RODM: A control information base

by A. J. Finkel
S. B. Calo

Operational management of computers and computer networks was formerly performed exclusively by an operator or a team of operators equipped only with consoles for the display of status messages. Each system component independently determined its own set of such messages, identifying conditions needing attention. To meet future challenges, however, a structured approach to systems and network management and associated automation will be necessary. The amount and complexity of the status information needed for control and coordination will make it unlikely that operators will be able to keep up with such needs unaided. This control information must be made available to a family of systems and network management applications (including operator display programs). The NetView® Resource Object Data Manager (RODM) is designed to facilitate the storage and retrieval of control information. It provides services for defining a structured data model of a computer system. The control information is not kept simply in the form of messages, but instead the data are organized into units called objects. This allows the model to effectively capture interrelationships and dependencies as well as status information.

As computer systems and computer networks increase in size and complexity, their careful management becomes even more crucial to their user organizations. In the past, the operators of large computer systems and computer networks had to rely on a nearly manual approach to the management and control of their systems. Network and system configuration were often available only in hard-copy format, and the correlation of system problems with such configuration information was a labor-intensive task. This task might involve reading system messages from an operator console, listening to user complaints, and examining the system configuration to pinpoint the location of a problem.

Automation of the management and operation of information systems and computer networks is a natural goal. However, the present organization of these systems tends to make writing such automation applications difficult and cumbersome. The need for a unified approach to systems management has led IBM to develop an enterprise-wide structure for the integration of management functions. This structure is known as *SystemView**.[1]

SystemView is meant to provide a consistent environment for the development of systems and network management applications. The scope and complexity of the tasks involved in managing information processing systems are so great that numerous tools are needed to accomplish the tasks effectively. Different applications are typically used for: displaying information to operators or administrators, managing communications networks (e.g., alarm correlation, fault diagnosis), coordinating the use of host resources (e.g., job scheduling, storage assignment), maintaining system integrity (e.g., user registration, backup procedures), etc. This variety often leads to situations where multiple user interfaces need to be learned, the same data have to be entered multiple times and in different

formats, and multiple application environments have to be invoked manually and in certain sequences.

In order to ameliorate such difficulties, SystemView provides interfaces and common services that can be used to eliminate the differences in end-user support, standardize the format of data used by management applications, and normalize the specification of interactions and flows. These objectives are addressed by three structural elements called dimensions in SystemView: an end-use dimension, which provides presentation facilities and services; an application dimension, which defines guidelines for the implementation and integration of systems management applications; and a data dimension, which provides services and facilities for the standardization of data definitions and data access. SystemView also defines levels for structuring systems management tasks in terms of the major divisions of function that need to be addressed. The administration level includes functions that handle overall administration, planning, and policy setting for the enterprise. The coordination level deals with the control of the system resources—carrying out the set plans and policies, monitoring the activities of resources, and reporting relevant information to administration functions in support of longer-term considerations. The execution level deals with the specifics of running and monitoring the actual resources that provide the information services. Only execution-level applications should have to have knowledge of the detailed structure and idiosyncrasies of particular devices, thus allowing coordination-level applications to be more generic and more widely useful.

In this paper we are primarily concerned with the SystemView data dimension and the facilities that it provides. Two types of information support are defined in SystemView, and they are provided by two major data dimension components, the enterprise information base (EIB) and the control information base (CIB). The EIB is conceptually a single, logical database with an enterprise-wide scope, and it contains the data needed for planning and administration. In any given enterprise, there may be multiple CIBs, each associated with a particular management domain. A CIB contains operational data about information processing resources and whatever subset of the planning and definition information from the EIB is needed to support the management interactions between the applications in its domain and the resources that they are managing. The SystemView structure only specifies the characteristics of a CIB and an EIB. The actual architecture of these data dimension components is determined by the products that implement them.

All of the coordination and control applications need access to management information. This information may take the form of computer messages or the alarms and alerts that may be generated by a computer network. It may also take the form of data generated by a performance monitor such as the IBM Resource Measurement Facility (RMF). A characteristic of these data is that they are vital to the real-time management of the system and that they change rapidly in time.

Without the supporting information facilities, manipulating this control information is a very cumbersome task for an automation application. The desired information must be obtained by submitting queries and commands to the computer system or network or by monitoring message queues. The messages, alarms, and alerts of interest must then be processed. This processing includes the parsing of messages and the storing of messages in program variables. The application must also be concerned with such issues as the timeliness of the information that it is storing. Indeed, experience has shown that upwards of two-thirds of the effort involved in writing a systems or network management application has to do with the manipulation and management of control information.

The design and coding of systems and network management applications can be greatly simplified by providing a data manager to handle the repetitive data management tasks previously handled by *each* application on an ad hoc basis. This role is one of several that can be served by a control information base. Resource Object Data Manager (RODM) is a feature of NetView* Version 2 Release 3. RODM is a data manager that can provide the services of a control information base and can also be used to store execution-level information. RODM is the SystemView control information base for the Multiple Virtual Storage/Extended Architecture (MVS/XA*) and Multiple Virtual Storage/Enterprise Systems Architecture (MVS/ESA*) operating system platforms. In particular, RODM provides facilities so that:

- The architecture of a data model of a computer system or computer network may be designed and the model defined.
- The control and status information contained in computer messages, alarms, and alerts may be processed, and the appropriate parts of the model correspondingly updated.
- Applications may subscribe to be notified about changes to specific information, being informed of those changes to system status (and only those changes) in which they are interested.

The basic unit of data in RODM is an *object.* An object is an instance of a *class.* By means of an application programming interface (API), classes can be defined, objects can be created or destroyed, and information regarding objects can be queried or manipulated.

Active objects

Objects in RODM are said to be active in that they can maintain their own state. Data in conventional databases are usually quite passive, requiring an external application to specifically alter a data record in order for the value of that record to change. In RODM, in contrast, facilities are provided so that programs allowing an object to automatically maintain itself can be included in the object definition.

Many of the specific technical characteristics of RODM arise from the environment in which it was designed to operate and from the primary role that it was designed to fulfill. RODM was intended to provide a common representation service for the management of complex information processing networks and systems, thus establishing a focus of integration for automation applications. Conceptually, RODM maintains an active, object-oriented model of the system to be managed, and the automation applications deal exclusively with this model to effect management processes. Any real-world object within the system that needed to be managed would thus be represented as an object in RODM. The required behaviors of these objects would be captured in their associated methods, and relationships and configurations of objects would be maintained by object linkages.

The objects contained in RODM are therefore typically abstractions of real-world objects, such as tape drives, DASDs, modems, terminals, and control units. The goal is to allow applications to manipulate the RODM object as if it were the real object. For example, to query the status of a modem, an automation application might query the status field of the RODM object that corresponded to the modem. Programs stored with the object, called *methods*, ensure that the status field contains correct information. The application need have no contact with the actual modem. The methods associated with the object maintain contact with the real modem and ensure the validity of the data maintained in the RODM object.

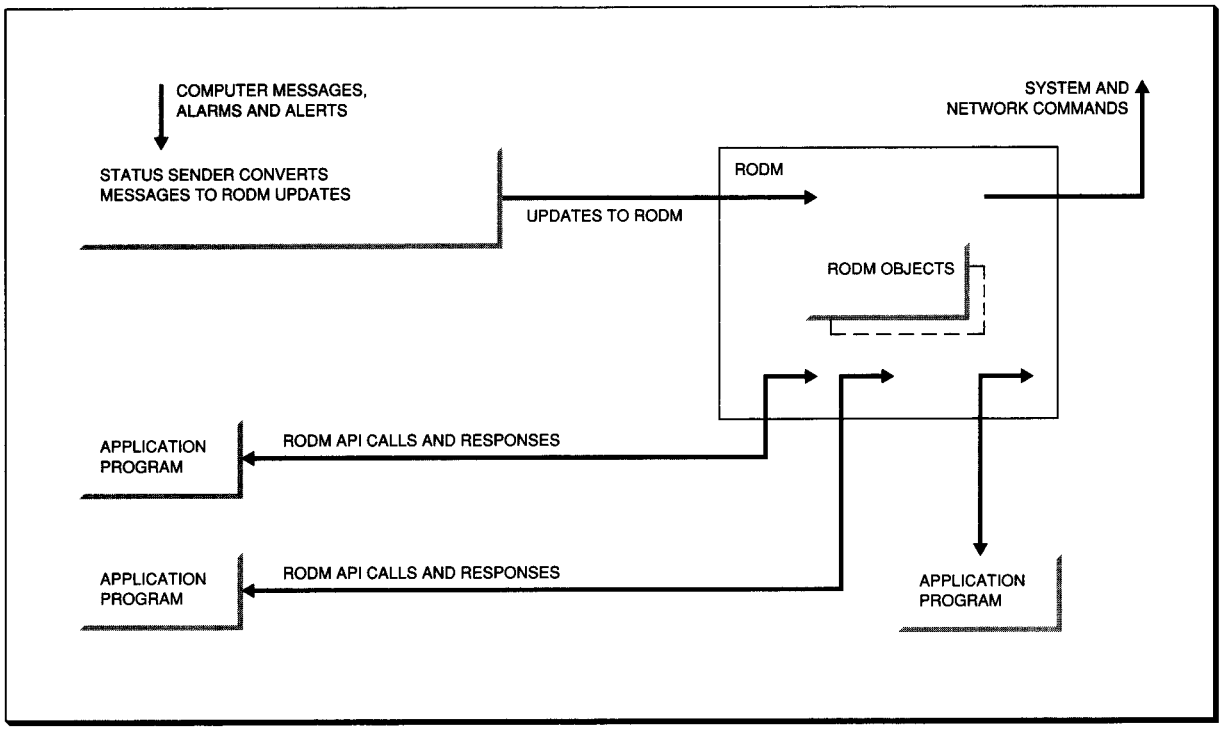
In a similar way, an application may change the status of the modem from on line to off line by issuing a request to change the value of the status field of the modem object to off line. Once again, a method associated with the RODM modem will interact with the physical modem to carry out the desired function.

A query method associated with a status field of a RODM object might perform the following processing when a query request is received:

1. Check the value of the field and determine if the data stored in that field are still valid. Some data are short-lived and may only be valid for a few minutes or seconds. If the data are still valid, the query method may terminate, and RODM will return the value of the field to the caller.
2. Interrogate the actual object to learn its status if the data are no longer valid. The query method can then set the field with its new value, and this value will then be returned to the caller.
3. Occasionally, the interrogation of an object may involve issuing a command that will return data asynchronously. In this case, the query method can install a notification that will inform the caller when valid data are returned. The query method may mark the field as pending and return an indication to the caller of what has happened.

The information needed for maintaining an accurate model of the real resources is typically contained in computer messages, alarms, and alerts. When commands are issued to physical devices, they usually respond with messages. Special applications called status senders must then be provided to recognize these messages, parse them,

Figure 1 Data flow for an active object



determine their relevancy, and use the RODM API to update the appropriate objects. (Alternatively, for classes of resources that do not formulate messages, but store status information privately, similar applications need to be provided to locate, interpret, and report such information in terms of RODM updates.)

The design of status sender applications and the construction of methods are the responsibility of the designer of the RODM object class. Automation applications need not be concerned with such details. The applications merely reference the RODM object in order to obtain the needed information. The situation is described in Figure 1.

The picture presented to the developer of an automation application is therefore simplified. Instead of being concerned with messages, commands, and the validity of data, the developer uses the RODM API to access data objects contained in a structured data model. To the developer, the data contained inside RODM is self-maintaining.

RODM functional design

RODM objects. RODM is an object-oriented data manager. Data are organized into units called *objects*. Encapsulated within an object is all of the information necessary for maintaining its defined behavior. This information takes the form of data or executable programs called *methods*.

Each object must have a unique name. An object is made up of fields. Each field within an object must also have a unique name. Fields may have certain subfields associated with them. One subfield that must always be present is the value subfield, which holds the data associated with the field. These data must be of a type supported by RODM. Some data types supported are: CharVar, Floating, Integer, and METHODSPEC. The data type METHODSPEC allows the value subfield to be set to specify a small program called a *named method*. API requests against this field can cause the identified method to be executed. Figure 2 shows the structure of a sample object using a PL/I-like syntax. The syntax used is only for illustrative purposes. No subfields are shown.

Figure 2 Structure of a RODM object

```
Declare

1  Printer3835                          ,
   2  MyPrimaryParentID       ClassID   ,/* Required Field  */
   2  MyPrimaryParentName     ShortName ,/* Required Field  */
   2  MyID                    ObjectID  ,/* Required Field  */
   2  MyName                  ObjectName,/* Required Field  */
   2  WhatIAm                 Integer   ,/* Required Field  */
                                         /*Optional Fields  */
                                         /*follow           */
   2  Model                   CharVar,   /* Model of Printer*/
   2  PaperType               CharVar,   /* Type of Paper   */
   2  AlterPaperType          CharVar,   /* Alternate Paper */
   2  Location                CharVar,
   2  Status                  CharVar;
```

Table 1 RODM subfield organization

Subfield Name	Required	System-Administered	Holds Methods or Values
VALUE	yes	no	values
CHANGE	no	no	methods
QUERY	no	no	methods
NOTIFY	no	no	methods and recipient information
TIMESTAMP	no	yes	values
PREV_VAL	no	yes	values

(Also note that the structure of an object is completely determined by the definition of the class to which it belongs.)

Several optional subfields may be associated with a field. The query subfield is used to specify a query method and has data type METHODSPEC. When the query subfield is present, the query method is invoked just before the contents of the value subfield are read and returned to the caller. The query method may examine the value subfield and alter its contents, or it may examine and set other fields in the object. This allows values to be calculated or updated appropriately before being returned to the caller.

A change subfield, if present, is used to specify a change method. The change method is invoked whenever an update to the value subfield is to be performed. This procedure allows calculations, normalizations, change tracking, and change propagation to be accommodated.

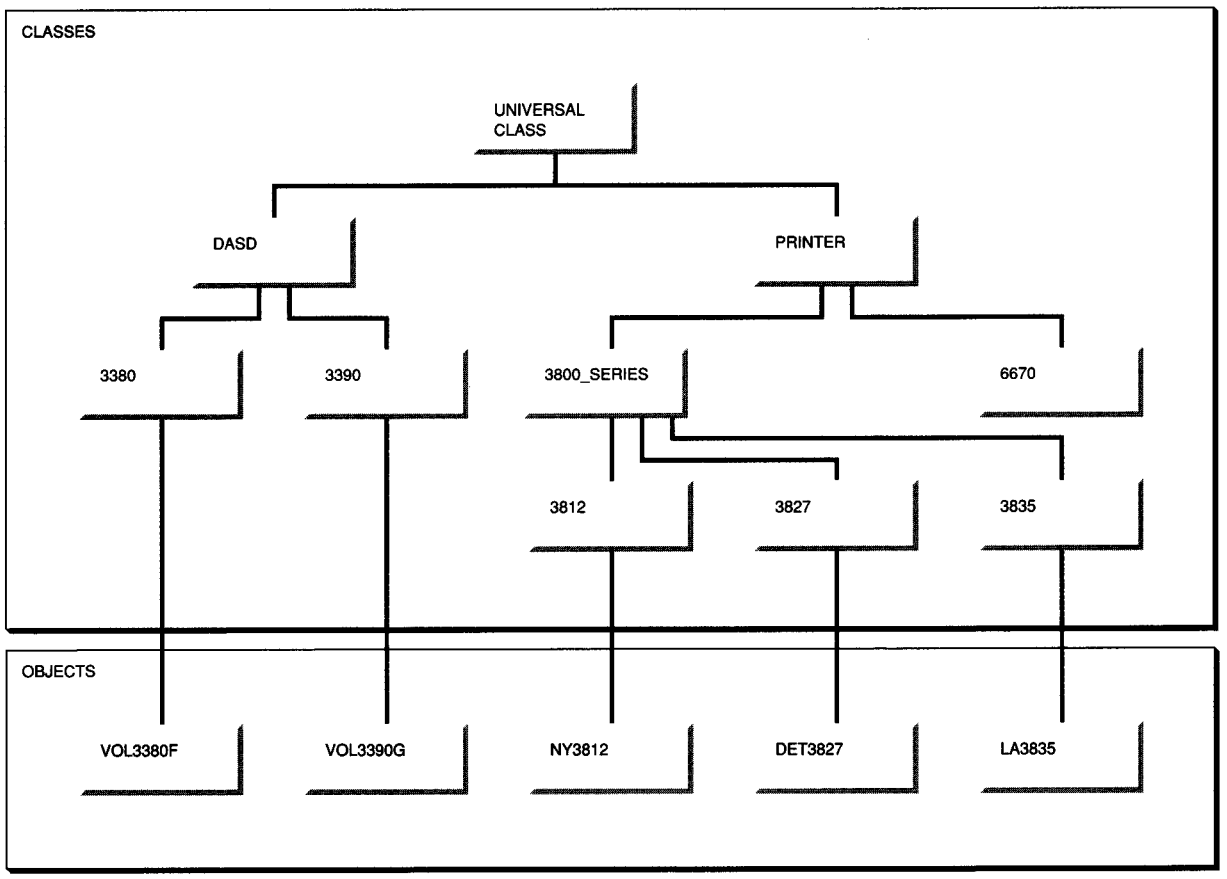
The notify subfield, if present, holds the identifiers of a list of methods called notification methods. These methods are intended to notify RODM users or other objects that certain changes have taken place to the field on which they are installed. Each method in the subfield is executed whenever the associated field is changed. These methods may perform certain tests (e.g., check threshold settings) before deciding to notify the indicated users or other objects.

Two other optional subfields may be present. The timestamp subfield is used to identify the time at which the value subfield last changed. It is set by RODM to the local system time whenever a change transaction is issued against the field. The prev_val subfield is used to hold the previous value of the value subfield and is also set by RODM whenever a change transaction is issued against the field.

Subfields can either hold values or method identifiers. A subfield is either automatically administered by the system or controlled through RODM API calls. The situation is summarized in Table 1.

Classes and inheritance. Each RODM object is a member of one and only one class. Classes may have many members, but each one is identical in structure. They have the same fields and field

Figure 3 RODM primary hierarchy



names. In this sense, the objects in a class are like a set of similar data records or rows of a relational database table.

The class to which an object belongs (or which an object is said to instantiate) is called its parent class. An object is said to *inherit* its structure from its parent class. In fact, the RODM API does not permit fields to be defined directly on objects. Fields are defined with respect to classes, and these definitions are inherited by objects from their parent classes.

Each class itself has one and only one parent (with but one exception, as noted below). This parent is another class. However, a class may have many children. In this way, the collection of RODM classes forms a tree-structured hierarchy called the *primary hierarchy*. At the root of this inheritance tree is a special class called the Universal Class, which has no parent. Figure 3 shows a part of a primary hierarchy.

Each class has a unique name and is described in terms of its fields. The fields of a class may be either private or public. Only the public fields are inherited by the children of the class. The private fields are not inherited and are typically used to hold aggregate information for the class. When a field is inherited, all of the subfields defined for that field are also inherited, and the object or class that inherits the field may not add additional subfields.

In the same way that an object inherits public fields from its parent, a class also inherits public fields from its parent class. The inheritance of these fields therefore percolates down the primary hierarchy. A class may have many children, but in the version of RODM supplied with NetView

Table 2 Structure Inheritance

Class or Object Name	Locally Defined Fields	Inherited Fields
DASD	State Control_Unit Peers_in_String OwnerID	
3380	Seeks_in_Last_Hour Percent_of_Capacity Date_of_Last_Service	State Control_Unit Peers_in_String OwnerID
3390	Date_of_Installation Number_of_Failures	State Control_Unit Peers_in_String OwnerID
VOL3380F		State Control_Unit Peers_in_String OwnerID Seeks_in_Last_Hour Percent_of_Capacity Date_of_Last_Service

2.3, these must be either all classes, or all objects, i.e., no class may be the parent of both objects and other classes.

The inheritance of field structure is mainly intended as a reusability mechanism. Objects of similar structure need be defined only once, at the class level. Classes may be further refined so that structure can be shared. For example, in Figure 3, class 3800_SERIES inherits all of the public fields of class PRINTER. Class 3800_SERIES may also define its own fields. Similarly, class 3827 inherits all of the public fields of class 3800_SERIES and may also define its own fields. Table 2 explicitly demonstrates how structure inheritance works for another part of the primary hierarchy in Figure 3.

In that example, the class DASD defines four fields. The structure of these four fields is inherited by the classes 3380 and 3390. Each of these classes defines some of its own fields. Finally, the object VOL3380F inherits its field structure entirely from its parent class 3380.

Just as structure may be inherited by classes and objects, so may the values of subfields. Value inheritance is dynamic and allows for an even greater level of information sharing. If an object or class inherits a field from its parent, the default behavior is for it to inherit the data value of the value subfield as well, along with the contents of the change and/or query subfields, if either or both of them are present. However, inheritance may be overridden by explicitly assigning a new value to the appropriate subfield. Value inheritance will remain overridden until a request is made to RODM to restore default inheritance.

Value inheritance is intended to allow default values to be shared among objects and classes. For example, a change method that captures the behavior of an object when a particular field is updated would typically be the same for all objects of a given class and would rarely change. It can be specified once at the class level and inherited by all objects that need to use the method. In special cases, inheritance may be overridden, and a special change method may be assigned to a particular object that is meant to have a unique personality.

Value inheritance is maintained dynamically. If an object or class inherits the value of a subfield from its parent, and a change is made to the value of that subfield at the parent, the change is immediately propagated down to the child. This feature allows the definitions of methods to be changed once at the class level and the change in behavior to become immediately associated with all objects. Such default value propagation could have significant performance costs if used excessively. It is assumed, however, that the class hierarchy is predominantly definitional in nature and thus does not often change. It defines the structure of the objects and the default values to be used for their inherited characteristics.

Whereas changes to the values of subfields at the class level are assumed to be relatively infrequent, changes to the values of subfields at the object level are expected to be quite frequent. Performance expectations for these changes are consequently much higher. A general rule of thumb is that any change or query of a subfield should take less time the lower in the primary hierarchy it occurs.

Value inheritance for the notify subfield is slightly more complicated than for other subfields. An object inherits the value of the notify subfield of its

parent and may also add its own local notification methods to this list. Thus, when notification methods are triggered, all of the methods locally defined on the subfield are triggered first; then all of the methods associated with the corresponding notify subfield at the parent class are also triggered. This sequence occurs because of the semantics of the notification operation. Different users may be interested in activities at different levels of the hierarchy. A storage assignment application may subscribe to notifications against fields of the object VOL3380F, whereas a storage capacity planning application may be interested in tracking its parent class 3380. The local notifications cannot be interpreted as overrides to the more global ones.

Each RODM object or class must have certain required fields. Figure 2 shows some of the required fields for an object. They include MyPrimaryParentID, MyPrimaryParentName, MyID, MyName, and WhatIAm. These fields may be thought of as having been defined at the Universal Class level of the hierarchy. They are read-only, and value inheritance is always overridden. In addition, classes are required to have both a MyClassChildren field and a MyObjectChildren field for which values are also not inherited. Even structural inheritance works slightly differently for these fields than as described above, in that their existence is not inherited by objects.

Object naming. Object naming is a difficult issue in object-oriented systems. RODM takes the following approach. Each object has a unique character string name that may be 1 to 254 characters in length. Each class has a unique character string name that may be 1 to 64 characters in length. An object or class name must begin with a letter of the alphabet or a positive integer.

A class is uniquely identified by specifying its class name. Each class also has a corresponding ClassID that can be used to uniquely locate the class. The ClassID is calculated by RODM and is returned as a result of any API call that manipulates or examines the class. Applications can then subsequently use the ClassID rather than the class name in API calls, generally resulting in improved performance.

An object may also be uniquely located by specifying its ObjectID. As with the ClassID, the ObjectID of an object is calculated by RODM and is returned after any API call that manipulates or examines the object. In general, uniquely specifying an object in RODM is slightly more complicated than specifying a class. An object is identified by its ObjectID, or the object name and the name of the object's parent class, or the object name and the ClassID of the object's parent class.

Besides having a name, each field in an object or class has a FieldID. The FieldID is obtained through certain API calls and may be specified in some API calls. Any two fields that have the same name have the same FieldID, even if the fields are in different objects or classes. A field of an object is referenced by supplying either a FieldID or a field name along with the object location information described above. A field of an object can generally be located more quickly if a FieldID is supplied instead of a field name.

Objects in the same class share the same field names. The FieldIDs of these fields are also the same. When manipulating fields from several objects in the same class, the FieldID need only be obtained once. The FieldID can then be supplied on subsequent API calls that reference different objects in that class.

The RODM application programming interface. RODM provides an application programming interface (API) that allows application programs written in the C/370 or PL/I programming languages to request services. Many such applications may concurrently access RODM. API calls are generally synchronous, i.e., control is not returned to the caller until the API request is complete. The API calls are divided into several basic categories.

Not all RODM API services are available to every user. Access to RODM API services is based on a set of authority levels. A RODM user may be defined to be a member of one of six authority levels. Higher authority levels give a user increased capabilities. The IBM Resource Access Control Facility (RACF) may be used to administer these authority levels. We discuss a few of the RODM API categories below.

Create. The create services allow objects and classes to be created and fields of classes to be defined. Creation of a class involves schema definition. It requires a higher level of authority than is needed to create an object.

As previously noted, fields may not be created for objects. An object inherits its fields from its parent class. A new field may be added to an already existing class, in which case all of the children of the class immediately inherit the new field, along with the default values assigned to its subfields.

The create service is dynamic in nature. New objects and classes may be created without reloading the entire information base. Newly created objects and classes are available to all appropriately authorized RODM users immediately after their creation. A checkpoint is necessary to permanently save changes to disk.

Delete. The delete service allows objects and classes to be deleted. Deleting a class requires higher authority than deleting an object.

Query. The contents of a field, or of a particular subfield of a field, of an object or class may be queried. When a subfield is queried, no methods are invoked, and the contents of the field are returned to the caller.

When a field is queried, a query method is invoked, if present, before the contents of the value subfield are returned to the caller. The term *query a field* is often used to represent this action.

Change. The contents of a field, or of the value, query, or change subfields of a field, may be set to a certain value. When a change request is made against a field, a change method, if present, is invoked. It is the responsibility of the change method to perform the requested change (although the method is not required to do so). If no change method is present, RODM changes the value of the value subfield to the requested new value.

No methods are invoked for change requests against subfields, even the value subfield. Changing the value of the query or change subfield has the effect of specifying a new query method or change method.

Action. The RODM API provides a service for triggering a named method. Such methods can be used to implement actions unique to an object or class, and are contained in fields of type METHODSPEC. For example, a named method could be constructed and associated with class DASD of Figure 3. This method could contain instructions for formatting the direct access storage device (DASD). Through inheritance it could be contained in a field of the object VOL3380F. An action request against this field on the object VOL3380F would then trigger a process that caused the actual DASD to be formatted.

Notify. As discussed earlier, an application may ask that a notification method be installed on the notify subfield of a field. When the value of the field changes, the notification method will be invoked and may, depending on its internal logic, notify the subscriber of the change.

Connect and disconnect. Before making any other requests to RODM, an application must first start a session with RODM by making a connect request. To terminate such a session, a disconnect request must be issued.

Link and unlink. Although objects may be thought of as encapsulating information, it is sometimes necessary to explicitly capture relationships between different objects. A link request allows an association between two different objects to be defined. An unlink terminates such an association.

Link operations may be used to capture such things as the configuration of a computer network (by linking connected devices) or the composition of a complex device (by linking the component objects).

RODM methods. RODM supports two general kinds of methods: object-dependent methods and object-independent methods.

An object-dependent method may access only the one RODM object or class with which it is associated. The method has exclusive access to that object or class while it is executing. As described earlier, RODM supports several specific types of object-dependent methods:

- Query methods
- Change methods
- Notification methods
- Named methods

These types of methods have been discussed previously in the first two subsections of this section.

An object-independent method may gain exclusive access to several RODM objects at once. Such a method can then query or change fields in any

of the objects held. Object-independent methods are meant to be used only when atomic changes must be made to several objects at once. An object-independent method is like a callable subroutine that runs inside of RODM. It can be directly invoked by a call to the RODM API.

Writing methods with the method API. The tasks of defining classes and the methods that specify the behavior of objects would generally fall on the RODM modeler or the RODM administrator. Many applications that use RODM will also need to provide their own definitions of classes and methods, but it is felt that typically the construction of the methods themselves will be done by a systems programmer, and not by the writers of automation applications. However, many applications supplied by IBM, or by vendors such as the SystemView International Alliance Partners, will provide their own set of methods for RODM. The NetView Graphic Monitor Facility[2] is an example of such an application. In these cases, there will be little need for customer personnel to supply methods. Some customization may be necessary.

Methods can be written in the PL/I or C/370 programming languages.

Methods gain access to RODM services via an application programming interface called the method API. The method API provides many of the same services that the RODM API provides. The services available to object-dependent methods and object-independent methods vary slightly.

Object-dependent methods may issue query and change requests against any field, or subfield of a field, of the object or class with which they are associated. These methods may also install notification subscriptions and trigger named methods and object-independent methods. In addition, object-independent methods may link and unlink objects and create and delete objects.

RODM architecture on MVS/ESA and MVS/XA

RODM runs as a subsystem on both MVS/XA and MVS/ESA. An application that uses RODM may run in its own address space. (This could be the address space of the NetView automation platform, a Time Sharing Option [TSO] address space, or a batch address space.) To communicate with RODM, an application must first issue a connect request. After a successful connection, the RODM API would be used to invoke the desired services. For local applications and users, communication with the RODM address space takes place via the PC, or Program Call, instruction. This instruction is a System/370* and System/390* hardware instruction. Program Call and an associated instruction, Program Return (PR), in effect provide a hardware-based remote procedure call facility. Although the facility is not compatible with most other remote procedure call implementations, it has the advantage of being extremely fast.

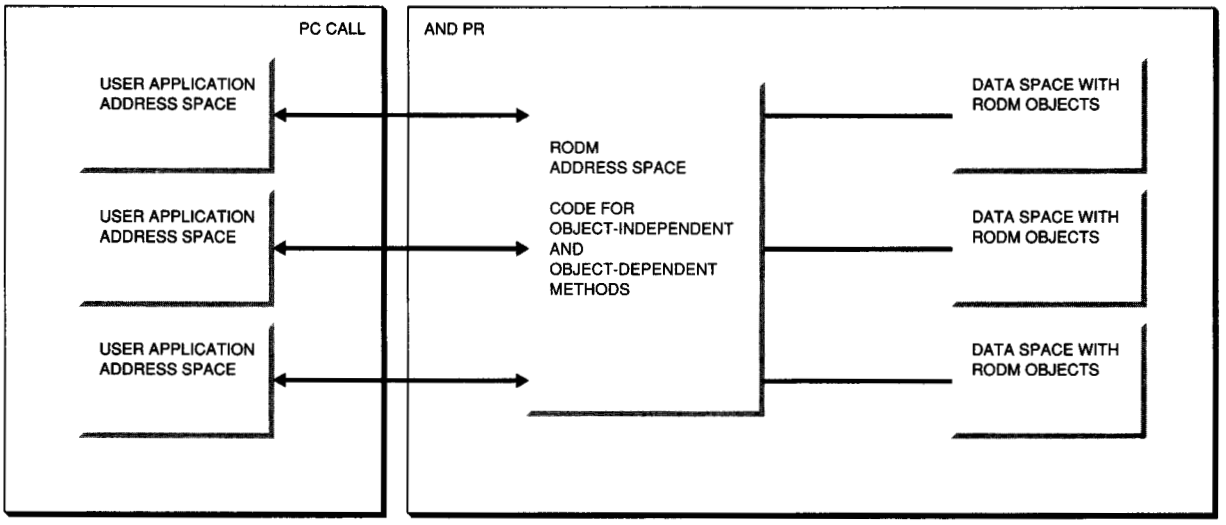
On MVS/ESA, RODM objects and classes are stored in data spaces. These data spaces are accessible only from the RODM address space. On MVS/XA, RODM objects and classes are stored only in the RODM address space. The available amount of storage is therefore more constrained on MVS/XA. Figure 4 presents an overview of the basic RODM architecture on MVS/ESA. Note that both object-independent and object-dependent methods execute inside the RODM address space.

RODM is intended to store both operational data and control information. Fields in objects contain information about the status of the resources that the objects represent. Applications query these fields to obtain information about system status. This status information is highly dynamic in nature. Therefore, RODM does not attempt to commit all changes to DASD as they are made. The information may well change soon after it has been saved (or even while it is being written).

Instead, RODM keeps these changes in memory. An authorized user can request that a snapshot of all RODM objects and classes be committed to DASD. This request is called a checkpoint request, and it is anticipated that such checkpoints will be taken as part of normal backup procedures. They would also normally be taken anytime that a substantial number of new objects have been created, such as after a cold start. The cold start process involves using the RODM API to create and define the class structure and primary hierarchy and then to define the object instances that describe the system. A checkpoint commits these changes to DASD.

Once the class definitions and object instances have been committed to DASD via a checkpoint,

Figure 4 RODM architecture on MVS/ESA



RODM may be warm-started. Warm start is a fairly quick process and will be the usual method for starting RODM.

Setting the stage: Preparing to write RODM applications

The RODM information manager is initially (at cold start) devoid of any data organization or data. Classes have not yet been defined and object instances have not yet been created. In order for an application to use RODM effectively, an information model must be designed and defined to the system. Typically, the development of such a model is the responsibility of systems administrators and systems programmers rather than the developers of automation applications.

The steps necessary to make RODM an effective tool for systems management are as follows:

1. Define classes—Definitions for RODM classes must be either supplied by a vendor or developed by supporting staff at the installation. If the classes are defined by a vendor, the installation would still typically have the option of modifying these definitions, usually by adding extra fields. These extra fields would model items of unique local interest.

   For systems and network management, class definitions for such devices as modems, encryptors, DASD, printers, and CSUs will most likely be included in the data model.

2. Develop methods—Change methods, query methods, named methods, and notification methods must be written in support of the model defined in Item 1. If the data model is supplied by a vendor, one can expect the supporting methods to be supplied as well. In this case, an installation may choose to customize the model by replacing or modifying some methods.

   The necessary methods must also be developed for any installation-defined fields or classes. RODM will supply a starter set of simple methods along with the data manager.

3. Write a status sender—A status sender converts computer messages, alarms, and alerts to RODM updates. If a vendor-supplied data model is used, the installation can expect a status sender application to be supplied. The installation may have to provide additional customization.

4. Define object instances—The installation must decide which hardware and software devices should be represented as RODM objects. These objects must be given RODM names. The installation must also describe the network and system configuration in terms of links between objects.

It may be possible to generate some or all of the RODM objects by using tools that automatically read system configuration files.

5. Generate the RODM data model—A cold start may be used to create classes, define fields, instantiate objects, install methods, and assign initial values to the fields and subfields of objects and classes. After the cold start, a checkpoint should be taken to save the data model.

   During a cold start, a special method called an initialization method is invoked. This method may read object and class definitions from a file and use the RODM method API to create the data model. The initialization method may be supplied by a vendor or written by the installation. In fact, IBM supplies a load utility with RODM. The utility may be invoked from an initialization method. The load utility generates class definitions and object instances from a load file. Entries in the load file follow a simple, well-defined syntax.

6. Write automation applications—The model and supporting services can then be used by application programmers for automating management tasks. The status of system resources of interest can be queried and changed by means of the RODM API.

Historical motivations

The characteristics of RODM and the corresponding roles that it is meant to play in the management of complex systems are based upon experience with automation and the evolution of systems requirements. As computer systems and computer networks grew in size and complexity, the need for automated control over information processing resources became more and more apparent.

Since computers and networks were designed to communicate with operators by means of text messages, many automation tools were developed to aid in message processing. The MVS Message Processing Facility (MPF), for example, incorporates a number of useful functions: message display at the operator console can be suppressed, different highlighting options can be used, and selective messages may be marked as eligible for automation processing. MPF can significantly reduce message traffic at a computer operator's terminal and allow an operator to more

Figure 5 Sample operational procedure

```
IF   there are too many jobs queued for
     the 3800 printer

        and

     the workload of the 3835 printer is
     light,

THEN transfer some printer jobs from
     the 3800 printer to the 3835
```

effectively monitor and control a computer system. However, it only deals with the handling of messages and does not address the creation of the automation applications themselves.

The Network Communications and Control Facility (NCCF) goes somewhat farther and allows users to write CLISTs, i.e., command lists (see Reference 3 for information on CLISTs in NetView), in response to computer messages. A systems programmer can specify that a particular CLIST program be executed whenever a particular type of message is encountered. The program can then parse the message and automatically enter appropriate system commands in response to the indicated situation. This procedure allows simple responses to well-defined, immediate conditions to be easily programmed. However, the automation capabilities remain quite limited. Each CLIST program executes in its own self-contained environment, and it is difficult to share information between CLIST programs executing in response to different messages. Furthermore, the mechanism for invoking CLISTs has no memory. Therefore, dependencies between messages in space and time cannot be taken into account.

Computer operators monitoring the status of a complex system make decisions that are fairly contextual. Interviews with computer operators found that they tended to think in terms of situation response rules instead of just computer messages (Figure 5).

In an attempt to exploit the seemingly rule-based nature of operational policy, an experimental expert system tool, YES/MVS,[4–6] was developed at IBM Research. YES/MVS used rule-based languages (OPS5[7] and IBM KnowledgeTool*[8] and its

prototype, YES/L1,[9] to encode operational policy. A YES/MVS rule could be thought of as an IF-THEN statement, much like the example in Figure 5. The left side of the rule contained conditions which, if satisfied, made the right side of the rule eligible for execution. The right side of the rule could then issue system commands to execute the indicated policy. Whereas CLIST programs were triggered only by the appearance of particular messages, expert system rules allowed more complicated situations to be detected.

As one would expect, the left sides of rules made extensive references to computer system status. Services thus had to be built to query the computer system to obtain status information, translate system messages and present the information in terms of program variables accessible to the expert system, and check the validity of data being maintained in the program variables. It became apparent that a separate facility should be developed to provide these services, and that facility was called a model manager. In such an environment, rules could be encoded that made direct reference to system status without regard to how or when the information was obtained. The model manager would ensure the validity of the status information in the left side of the rule.

Another aspect of importance in support of automation applications is data sharing. Early versions of YES/MVS provided little data sharing between problem areas, thus leading to much duplication of information and effort. Later versions attempted to correct this fault by providing for a data model that could be shared among all problem areas. However, this data model was accessible only to computer programs coded in the expert system shell KnowledgeTool and could not support concurrent users. It lacked permanence and thus had to be rebuilt each time the expert system was initialized, and it was not dynamic, so that it could not be changed without bringing down the expert system.

In summary, then, experience has shown that a common representation service was needed that provided:

- Dynamic data modeling capabilities
- Active data management
- Concurrent data sharing
- Permanence

These considerations have been of key importance in establishing the desired characteristics of RODM.

Finally, we note that NetView now provides substantial automation facilities. It incorporates a message table that can be used to trigger REXX, PL/I, or C/370[10] programs. KnowledgeTool programs can also be written (see Reference 11, for example). Services are provided for parsing messages, sharing data among REXX programs, and routing messages among already active programs. The RODM data model in effect provides a set of global shared variables for these programs and should enhance the power of these languages as automation tools.

RODM design issues

As just discussed in the previous section, a model management facility is needed in order to effectively support automation applications. Accordingly, RODM attempts to fill that need by providing:

1. Data modeling capabilities—RODM allows a data administrator or systems programmer to design and develop the structure of a data model describing the objects of interest to the application programmers. The structure of these objects is captured in terms of a defined class schema.
2. Data management services—The incorporation of methods into the information model allows objects in RODM to manage their own state and maintain consistency with the state of the real system objects that they represent.
3. Data sharing—The RODM data model can be shared among many users and applications. RODM services are accessible to programs written in high-level languages such as PL/I and C/370. Many simultaneous sessions with other address spaces can be supported. To the programmer, RODM objects can be viewed as shared program variables that are permanent in the sense that they do not lose their values when the application program terminates.
4. Permanence—The data model can be permanently stored on DASD and need not be rebuilt each time RODM is started. This permanence is accomplished by means of a checkpoint and warm-start capability.
5. Dynamic schema evolution—The object schema will change over time, so RODM permits new class definitions, the addition of fields to classes

Figure 6 An MVS/JES3 systems response to the '8I S' query

```
IAT5638 F=000,W=000,A=000,U=000,V=000,E=000,B=000,R=000,AL=A,SCUR=000
IAT5638 MVSA      ONLINE    IPLD SMAX=255 SCUR=000 DA=334,000 TA=019,013
IAT5638 MVSC      ONLINE    IPLD SMAX=255 SCUR=000 DA=334,000 TA=019,010
```

with automatic propagation of the new fields to existing children, and the creation and deletion of objects. All such changes can occur dynamically, without bringing down the data manager.

6. Support for concurrent users—Numerous automation activities may take place at the same time. Each activity may naturally be viewed as a different task. NetView automation allows different operator tasks, and RODM supports concurrent users with relatively high performance.

Transactions against RODM will mostly take the form of change or query requests against fields in objects. Although RODM allows for fine grain data sharing, access to a single object may be serialized. Since concurrent requests to access fields in objects should be somewhat randomized, this scheme should support a high degree of concurrency.

Enabling automation applications

A data model. RODM provides interfaces to the application programmer that permit automation applications to be written more efficiently. A well-designed data model plays a crucial role. To the programmer, the data model can be viewed as a set of shared program variables accessible through an API. The sharing of these variables is managed by RODM. Such a data model should also be extensible and self-maintaining.

Applications that use RODM will be written by IBM, other vendors, and customers. RODM was designed with the intention of allowing these application providers to extend and customize the data model. In particular, application providers may have to extend RODM class definitions and provide additional class definitions. Importantly, this can be done dynamically, without bringing RODM down for a cold start.

A vendor that supplies a modem, for example, might provide the class definition and methods necessary to support that modem. An application provider might need to define additional fields and methods for those fields. The class definition can be extended without interfering with other applications that access the class or its children. At the same time, the new fields can be shared by any additional applications that have knowledge of them.

For example, Gottschalk[2] describes how NetView Version 2 Release 3 provides a data model to exploit the capabilities of RODM in managing complex systems and networks. Customers who use the NetView Version 2 Release 3 data model as a basis for automation are also able to graphically display the configuration and status of resources modeled in RODM by means of the NetView Graphic Monitor Facility.

Breaching the message-command interface. Historically computer operators and automation applications have interacted with a computer system through a message-command interface. Figure 6 gives an example of such an interface. An operator sitting at a console is able to submit commands and queries to the computer system or computer network. Responses are received in the form of messages. In addition, many unsolicited messages are also received. More recently, interfaces for automation applications to submit commands and receive messages in the form of byte strings have been provided.

A look at Figure 6 shows some of the difficulties automation programs have in dealing directly with message-command interfaces. To be usable by an automation program, messages must first be parsed. Even if they are identified by message number, the format of the text may vary. Also, as indicated by the strings MVSA and MVSC in Figure

Figure 7 Using objects instead of messages

```
 Declare

1 JES_SETUP_SUMMARY                ,
                                     /*Fields indicate   */
                                     /*number of jobs in:*/
  2 Fetch_Queue           Integer , /* MDS Fetch Queue  */
  2 WaitVol_Queue         Integer , /* WAITVOL Queue    */
  2 Allocate_Queue        Integer , /* Allocate Queue   */
  2 Unavailable_Queue     Integer ,
  2 Verify_Queue          Integer ,
  2 Error_Queue           Integer ,
  2 Deallocated           Integer ,
  2 Restart_Queue         Integer ,
  2 Allocate_Mode         CharVar , /*A for automatic   */
                                    /*allocation mode   */
                                    /*M for manual mode */
  2 Setup_Total           Integer   /*Total number of   */
                                    /*jobs in JES setup */
```

6, processing of computer system messages requires knowledge of system configuration.

In addition, the application must possess a detailed knowledge of the system command structure so that the correct command can be used to elicit the desired information. Temporal issues are also important. Messages sent in response to a query or command are generally received asynchronously and may not be time-concurrent or in any particular order. The application must take such considerations into account.

Finally, the application must make judgments about data validity, i.e., it must judge when to submit queries for new information. The lifetime of data contained in a computer system message is limited.

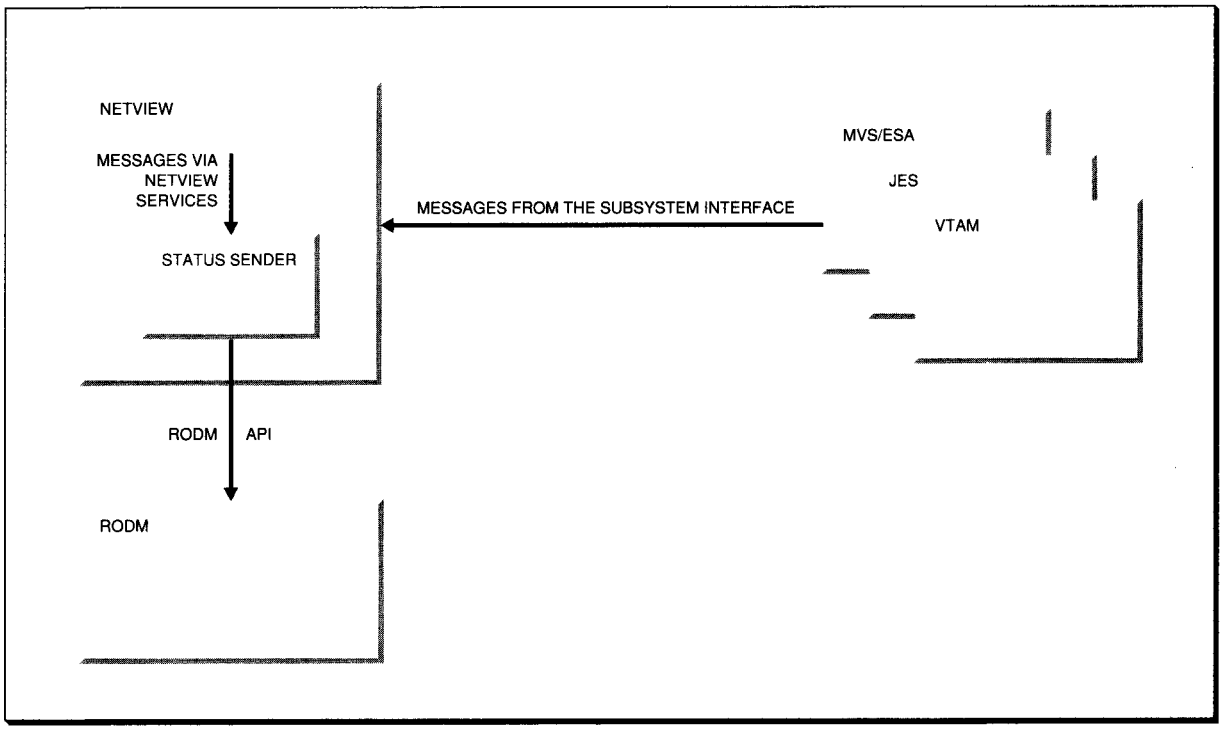
Automation applications that use RODM interact only with the system data model through the RODM API, rather than directly with the computer system itself through a message-command interface. Thus, much of the data maintenance functionality can be supplied in common.

Figure 7 illustrates a declaration of an object, JES_SETUP_SUMMARY, that models the information contained in the IAT5638 message in Figure 6 (note that required fields are not shown in Figure 7 for simplicity). Change and query methods can be used to keep the information in the fields of JES_SETUP_SUMMARY up to date. The writer of an automation application can treat JES_SETUP_SUMMARY as a self-maintaining data structure.

Building a status sender. A status sender application provides the bridge between the message-command interface and RODM. Such applications would normally be provided by the architects of the data model. A status sender has the responsibility of sending updates about the changes in status of relevant resources to the RODM objects that represent those resources.

Figure 8 shows how a status sender could be designed to interact with MVS/ESA and some of its subsystems. (VTAM*, the Virtual Telecommunications Access Method, is an MVS subsystem that manages telecommunications, and JES, the Job Entry Subsystem, is an MVS subsystem that manages batch jobs. Two versions of JES are available with MVS: JES2 and JES3.) The status sender runs as a task in the NetView address space. It uses NetView services to receive messages from MVS/ESA and some of its subsystems. The application parses and interprets these messages and uses the RODM API change requests to update the RODM data model.

Figure 8 A status sender for messages from MVS/ESA and some subsystems



Conclusion

Although the relational model[12] proved highly successful for modeling the passive characteristics of data objects, the nature of computer and network management applications required new information management techniques.

RODM provides facilities for the management of active objects, which can be used to model at a more abstract level the real-world objects that comprise a system. Through the RODM API, systems and network management applications may define, reference, and share a structured data model of the objects typically found in computer systems and communications networks. By means of defined methods, RODM objects can be made self-maintaining.

It is important to note that, although most of the discussion and many of the examples given illustrate how RODM can be used to monitor and control local devices, the same kinds of techniques can be applied more globally to effectively model many of the logical constructs needed for the management of complex systems. Concepts like *job*, *subsystem*, *queue*, or *software package* can be conveniently described in terms of attributes (fields) and associated operations (methods). RODM can thus provide representation services and active model management for very complicated, real-time, automation processes. Additionally, although it is implemented on MVS, it is not limited to dealing solely with MVS systems. As long as the appropriate status sender applications exist to forward relevant information to RODM, it can deal with objects describing any particular system, or even a collection of heterogeneous systems.

Other databases have used objects as their basic conceptual units, including: Orion,[13] Iris,[14] GemStone,[15] and OZ+.[16] Such databases were typically constructed to provide general programming support, however, and were not designed to actively model the rapidly changing real-world objects found in computer systems and computer networks.

In contrast, the nature of systems automation applications and the real-world objects that they

need to reference has greatly influenced the characteristics of RODM. RODM applications will not typically lock and manipulate a large portion of a database as private data, but will submit a large number of transactions at the granularity of the modeled objects. These objects are meant to be shared among applications and to be updated in real time to reflect system status. RODM thus provides for a high degree of data sharing and can support a relatively large number of concurrent transactions.

In short, the NetView Resource Object Data Manager is a general facility that supports the maintenance of control information and, as such, provides a set of essential services for management applications.

Acknowledgments

Many people besides the authors contributed to the design and implementation of RODM. Keith Milliken and Jack Hackenson made noteworthy contributions. Charlotte DiLeonardo provided essential help. Phil Kalinowsky developed an early prototype. The efforts of Eugene Conroy, John Farrell, David Feiner, Peter Ruppert, Ling Tai, Norman Waite, Jen Wang, Tammy Wu, Jeffrey Yeh, and Heather Albright are also acknowledged.

Cited references

1. *An Introduction to SystemView*, SC23-0576, IBM Corporation; available through IBM branch offices.
2. K. D. Gottschalk, "NetView Version 2 Release 3 Graphic Monitor Facility: Network Management Graphics Support for the 1990s," *IBM Systems Journal* **31**, No. 2, 223–251 (1992, this issue).
3. *NetView Customization: Writing Command Lists*, SC31-6015, IBM Corporation; available through IBM branch offices.
4. R. Ennis, J. Griesmer, S. Hong, M. Karnaugh, J. Kastner, D. Klein, K. Milliken, M. Schor, and H. Van Woerkom, "A Continuous Real-Time Expert System for Computer Operations," *IBM Journal of Research and Development* **30**, No. 1, 14–28 (1986).
5. J. Griesmer, S. Hong, M. Karnaugh, J. Kastner, M. Schor, R. Ennis, D. Klein, K. Milliken, and H. Van Woerkom, "YES/MVS: A Continuous Real Time Expert System," *Proceedings of AAAI 1984* (1984), pp. 130–175.
6. K. R. Milliken, A. V. Cruise, R. L. Ennis, A. J. Finkel, J. L. Hellerstein, D. J. Loeb, D. A. Klein, M. J. Masullo, H. M. Van Woerkom, and N. B. Waite, "YES/MVS and the Automation of Operations for Large Computer Complexes," *IBM Systems Journal* **25**, No. 2, 159–180 (1986).
7. C. Forgy, *OPS5 User's Manual*, CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA (1981).
8. *KnowledgeTool General Information*, GH20-925987, IBM Corporation; available through IBM branch offices.
9. A. Cruise, R. Ennis, A. Finkel, J. Hellerstein, D. Klein, D. Loeb, M. Masullo, K. Milliken, H. Van Woerkom, and N. Waite, "YES/L1: Integrating Rule-Based, Procedural, and Real-Time Programming for Industrial Applications," *Proceedings of Third Conference on Artificial Intelligence Applications* (1987), pp. 134–139.
10. *NetView Customization: Using PL/I and C*, SC31-6037, IBM Corporation; available through IBM branch offices.
11. *Expert System Prototype for Automated Console Operations Using NetView R3 and KnowledgeTool V2*, GG24-3450, IBM Corporation; available through IBM branch offices.
12. E. Codd, "A Relational Model for Large Shared Data Banks," *Communications of the ACM* **13**, No. 6, 377–387 (June 1970).
13. E. Kim, N. Ballou, H. T. Chou, J. Garza, and D. Woelk, "Features of the Orion Object-Oriented Database System," *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. Lochovsky, Editors, ACM Press, Addison-Wesley Publishing Company, Reading, MA (1989), pp. 251–282.
14. D. Fishman, J. Annevelink, D. Beech, E. Chow, T. Connors, J. W. Davis, W. Hasan, C. G. Hoch, W. Kent, S. Leichner, P. Lyngback, B. Mahbod, M. A. Neimat, T. Risch, M. C. Shan, and W. K. Wilkinson, "Overview of the Iris DBMS," *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. Lochovsky, Editors, ACM Press, Addison-Wesley Publishing Company, Reading, MA (1989), pp. 219–250.
15. R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E. H. Williams, and M. Williams, "The GemStone Data Management System," *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. Lochovsky, Editors, ACM Press, Addison-Wesley Publishing Company, Reading, MA (1989), pp. 283–308.
16. S. Weiser and F. Lochovsky, "OZ+: An Object-Oriented Database System," *Object-Oriented Concepts, Databases and Applications*, W. Kim and F. Lochovsky, Editors, ACM Press, Addison-Wesley Publishing Company, Reading, MA (1989), pp. 309–337.

Accepted for publication December 13, 1991.

Allan J. Finkel *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Finkel is a research staff member and project leader of the Systems Management project. He received a bachelor's degree from the State University of New York at Binghamton in 1977 and a Ph.D. in mathematics in 1982 from New York University where he studied at the Courant Institute of Mathematical Sciences. During 1982–1983 he was a member of the Institute for Advanced Study in Princeton, New Jersey. Dr. Finkel joined the Mathematical Science Department of IBM Research in the fall of 1983 as a postdoctoral fellow and moved to the Computer Science Department in 1985. His research interests include expert systems, systems management, and object-oriented databases.

Seraphin B. Calo *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York*

10598. Dr. Calo received a Ph.D. degree in electrical engineering from Princeton University in 1976. Since 1977 he has been a research staff member in the Computer Science Department and has worked and published in the areas of queuing theory, data communication networks, multiaccess protocols, expert systems, and complex systems management. He has managed research projects in the communications and systems performance areas and has served on the staff of the IBM Research Vice-President, Systems. He joined the Systems Analysis Department in 1987 and is currently manager of the Systems Management research group. Dr. Calo was directly involved in the architecture work that led to the definition of the IBM SystemView structure and continues to be involved in its technical evolution. He also participated in the related design work that led to the development of the Resource Object Data Manager (RODM). He holds two United States patents and has received two Research Division awards.

Reprint Order No. G321-5472.