



CMIP Run! is a newsletter dedicated to popularizing and explaining the various network and systems management technologies (especially CMIP, the Common Management Information Protocol). **CMIP Run!** provides guest articles on technical topics, some tutorial information, answers to commonly asked questions, and news about management-related events such as standards meetings. The content of each issue will vary. For contact information, see the last page of the newsletter.

Contents:

In This Issue	1
Low-fat CMIP.....	1
Inheritance: It's Not Just for CMIP Any More	4
XOM and XMP for Programmers.....	6
See MIP Answer!.....	8
Standards Activities and Status	11
General Newsletter Information	12

You are free to copy, distribute, or cite information in **CMIP Run!** However, any copy must credit both the contributor and **CMIP Run!** Any trademarks appearing herein are the property of their respective owners. No representations are made as to whether the information can be used clear of IBM or third party patents. No license under IBM patents is given. Further, the information in this publication is provided "as is", without warranty of any kind. Neither the IBM Corporation nor any contributor shall be liable for any loss or damage caused or alleged to be caused directly or indirectly by the information contained in **CMIP Run!**

In This Issue

In our fourth issue of **CMIP Run!**, we once again deal with a variety of topics:

- In the first of a series of articles, Bob Moore and Mark Zelek begin to explore ways to reduce the amount of optionality in CMIP, to make implementations less costly. Their specific topic in this article is optional

elements in syntax.

- Alison Cohen and Mark Zelek discuss possibilities for making SNMP managers and agents object-oriented.
- Vik Chandra provides the first in a series of articles on X/Open's XOM and XMP API's. The first article provides an introduction to the XOM interface.
- In **See MIP Answer!**, MIP responds to his first fan mail, clarifying some aspects of *MIM Think* for a reader. He also answers a number of other questions.
- In **Standards Activities and Status**, Mike Allen reports on some ongoing work in the area of Advanced Peer-to-Peer Networking (APPN) network management.

Your CMIP Run! editorial team

*Vik Chandra
Alison Cohen
Alan Lee
Bob Moore
Julie Spitzer
Mark Zelek*

This is the first in a series of articles that explores what can be done to make CMIP easier to implement.

Low-fat CMIP

When Vik, the newest member of the **CMIP Run!** team and our local SNMP expert, first saw our superhero MIP, he said, "He's fat - just like CMIP." Vik is right on both accounts. We've done a lot of thinking about where the fat is in CMIP, and how to remove it. One of the biggest areas is the reckless use of optionality, both in the protocol itself and in managed object class specifications. All of this optionality makes managed object implementations and the applications that use them more complex than necessary.

Some will try to justify optionality in CMIP on the grounds that it makes implementations more flexible. In practice, this flexibility is often used to "resolve" political disagreements. The disagreements are not resolved by the

use of optionality: they are instead enshrined forever in the GDMO and ASN.1 specifications! This is not power - it is an abuse of power. Managed object implementers can reduce a lot of optionality at the agent by simply deciding not to support it. Implementers on the manager side, however, are not so lucky. A management application *must* be written to tolerate a wide variety of alternatives from managed objects of a given class, yet it cannot depend on finding any particular optional feature in a given instance of the class.

That's the bad news. The good news is that we are not required to use optionality as much as we have been. We can limit its use. The Network Management Forum's document on modelling guidelines (*FORUM TR102: Modelling Principles for Managed Objects*, Technical Report issue 1.0, January 1991) did a good job of explaining why it would be beneficial for optionality to be used only to reflect optionality in the resources being modelled. Unfortunately, these guidelines have not been widely followed.

The first step on the road to reducing optionality in CMIP is to recognize three broad areas where it can creep in:

- OPTIONAL elements in SEQUENCES
- CHOICE syntaxes
- Conditional packages



MIP on a high optionality diet

There is nothing inherently wrong with using optionality in any of these areas. It makes sense for the linked-ID field in an ROIV to be optional; the only practical way to express the syntax of a CMIS filter is with a CHOICE syntax; and wise use of conditional packages can reduce the number of

object classes needed to model a set of resources.¹

The rest of this article will focus on the first area, optional elements in sequences. The other areas will be discussed in future issues of *CMIP Run!* With a bit of effort, we can see MIP lose some weight - and make it easier to implement OSI management.

Consider the heart of a typical CMIP response: all of the elements are optional! In some cases, the entire sequence is optional as well.

```
{managedObjectClass    ObjectClass          OPTIONAL,
 managedObjectInstance ObjectInstance       OPTIONAL,
 currentTime           GeneralizedTime      OPTIONAL,
 usefulStuff           UsefulStuff          OPTIONAL}
```

All of this optionality may appear to be wonderful, because it gives an agent the flexibility to reduce bandwidth usage, by sending only what it considers absolutely necessary in each of its responses. This makes interoperability extremely complicated, since there is so little that a management application can depend on in a response, unless it is developed in concert with an agent. In other words, the manager and agent must reach agreements on what the agent must send and what the manager must receive. Unless these agreements are made public, there's very little chance that the manager will work with any *other* agent that implements the very same managed object classes.

For example, suppose that a management application gets the distinguished name of an instance from a role attribute and wants to find the class of that instance. A simple **M-GET** to the instance requesting no attributes may or may not work. The instance *might* include managedObjectClass in its response to the **M-GET**, or it might not. The only surefire way for an application to determine the class of the instance is to request the value of the *objectClass* attribute in its **M-GET**. The response *might* contain the value of the *objectClass* attribute twice, once in the managedObjectClass field and once in the attributeList field, but it's *guaranteed* to be there at least once, in the attributeList field. If it's there twice, oh well, we wasted a little bandwidth.

Here's another example: suppose an object has a set-valued attribute. Your application would like to add two new values to that attribute via an **M-SET**, and then know all the values that are in the attribute at the end of the operation.

¹ Object-oriented purists may well disagree with the need to reduce the number of classes. However, for non-object-oriented implementations, the cost of "extra" classes is not always worthwhile.

First, make sure the **M-SET** operation is confirmed. If you use the unconfirmed flavor, you'll never get any response.

At this point, however, your application is completely at the whim of the managed object implementation. The agent *might* return the new value of the attribute in its response to the **M-SET**, but it would be equally legitimate for it to return just the `currentTime`, or just the `managedObjectClass`, or even nothing at all.

No problem, you say; I'll just wait for the *attributeValueChange* notification to be emitted. If, perchance, the two values were *already* in the set, the instance might not emit a notification. If the values are truly new, the instance *still* might not emit the notification, depending on the class definition.

Get the picture? There's no way the application can know, unless it waits until it gets the confirmation for the **M-SET**, checks the response *just in case* the instance was nice enough to return the new value of the attribute, and if not, follows it up with an **M-GET** for the attribute. See how all this flexibility in CMIP is making our management applications more complicated?

One more example: suppose you create an instance that has some attributes that are initialized with values that the local system chooses. How do you know what these values are? The response to the **M-CREATE** *may or may not* include the `attributeList` field. If this field is present, it *may or may not* contain the attributes you're interested in. The only surefire way is to be prepared to follow up the **M-CREATE** with an **M-GET** for those attributes. Wouldn't it be much nicer if the management application knew what it could expect from an agent?

The common thread in these examples is that the optional elements in the CMIP response make it more difficult for a management application to get the information it needs. It is unfortunate that the profiles for CMIP (AOM-11 and AOM-12) did not remove some of the options. The next best thing we can do is remove some options in ensembles², since an ensemble can specify conformance requirements above and beyond those in the profiles. In particular, we recommend:

1. Always include `managedObjectClass` and `managedObjectInstance` in responses where they may

² See "*Ensembles: Why They Sound Good*" in *CMIP Run!*, Volume 1 Number 2 for an explanation of ensembles.
VOLUME 2, NUMBER 2

appear.³

2. Always include the `attributeList` in **M-SET** responses.
3. Do not include the `attributeList` in **M-CREATE** responses, since there could be a lot of attributes that are not of interest.

We do not have any strong opinions one way or the other about `currentTime`. If you have strong opinions, let us know.

Next, let's look at optional elements in the syntaxes for characteristics in managed object classes. We examined many definitions and found that by far the most optional elements occur in notifications and action responses. For example, here's the syntax shared by the *objectCreation* and *objectDeletion* notifications:

```
ObjectInfo ::= SEQUENCE {
    sourceIndicator      SourceIndicator      OPTIONAL,
    attributeList        AttributeList        OPTIONAL,
    notificationIdentifier NotificationIdentifier OPTIONAL,
    correlatedNotifications CorrelatedNotifications OPTIONAL,
    additionalText       AdditionalText       OPTIONAL,
    additionalInformation AdditionalInformation OPTIONAL}
```

Our first impression was, "Oh no! More optionality!" As we considered how the absence of any one of these elements would affect a management application, though, we saw that the optionality wasn't really bad in this case. Yes, it is a little wearisome to have to check for the presence or absence of each of these elements, but it isn't difficult. The key difference between this example and the previous case of a typical CMIP response is that a management application can always assume a default meaning if a particular element is not present in a notification, whereas it cannot always make such inferences about the missing elements in a CMIP response. Here is what a management application can assume if the elements of `ObjectInfo` are not present:

- `sourceIndicator`: the cause of the creation/deletion is not known
- `attributeList`: no attributes are included - a manager can do an **M-GET** for any attributes that it needs
- `notificationIdentifier`: the agent is not correlating this notification with any other notifications
- `correlatedNotifications`: the agent has not correlated any other notifications with this notification

³ For managed systems that support scoping, this can actually simplify matters. These two elements are required on responses to scoped operations, so by always including them for unscoped operations as well, an agent can cut down on the special case processing for scoped operations vis-a-vis unscoped ones.

- additionalText: there is no additional text
- additionalInformation: there is no additional information

The lesson we've learned is that if a default meaning can always be inferred when an optional element is not present, the optionality does not make management applications any more difficult (parsing per se does get a little harder). In such cases, we encourage you to use the ASN.1 DEFAULT construct to indicate what the desired default meaning is - please don't leave others guessing! If a default meaning cannot always be inferred when an optional element is missing, make it mandatory! By limiting the use of optional elements in this way, we can start CMIP on the path to fitness, implementability, and (therefore) a long and productive life.

Next time: "Choices: Make One!"

Bob Moore
Mark Zelek

Bob Moore is a member of Network and Systems Management Architecture in RTP. He is currently working on integrating SNMP and CMIP. Formerly he represented IBM in the U.S. committee for OSI Systems Management, ASC X3T5.4, and attended ISO/IEC JTC1/SC21/WG4 as a U.S. delegate.

Mark Zelek is a member of Network and Systems Management Architecture in RTP. He is currently working on integrating SNMP and CMIP. He was an author of the Network Management Forum's managed object library and Ensembles documents for OMNIPoint 1.

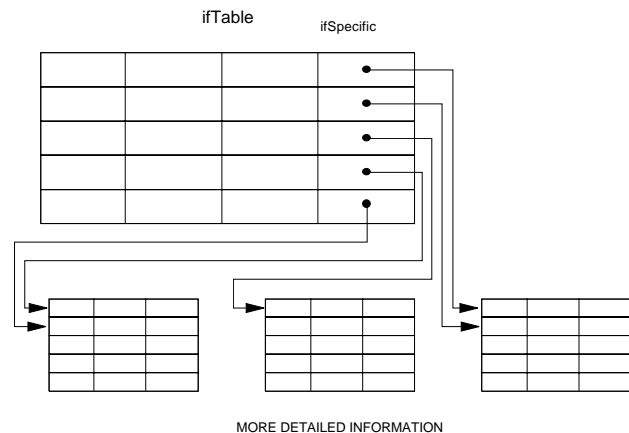
Inheritance: It's Not Just for CMIP Any More

Many believe that CMIP's biggest advantage over SNMP is that CMIP is object-oriented, whereas SNMP is object-based. An object-based design encapsulates functions and data into bundles called *objects*. An object-oriented design features objects, but also defines groups of objects that share the same functions, called *classes*, and defines *inheritance* relationships between those classes. Inheritance can make it easier to extend programs and can open up the possibility of re-using code, since a subclass inherits the functions of its superclasses. Object-oriented design is just a set of techniques, in the same way that structured programming is just a set of techniques. Hence, inheritance never really

belonged *just* to CMIP.⁴ Assembly languages, for example, don't have features that support structured programming, but assembly language programs *can* be well-structured. Similarly, SNMP does not have features that support object-oriented design, but both MIBs and the applications that use them *can* re-use code and be easy to extend. SNMP can simulate some aspects of inheritance, as we shall see in this article. Rather than debating at length whether SNMP could ever truly be object-oriented, what's important is that managers and agents be easily extendible in a way that re-uses as much code as possible. This article will show you a few techniques for doing this with your SNMP MIBs and the applications that use them.

Okay, you say, but who would *ever* really want to simulate inheritance in SNMP? It has already been done, by the folks who gave us MIB-II, quietly and without fanfare. Let's look at two different techniques they used and see how these techniques simulate some aspects of inheritance.

One technique is used in the interface table (*ifTable*) in the interface group in MIB-II. The interface table contains information common to any interface, plus a pointer to more detailed information about that interface (*ifSpecific*) and the type of that interface (*ifType*). The figure below shows how this table works.



Notice that the pointers in the interface table can point to different tables. The interface table itself represents information common to all types of interfaces - i.e., a superclass! The tables with more detailed information represent particular types for interfaces - i.e., subclasses!

⁴In fact, there are certain features in the *Management Information Model* (CCITT Rec. X.720 (1992) | ISO/IEC 10165-1:1992) that *are not* object-oriented, such as conditional packages.

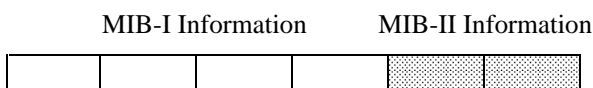
Viewed in this light, the *ifType* variable is just an identifier of a particular subclass. Purists will argue that this simulation of inheritance is not transparent. A management application must know that it has to follow the *ifSpecific* pointer to the tables containing more detailed information. So what! It's a nice, extendible design. That's what matters.

Suppose that an agent has implemented MIB-II and three specific types of interfaces, and that a manager has been written that understands the interface table and these three extensions. Now, let's add a fourth type of interface and see what happens to the agent and the manager.

The agent will need to support a new value for *ifType* and a new table for the new type of interface. The manager can already support the generic aspects of the fourth type of interface. If it doesn't recognize the value of *ifType*, it simply doesn't follow the *ifSpecific* pointer, because it would not understand the information in that table. By being written in this manner, the manager gets the same benefits that allomorhism gives OSI management applications. Who would have imagined that it would be easy to do something akin to allomorhism using SNMP?

Continuing the example, to support the specific details of the fourth type of interface, the manager must recognize the new value for *ifType* and understand the new table.

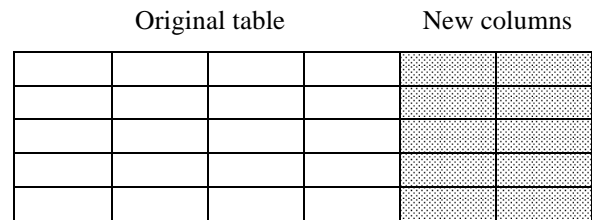
Next, let's take a look at the second technique, which was used in the systems group. MIB-II added new variables to this group by simply extending it. The original group functions as a superclass. The additional variables represent what has been added in a subclass. Note that this technique does not allow for multiple subclasses.



It's pretty easy to see how this makes it easier to extend agents. Note that managers that only know the "old" information can still work with agents that support the "new" information. The managers simply don't ask for the "new" information!

Not only can this be done for groups, it can be done for tables as well, as illustrated in the figure below. When walking a table, a manager should check if the OBJECT IDENTIFIER returned in a **GET-NEXT** is past the last column in the table that it knows about. This way, when new columns are added, the manager continues to work as it did before. Once again, notice the similarity to

allomorhism. And it's not difficult!



Are there any aspects of inheritance that cannot be supported with SNMP? We haven't fully investigated this area, but there are some aspects that might be tricky. Multiple inheritance, for example, could perhaps be simulated by having more than one generic table point to the same specific table. It's not clear whether this approach would be worthwhile.

A pure virtual function is an operation that is defined in an abstract class because the operation applies to all subclasses of that class, but there is not enough information in the abstract class to determine how the operation can be implemented. Given SNMP's (intentional) lack of imperative commands, it's not obvious at first glance whether pure virtual functions could be simulated, or if it would even make sense to simulate them.

Clever readers will no doubt find other ways to simulate aspects of inheritance, perhaps even in the areas that we haven't fully investigated. We invite you to share your ideas.

*Alison Cohen
Mark Zelek*

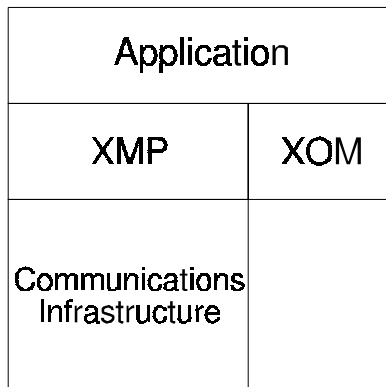
Alison Cohen is a member of Network and Systems Management Architecture in RTP. She is currently implementing network management applications.

Mark Zelek is a member of Network and Systems Management Architecture in RTP. He is currently working on integrating SNMP and CMIP. He was an author of the Network Management Forum's managed object library and Ensembles documents for OMNIPoint 1.

XOM and XMP for Programmers

This is the first in a series of articles on the XOM and XMP X/Open API's to be presented in *CMIP Run!* These articles provide an overview of the XOM and XMP API's from a programmer's perspective. The first article provides an overview of OM-objects of the XOM API; the future articles will dive further into XOM, XMP, and associated implementation techniques.

The X/Open OSI-Abstract-Data Manipulation (XOM) API is a standard API for dealing with ASN.1 data types in C. XOM provides a new construct called the OM-object (yet another kind of object!), and interface routines to manipulate these objects. XOM by itself is not particularly useful. It is intended to coexist with other application-specific API's, such as the X/Open Management Protocols (XMP) API.



The typical XOM and XMP-based environment is shown in the figure above. However, there will ordinarily be more than one application sitting on top of the XOM and XMP API's. Notice that XOM does not have anything underneath it, because it is not used to communicate with other entities in the network.

As mentioned earlier, XOM uses OM-objects to represent ASN.1 data types in C. OM-objects are mildly object-oriented. They have attributes and inheritance for these attributes, but no methods⁵ and no notion of data encapsulation.

There are two kinds of OM-objects, private and public. The public OM-objects are directly accessible to the client

⁵The methods are the interface routines provided by XOM - however, these are independent of the OM-object definitions.

application, whereas the private OM-objects are only accessible via the interface functions. The public OM-objects are in C structures defined by the XOM standard. The private OM-objects can exist in any implementation-specific form. There are two kinds of public OM-objects, service-generated and client-generated. The service-generated public OM-objects are created by the interface service in storage that it allocates, whereas the client-generated public OM-objects are created by the client in storage allocated by it. However, the client-generated public OM-objects are typically statically defined.

In order to understand OM-objects, the public form of the OM-object should be studied. The C data type OM_object is defined to be a pointer to an OM_descriptor. The OM_object and the associated C structures are defined as follows:

```
typedef struct OM_descriptor_struct *OM_object;

typedef struct OM_descriptor_struct {
    OM_type        type;
    OM_syntax      syntax;
    OM_value       value;
} OM_descriptor;

typedef OM_uint16  OM_type;           // 16 bit unsigned

typedef OM_uint16  OM_syntax;        // 16 bit unsigned

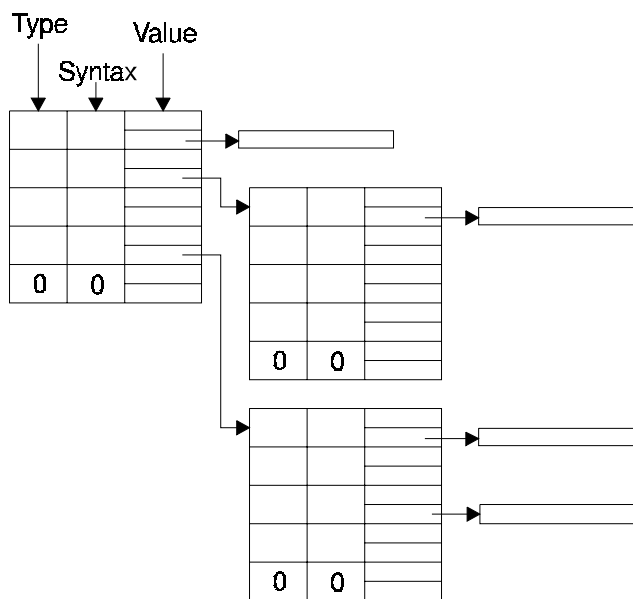
typedef union OM_value_union {
    OM_string      string;
    OM_boolean     boolean;         // 32 bit unsigned
    OM_enumeration enumeration;    // 32 bit signed
    OM_integer     integer;        // 32 bit signed
    OM_padded_object object;
} OM_value;

typedef struct {
    OM_string_length length;       // 32 bit unsigned
    void             *elements;
} OM_string;

typedef struct {
    OM_uint32       padding;
    OM_object       object;
} OM_padded_object;
```

Notice that XOM does not support REAL values, and that the largest integer supported is 4 bytes. Work is underway within X/Open to address these shortcomings.

Here is what a public OM-object looks like:



An OM_object (the C typedef) points to the first element in an array of OM_descriptors. Each descriptor contains one attribute of the OM-object. The first descriptor always contains the attribute "class", which is an OBJECT IDENTIFIER to identify the class of the OM-object. The "class" attribute is inherited from the OM-object class "object". The OM-object class "object" is the highest class in the OM-object inheritance hierarchy. The last descriptor in the array contains NULLs in the type, syntax, and value fields.

The type field of an OM_descriptor distinguishes one OM attribute from other OM attributes. The lower 10 bits of the syntax field identify the OM syntax of the attribute and the upper 6 bits further qualify the OM syntax. The value field is an 8 byte field. If the value is boolean, enumerated, or integer, the first 4 bytes contain the value and the last 4 bytes are not used. If the value is a string, the first 4 bytes contain the length of the string and the last 4 bytes point to the string. If the value is an object, the first 4 bytes are not used and the last four bytes point to an array of OM_descriptors.

The ASN.1 data types can be mapped to XOM C constructs in the following manner.

ASN.1 Construct	C Structure
ANY	any syntax
BIT STRING	OM_string
BOOLEAN	OM_boolean
CHOICE	OM_object
ENUMERATED	OM_enumeration
GeneralizedTime	OM_string

GraphicString	OM_string
INTEGER	OM_integer
ObjectDescriptor	OM_string
OBJECT IDENTIFIER	OM_object_identifier (OM_string)
OCTET STRING	OM_string
REAL	-
SET	OM_object
SET OF	OM_object
SEQUENCE	OM_object
SEQUENCE OF	OM_object
UniversalTime	OM_string

The following shows how the CMIS Event Report Argument is represented with an OM-object. The ASN.1 definition of EventReportArgument is:

```
EventReportArgument ::= SEQUENCE {
    managedObjectClass ObjectClass,
    managedObjectInstance ObjectInstance,
    eventTime [5] IMPLICIT GeneralizedTime OPTIONAL,
    eventType EventTypeId,
    eventInfo [8] ANY DEFINED BY eventType OPTIONAL}
```

The OM-object class corresponding to a CMIS Event Report Argument is defined as follows:

OM Attributes of a CMIS-Event-Report-Argument

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
managed-Object-Class	Object(Object-Class)	-	1	-
managed-Object-Instance	Object(Object-Instance)	-	1	-
event-Time	String(Generalised-Time)	-	0-1	-
event-Type	Object(Event-Type-Id)	-	1	-
event-Info	any	-	0-1	-

The OM-object class definitions are written up in tables along with some text which further qualifies the definition. The CMIS-Event-Report-Argument OM-object can have up to 6 attributes: the five that are shown in the table above, plus the attribute "class" inherited from the OM-object class "object". Normally, the inherited attributes are not shown in the class definition. The text that accompanies a class definition identifies the superclasses of the specified OM-object class.

The first column in an OM-object class definition identifies the attributes of the OM-object class. The second column defines the syntax of the attribute. The third column identifies any length constraints that may exist on the length of the value of the attribute. The fourth column defines the rules of presence. The rules of presence are further qualified in the accompanying text. The final column specifies any initial values of the attributes that may exist.

In the example above, the attribute event-Time is optional. If present, it can only be present once in an instance, and its

syntax is an OM_string. The attribute event-Info is optional. Its syntax is "any", which means that it can be of any OM_syntax. In the case of an Alarm, the attribute event-Info has the syntax Object(Alarm-Info), which means that the syntax field of the OM_descriptor contains a 127 (OM_S_OBJECT) and the last four bytes of the value field point to an instance of OM-object class Alarm-Info.

The attribute managed-Object-Class must exist once in every instance of the OM-object class CMIS-Event-Report-Argument. The syntax for the attribute managed-Object-Class is another OM-object of class "Object-Class". This follows directly from the ASN.1. In the EventReportArgument SEQUENCE, managedObjectClass is of type ObjectClass, which is defined to be the following:

```
ObjectClass ::= CHOICE {
    globalForm      [0] IMPLICIT OBJECT IDENTIFIER,
    localForm      [3] IMPLICIT INTEGER }
```

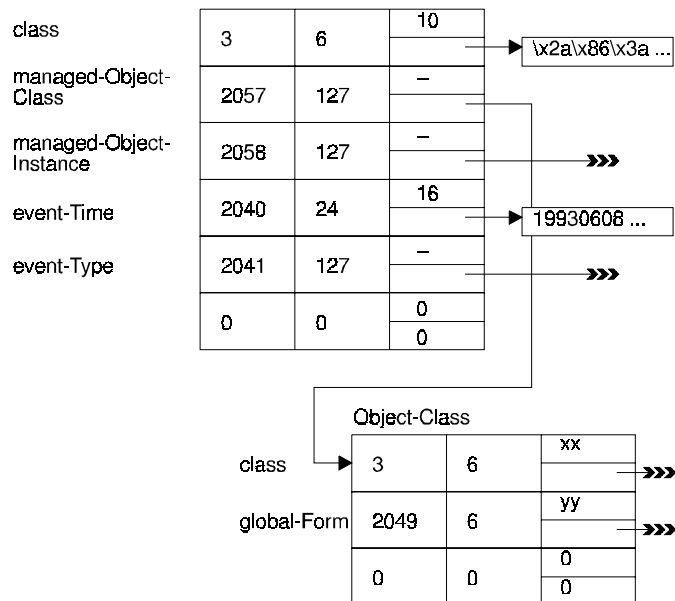
The corresponding OM-object class "Object-Class" is defined to be the following:

OM Attributes of an Object-Class

OM Attribute	Value Syntax	Value Length	Value Number	Value Initially
global-Form	String(Object-Identifier)	-	0-1	-
local-Form	Integer	-	0-1	-

Exactly one of the attributes may be present in an instance.

The similarities between ASN.1 definitions and the corresponding OM-object classes should be obvious.



The figure above only shows a portion of a public OM-object instance of the OM-object class CMIS-Event-Report-Argument. The event-Info attribute is missing, and the details of various attribute values have been left out.

As discussed earlier, the first OM_descriptor of a public OM-object instance always contains the attribute "class". The type for the attribute "class" is 3 (the types may vary across implementations) and the syntax is 6 which is the #define OM_S_OBJECT_IDENTIFIER_STRING from xom.h. The first 4 bytes of the value contains the length of the OBJECT IDENTIFIER and the last 4 bytes of the value point to the BER-encoded OBJECT IDENTIFIER.

The type for the attribute managed-Object-Class is 2057 and its syntax is 127 (OM_S_OBJECT). The first 4 bytes of the value are meaningless, and the last 4 bytes of the value point to a public instance of the OM-object class "Object-Class".

In the instance of OM-object class "Object-Class", the first OM_descriptor is used to contain the attribute "class". In this example the global-Form attribute is present, the syntax of which is 6 (OM_S_OBJECT_IDENTIFIER_STRING) and the value contains the length of the OBJECT IDENTIFIER along with the pointer to the BER-encoded OBJECT IDENTIFIER.

Each array of OM_descriptors contains a NULL OM_descriptor at the end.

This article has presented an introduction to OM-objects. The next article in the series will discuss the interface functions provided by XOM.

Vik Chandra

Vik Chandra is a member of the Network and Systems Management Manager Application group in RTP. He is currently implementing a network management application. He is also IBM's primary SNMP contact and the SNMP registration authority.

See MIP Answer!

See MIP provide superhuman answers to commonly asked questions from our readership. Contact **CMIP Run!** with any questions on network and systems management that you would like MIP to answer.



Q: Suppose I have two related resources: one that I want to model with a managed object and another for which I only need to know its type. I'm using an attribute in the managed object that tells the type of the other resource. Which is a better syntax for this attribute, INTEGER or ENUMERATED?

A: If you have a *complete* list of the types of the other resource, ENUMERATED will work. If you have any doubt about whether you might want to add new types in the future, play it safe and stay away from ENUMERATED. Why? Once you've defined an attribute and registered it, you cannot change any aspect of that definition. This means you can't add a new value to an ENUMERATED type - this changes the syntax of the attribute (albeit ever so slightly). Defining a new ENUMERATED type won't help either, because the attribute template refers to the existing ENUMERATED type. You can't change that reference without changing the template, which means you've defined a new attribute. It's even worse, because now you'll need a new package and a new class for that new attribute.

INTEGER is a safer way to go. New values can be assigned, but there is one important consideration: how can you be sure that the values you choose will not be independently chosen by someone else for some entirely different meaning? Unless someone, such as the original managed object definer, hands out new values, collisions will result.

Well, you say, I'll just define my own subclass, and then I can choose my own values and not have to worry about collisions. This approach will work, but it's not very

desirable. Suppose the original attribute definer assigned meanings to values 0 through 5. You define a subclass that assigns meaning to value 6. Oh, and by the way, two other vendors define their own subclasses, and both give the value 6 different meanings. Yes, the value 6 is unique, provided you know in which subclass it appears. Yuck! I wouldn't recommend this approach at all.

If you expect others to define subclasses of one of your classes with one of these kinds of attributes, and you know that new values are likely to be needed, the absolute safest approach is to use OBJECT IDENTIFIER as your syntax rather than INTEGER. This approach is more costly, since it's easier to manipulate an INTEGER than an OBJECT IDENTIFIER, but if you expect others to build off of your definitions and you don't plan to hand out values, OBJECT IDENTIFIER is the way to go.

Q: In *CMIP Run!* Volume 2, Number 1, I have read the article on "MIM Think" dealing with the problem of interpreting more into GDMO definitions than they really imply. In particular, your statement was very surprising to me that the **GET** property of an attribute should not imply that the attribute is "**GET** only". Reading the standards on this topic very carefully, I have found the following statements in ISO 10165-1:

Clause 5.3.3.3 Replace attribute value

scope: This operation applies to attributes that are defined as writable.

Clause 5.3.4.1, Create, Behavior point 2.

...When the managed object is created, explicit values may be assigned to non-writable attributes if explicitly allowed by the managed object class definition. Once assigned, such values cannot be modified by attribute oriented operations...

To me, (using common sense) this means that **GET** really means "**GET** only".

Looking forward for your response,

Brigitte Baer
 European Networking Center
 IBM Heidelberg
 Germany
 e-mail: bschott at DHDIBM1.bitnet

A: Wow! I'm impressed that you uncovered this! You came very close to finding a defect in MIM, Brigitte. There's an extremely subtle point that will explain why this isn't a defect. Clause 5.3.3.3 says "... attributes that are *defined* as writable." Clause 5.3.4.1 makes a switch from definition to instance: "When the managed object [instance] is created" MIM doesn't define what "non-writable attribute" means in an instance (as opposed to in a managed object class definition). There are two possible meanings of "non-writable attribute" for a managed object:

- 1: The class definition does not specify replace, replace with default, add, or remove.
- 2: The instance does not support any operations that modify the value of the attribute.

Using the first interpretation, Clause 5.3.4.1 would indeed violate *MIM Think*. Instead, the clause would have to say that if a managed object class definition explicitly allows it, explicit values can be assigned at create time to attributes for which a conformant instance of the class is *not required to accept* operations that modify the value.

Using the second interpretation, however, Clause 5.3.4.1 is correct. Such attributes cannot be modified by attribute oriented operations, because the instance does not support any operations that modify the attributes. I believe the authors of MIM intended this interpretation. They certainly could have expressed it more clearly!

By the way, a lot of confusion has arisen because GDMO never explained what the properties **GET**, **REPLACE**, **GET-REPLACE**, **ADD**, **REMOVE**, and **ADD-REMOVE** mean. Work is underway within the standards bodies to add these explanations.

Q: Can you show me how to make an attribute with the **GET** property in a superclass have the **GET-REPLACE** property in a subclass?

A: It's really pretty simple. Clause 8.3.3 of GDMO explains how it works. If an attribute appears in more than one package that is part of a managed object, the attribute is only present once, but its properties are the logical OR of the properties specified in each of the packages in which it appears⁶.

⁶The PERMITTED VALUES, REQUIRED VALUES, DEFAULT VALUE VOLUME 2, NUMBER 2

In a mandatory package in your subclass, simply include the desired attribute with the **GET-REPLACE** property. The logical OR of **GET** and **GET-REPLACE** is **GET-REPLACE**. That's all you have to do!

Or, you *could* just include the desired attribute with the **REPLACE** property in your subclass, since the logical OR of **GET** and **REPLACE** is also **GET-REPLACE**. In fact, if you have an object-oriented implementation, this way will feel more natural. However, for the sake of those who don't have object-oriented implementations, at least include a comment so they will know what you're doing.

Q: Through multiple inheritance, a class I've defined has two naming attributes. I can use either one, but what should I do with the other? It seems like I'm wasting storage, since if someone requests an **M-GET** operation for the other attribute, I have to supply some value. Any suggestions?

A: Your question reminds me of a debate that I always have with a colleague: are attributes data made visible via operations, or operations that make data visible? Believe me, there is a difference! My colleague and you primarily think of attributes as data, whereas I primarily think of attributes as operations. Recall that the details of how a managed object supplies the value for an attribute are hidden behind the managed object boundary. A managed object might retrieve a field out of a control block, read a database record, or perform some procedure to get the value for an attribute. I always like to think that some procedure is performed - even if all that procedure does is retrieves a field out of a control block or reads a database record.

If you think of attributes as fields in control blocks or columns in database records, you'll have the mindset that one more attribute means one more field or one more column in a database table.

On the other hand, try to think of attributes as function calls (you can "inline" them for efficiency). This means that the "**M-GET xxxID**" function returns some value, and so does the "**M-GET yyyID**" function. It really doesn't matter what value *yyyID* has if it is not being used for naming, so why not just have the two functions retrieve their values from the same place? It doesn't cost you any additional storage!

and INITIAL VALUE properties observe different rules.

Q: I understand that SNMP's **GET-NEXT** operation walks a table in column-major order, i.e., it goes all the way down the first column from top to bottom, then all the way down the second column, and so on. Data in a table, however, is ordinarily organized by row, not by column. Is there really no provision in SNMP for retrieving a row of a table?

A: SNMP has a very nice mechanism for retrieving a row of a table, provided the manager understands the structure of the table. The manager simply issues a **GET-NEXT** with multiple operands, one for each column of the table. If the manager wishes to retrieve an entire table by row⁷, it proceeds as follows: the first **GET-NEXT** is issued with the object identifiers for the columns themselves. The response to this **GET-NEXT** contains the object identifiers and values for the cells in the first row of the table. Subsequent rows are retrieved by issuing more **GET-NEXT**s, passing as arguments the object identifiers obtained with the previous **GET-NEXT**. In this way the manager proceeds row by row through the table, until the last row is reached. When the manager sends a **GET-NEXT** with the object identifiers for the last row of the table, the response "spills over": where the manager expected a cell from the first column it gets the object identifier for the second column, where it expected a cell from the second column it gets the object identifier for the third column, and so on. (Where a cell from the last column was expected, the manager receives an object identifier for an element of the MIB unrelated to the table.) Thus it is easy for the manager to detect that it has retrieved the last row of the table.

Q: I'm confused about how a managed object instance reports its object class when it is behaving allomorphically. Suppose I have an instance of the class *betterPrinter* that is behaving allomorphically to the class *printer*. If this instance receives an **M-GET** operation requesting its *objectClass* attribute, where the value specified for *baseManagedObjectClass* in the **M-GET** is the allomorphic class *printer*, what value does the agent put in its response to the **M-GET** (1) in the CMIP field *managedObjectClass*, and (2) for the attribute *objectClass* in the *attributeList*?

⁷If a table is very wide, the manager might not be able to retrieve an entire row at a time, since the responses would exceed the size limit for SNMP PDUs. In this case the manager can use the technique described here to retrieve a part of a row, then another part, and so on until it has retrieved the entire row.

A: In the attribute list you must put the actual value of the *objectClass* attribute in this instance, i.e., *betterPrinter*, except in error cases where the class is not known. Just remember that allomorphism affects which attributes you report in a response to an **M-GET**, but never the values of those attributes. For the *managedObjectClass* field, the story is a little more complicated. For a *scoped* request, i.e., a request whose scope includes more than just the base object instance, CMIS requires that the *managedObjectClass* field be present and report the actual class of the instance that is responding. (Note that the standards provide no way for a manager to request allomorphic behaviour for a *scoped* operation, since the *baseManagedObjectClass* field applies to the instance at the root of the scoping tree, not to the instances falling within the specified scope.) For an unscoped request, CMIS says nothing definitive about *managedObjectClass*: an agent is free to omit it entirely, to report the actual class of the managed object, or to echo the allomorphic class from the request. My recommendation is to go ahead and report the instance's actual class in the unscoped case as well, since, other things being equal, it's much easier to design and implement an agent if special case processing for *scoped* operations vis-a-vis unscoped ones can be reduced.

Standards Activities and Status

Here are some items of interest in the world of Advanced Peer-to-Peer Networking (APPN) network management.

At the APPN Implementers Workshop (AIW), a conference recently hosted by IBM in Chapel Hill, North Carolina, there was a presentation describing an IBM prototype implementation that uses CMIP to provide monitoring and control of an APPN network configuration and to control and collect network accounting information. No product plans for the prototype have been announced. This workshop was the first held for APPN developers across the industry, and follow on workshops will be scheduled.

The next AIW will be held in conjunction with the APPC (Advanced Program-to-Program Communication) Platform Developers conference, which will be held in Research Triangle Park, North Carolina on June 21-24, 1993 at the Sheraton Imperial Hotel and Conference Center. The conference will have numerous sessions of interest to developers of both APPC and APPN platforms. Two sessions are of interest to the network management community:

APPN Management Today

This session will describe protocols and offerings for managing APPN via SNA/Management Services and SNMP. The presentation will extensively cover the current SNMP MIB for APPN and the management functions that it provides. Plans for future MIB extensions will also be discussed.

Working Session on SNMP MIBs for APPN and APPC

There is a large community of interest in developing industry standard SNMP MIBs for the management of SNA resources. Several companies in the AIW (including IBM) are active members of the SNA NAU Services MIB (snanau) working group of IETF that has recently begun the task of developing these industry-standard MIBs. The initial goals of the IETF working group are MIBs for PU 2 and T2.1 LEN nodes, and LU types 1, 2, and 3, with APPC and APPN MIBs to follow some time later. This AIW working session will explore how interested members of the workshop can enhance the efforts of the IETF working group. Our hope is that by working together both in the AIW and in the IETF, we can expedite the development and acceptance of industry-standard MIBs for APPN and APPC.

The agenda for the conference can be acquired from the same anonymous FTP server that **CMIP Run!** resides on:

ibmstandards.cary.ibm.com

The agenda file is called:

aiw/conference/pdc93.agenda

For conference information, contact:

Kay Sintal, PDC Conference Coordinator
Phone: (919) 254-4460
E-Mail: sintak@ralvm6.vnet.ibm.com
Fax: (919) 254-6050

For information by mail, write to:

APPC Market Enablement
Attn: Kay Sintal
PO Box 12195
RTP, NC 27709-2195

Mike Allen

Mike Allen is a member of APPN Architecture in RTP. His area of specialization is SNA management, with particular emphasis on the newer APPN architecture. He was part of the team that defined the base SNA/MS architecture for APPN nodes, and has more recently been involved in the creation of a managed object model for SNA resources. He currently represents IBM in an industry effort to define standardized SNMP MIBs for SNA resources.

General Newsletter Information

This newsletter is produced by a volunteer team in the Network & Systems Management Architecture & Standards area, within the IBM Networking Systems Architecture organization. This team acts as the editorial staff for this newsletter.

Contributions

Contributions and suggestions on the technical content, format, and scope of topics in this newsletter are welcome and encouraged, as are questions for MIP. All communication related to **CMIP Run!** can take place via any of the following vehicles:

To the VNET ID: *CMIPRUN* at *RALVM6*

To the Internet ID: *cmiprun@ralvm6.vnet.ibm.com*

By FAX to +1-919-254-5483, attention "**CMIP Run!**"

By mail to:

CMIP Run!
IBM Corporation
P.O. Box 12195
200 Silicon Drive
Department E78/Bldg. 673
Research Triangle Park, NC 27709
U.S.A.

Please note that there is no guarantee that contributions to **CMIP Run!** will be printed.