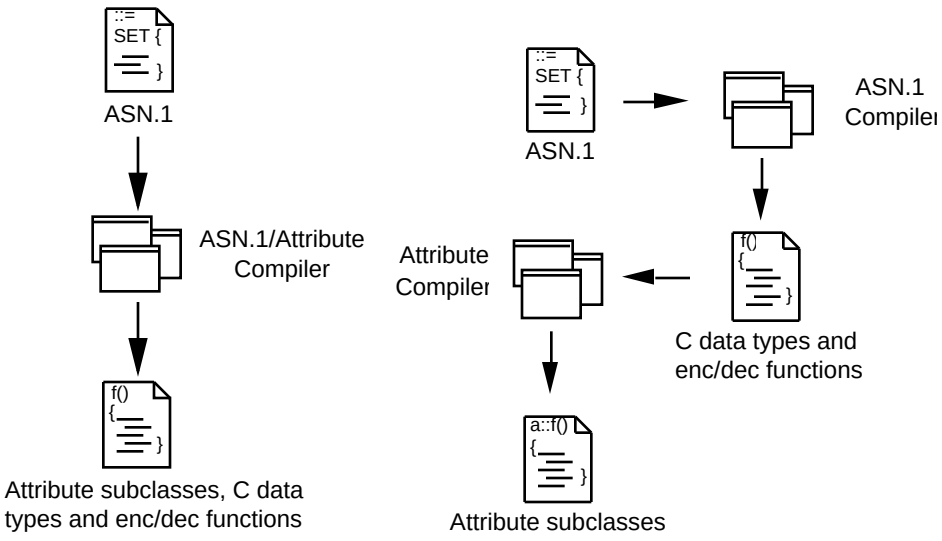

Fig. 3 - Generation of code for the Attribute subclasses

The Attribute compiler produces the copy, compare, free, print, parse, add and remove (when applicable) methods for each `Attribute` subclass derived from ASN.1.

This compiler is based on the 'divide et impera' principle. It is not possible to generate functions to compare, add, etc., general C types because the meaning of the type fields is not always known. Instead this may be done if the C types are generated, as in our case, by the ASN.1 compiler, because the naming rules for the C types are known and the different type structures generated by the ASN.1 compiler correspond to the number of ASN.1 keyword constructs. Thus by defining the methods to copy, free, etc., basic ASN.1 types such as `OBJECTIDENTIFIER` or `OCTETSTRING`, it is possible to generate the same methods for each other type as composition of the basic methods. The `Add`, `Remove` methods for set types may be defined as the composition of a function that scans the elements of the set / sequence and a function that compares them. For instance, to add an element to a set-valued type, the `Add` method should scan the set / sequence to find whether the element is already present and, if not, find the position where to insert it.

This architecture work in the most of the cases; it fails when a type has an embedded behaviour because a tool can treat a type as a stream of bytes to compare and free but cannot interpret the meaning of the fields. Consider the following ASN.1 type defined in X.208:

```
GeneralizedTime ::= [UNIVERSAL 24] IMPLICIT VisibleString
```

The Attribute compiler produces a method to compare two GeneralizedTime type instances and treats these instances as a stream of bits. So '19851106210627.3Z' is seen as different from '19851106210627.3-0500' although they are equal because a different representation for the time zone is used. In these cases, the `Compare` method generated by the Attribute compiler must be modified by hand; note that with this modification, the other data types that are a composition of this type automatically work properly because they use the `Compare` method that we have just modified.

The size of the code generated for a specific set of MOC's depends on several factors: the number of the attributes and the complexity of their syntaxes, the fact that some of them may be inherited from other classes, the amount of code needed to implement specific behaviour and the communication with the real resource, if any. The attribute compiler generates about 70-100 lines of code for an attribute class with a relatively complicated syntax and the GDMO compiler about 400 lines of code per MOC. Please note that the user has to customize this generated code by filling some

methods in order to implement the behaviour and to communicate with the real resource.

The framework defined in the previous sections allows automatic tools to be produced to generate the code for new MOCs; it was shown that it is possible do this by integrating existing tools, such as an ASN.1 compiler, without modifying them. This is very important in an industrial environment where backward compatibility is required.

4. A step forward: defining the behaviour

The previous section showed that the full automatic generation of code to implement MOCs is not possible because the behaviour is defined in natural language. The OSI community is currently working on the formal techniques to define the behaviour but a standard in this area is still a long way off. Instead of defining the behaviour, this section proposes that the problem be approached in terms of real implementations until a standard is established.

The idea is to enhance the previously defined C++ classes (`ManagedObjectClass`, `Attribute...`) by adding an attribute of type `char*`. This is for instance the `ManagedObjectClass` definition:

```
class ManagedObjectClass {
private:
  ... /* Former defined attributes */
  char* theBehaviourScript;
  ...};
```

This text field contains a script that is executed every time characteristics of the MOC (or Attribute values) are modified or every time an instance of the class is scheduled. The language used for this script is a high-level language. Note that this is an interpreted language, not a compiled one. The peculiar characteristics of the language are not the keywords but the self-defined variables that contain the state of the application and of every instance currently managed. The richer this set of variables, the more powerful the language. In fact the managing application may be seen as a particular database where the actions are performed depending on the results of queries. For that reason it is more important to provide powerful query functions rather than complex language constructs; this is exactly what happens with 4th generation query languages. Practically, the language has to define a set of variables representing the state of the application and a number of constructs that basically hide calls to class methods. Every time that the application calls the script,

the event type (`attributeValueChange`,`objectDeletion`...) is passed as a function argument. In this way, for instance, each `ManagedObjectClass` script has this skeleton

```
switch(theEvent) {
       case attributeValueChange:
              /* Execute the appropriate action */
       break;
       case objectDeletion:
       ... }
```

and the skeleton of each Attribute script is the following

```
switch(theEvent) {
       case getValue:
              /* Execute the appropriate action */
       break;
       case setValue:
       ...
}
```

This approach is totally event-driven:

- the script is executed only when there is an event to serve;
- the application can choose to catch the event filling the corresponding case branch with a non-empty action.

This language has powerful functions that allow:

- interaction with other MIB MO instances;
- attribute values to be obtained / set;
- execution of such actions as to call a C function or to issue a notification.

Suppose for instance that when the attribute K of the class J is modified, all the managed object instances (MOI) contained whose MOC is X, should set the attribute Y to the default value. In this case the script for the attribute J of the class K is:

```
switch(theEvent) {
       case setValue:
              InstanceSet z;
              Instance    w;
              -- The input for the SEARCH function is a string that contains
              -- a CMISFilter
              z = SEARCH("(item (equality (managedObjectClass = X)))");
              -- Now z contains all the MOI's that satisfy the criteria
              FOR EACH w in z DO
                BEGIN
                     -- For each MOI in the set of selected instances
                     -- the method SetToDefault() is called on
                     -- his Y attribute (this is equivalent to the C++
                     -- method call w->Y.SetToDefault())
                     SEND MESSAGE "SetToDefault" WITH PARAM "" TO Y
                END
       break;
       ... /* Other cases */
}
```

As seen in this example, the language allows interaction with other MOIs using a high-level API. Following this approach, it is easy to define the behaviour by simply implementing the script that formalizes what is written in plain English in the BEHAVIOUR clause. Because this language is close to C++, it is easy to write a tool that translates this language into C++. In this way the GDMO compiler may take as input a set of GDMO documents and a set of scripts and generate all the C++ code needed to implement the managing application. It is now possible to build an initialization function that reads the GDMO file and the file containing the enc/dec tables and builds the management application at run-time. When the standard that defines the behaviour is published, the script language will be changed for conformity.

Conclusion

This article shows that a library implementing the kernel of an OSI Management application is possible and feasible. The implementation discussed supports all the features present in GDMO, such as inheritance and containment, and offers an easily extensible framework to accommodate future modifications and additions to the OSI standards. Off-line tools have been defined to add new MOCs to the library. Finally a solution to the problem of implementing the behaviour has been proposed.

The final result is an general-purpose environment for full OSI-compliant management applications. The complete coverage of the various GDMO features permits the fulfilment of the largest range of the user requirements and greatly reduces the development efforts required for the production of the final applications. Despite its broad field of applications, the proposed implementation does not introduce additional costs, and the results are also attractive in terms of efficiency and size of code as shown in the section 3.

Acknowledgements

The code examples described in this article are derived from direct experience of the authors in the design and implementation of commercial and public-domain products, which are mentioned in the next section. The authors are aware of other existing products and public-domain implementations based on similar or different approaches, which are not mentioned simply because this article is by no means a review of existing products, nor wants to suggest the adoption of any of them. However, the authors wish to acknowledge the importance of the OSIMIS platform [9], that has greatly contributed to the diffusion of the OSI management and has introduced a number of concepts now integrated in many currently available implementations.

References

[1] ISO / IEC 7498-4: 1989, *"Information processing systems - Open Systems Interconnection - Basic Reference Model - Part 4: Management Framework"*

[2] CCITT Recommendation X.208 (1988), ISO / IEC 8824: 1988, *"Specification of Abstract Syntax Notation One (ASN.1)"*

[3] CCITT Recommendation X.710 (1990), ISO / IEC 9595: 1990, *"Information Technology - OSI, Common Management Information Service Definition (CMIS) for CCITT Applications"*

[4] CCITT Recommendation X.711 (1991), ISO / IEC 9596-1: 1991, *"Information Technology - OSI, Common Management Information Protocol (CMIP) - Part 1: Specification"*

[5] CCITT Recommendation X.701 (1992), ISO / IEC 10040: 1992, *"Information Processing System - Open System Interconnection - System Management Overview"*

[6] CCITT Recommendation X.720 (1992), ISO / IEC 10165-1: 1992, *"Information Technology - OSI - Management Information Services - Structure of Management Information - Part 1: Management Information Model"*

[7] CCITT Recommendation X.721 (1992), ISO / IEC 10165-2: 1992, *"Information Technology - OSI - Management Information Services - Structure of Management Information - Part 2: Definition of Management Information"*

[8] CCITT Recommendation X.722 (1992), ISO / IEC 10165-4: *"Information Technology - OSI - Management Information Services - Structure of Management Information - Part 4: Guidelines for the Definition of Managed Objects"*

[9] G.Pavlou, S.N.Bhatti, G.Knight, "*The OSI Management Information Service: User's Manual"*, Version 1.0, February 1993

[10] G.Geiger, W.Allen, A.Majtenyi, P.Reder, *"IBM cmipWorks: Technical paper"*, IBM, March 1994

[11] *"X/OBJ Agent platform: User documentation"*, Finsiel S.p.A. 1994

[12] *"X/ASN1, ASN.1 Compiler: User documentation"*, Finsiel S.p.A. 1991

[13] M.T.Rose, J.P.Onions, C.J.Robbins, "*The ISO Development Environment: User's Manual"*, Version 8.0, June 1992

Author Information

Luca Deri, formerly a research fellow at the University College of London and member of the OSIMIS development team, is currently working at IBM's Zurich Research Laboratory. He received his degree in Computer Science with a thesis on Network Management from Finsiel S.p.A. where he worked as a consultant after the graduation. His professional interests include OSI management, conformance testing and OO technology. His e–mail address is: lde@zurich.ibm.com.

Eugenio Mattei is working for Finsiel S.p.A. He has been involved in the development of commercial OSI products for several years, leading the team dedicated to the Network Management implementations. He also acted as Finsiel technical project leader in the context of European conformance testing projects specifically dealing with Network Management. His e–mail address is: mattei@tecsiel.it.