39

MIB View Language (MVL) for SNMP

Kazushige Arai
2nd Development Department
Data Communications Division
NEC Corporation,
1131 Hinode, Abiko, Chiba, 270-11, Japan
Phone: +81-471-85-7650
arai@dcd.trd.tmg.nec.co.jp

Yachiam Yemini*
Computer Science Department
Columbia University,
New York, NY, 10027, USA
Phone: +1-212-939-7123
yemini@cs.columbia.edu

Abstract

This paper introduces "MIB view language (MVL)" for network management systems to provide capability of restructuring management information models based on SNMP architecture. Views concept of database management systems is used for this purpose. Our MVL can provide "atomic operation" feature as well as "select" and "join" features to management applications without changing SNMP protocol itself.

Keywords: Network Modeling; Views; SNMP

1 Introduction

Network management agents provide a data model of element instrumentation to the network management system (NMS). For SNMP agents[1], this data model is captured by respective MIBs, defined in terms of the structure of managed information (SMI) language [RFC1155]. From a perspective of traditional database technology [EN89] a MIB can be viewed as a database of element instrumentation data. The protocol provides a data manipulation language (DML) to query MIBs and the SMI provides a data definition language (DDL) to define the MIB schema structures. Management applications executing at the NMS can access and manipulate MIB data using the protocol query mechanisms.

A central difficulty in developing management applications is the need to bridge the gap between the data models rigidly defined in MIB structures and the data model required by an application. As a simple example consider a fault management application which requires data on health measures [GY93] associated with a network element. These health measures may be computed by sampling MIB variables sufficiently fast. For example, the error-rate associated with an interface can be computed by sampling the respective error counter and computing its derivative. Ideally, the agent should export a data model of health parameters that can be accessed and manipulated by the fault management application. However, the specific data model required can vary from element to element and among different installations, and over time. The MIB designers can not possibly capture the large variety of possible health functions in a rigid MIB.

Of course, it is possible for the application to retrieve raw MIB data and compute the health data model at the NMS. This solution can be highly inefficient and unscalable as it would force

*Supported by ARPA contract F19628-93-C-0170.

[1]The techniques and concepts introduced by this paper are cast within the framework of SNMP. They could be mapped to the GDMO framework of CMIP where they would play an equally important role. This mapping will be described in future work.

A. S. Sethi et al. (eds.), *Integrated Network Management IV*

excessive polling of MIB data. Furthermore, it does not allow various applications that execute at multiple NMS to share the computations of the health data model effectively. In a multi-manager environment such sharing of data models is of great significance.

An alternative approach, developed in this paper, is to support effective computations of user-defined data models — *views* — at the agent side. The ability to define computed views of data has found a broad range of applications in traditional databases. View definition and manipulation capabilities are integral components of virtually all database systems. This paper proposes to extend the SMI and agent environment to support similar view computations to meet the need of management applications to transform raw MIB data into useful information.

The health data model, for example, could be defined in terms of the proposed MIB view language (MVL). The MVL computations could be delegated [YGY91] to the agent's environment, or to a local manager. Views can be organized in and accessed through agent's MIB. Applications could use standard SNMP queries to access and retrieve these view definitions. One can thus consider view MIB as a programmable layer of transformations of raw MIB data into information required by remote management applications.

There is an approach to transform MIB data, especially for OSI SMI architecture [SB93]. In this paper, we concentrate on SNMP architecture and are introducing actual MVL.

In following sections, we describe what can be done with views (Section 2) and then provides actual MVL specifications (Section 3).

2 Views of Managed Data

A database system includes an intrinsic data model defined by its schema. This intrinsic model is used to provide effective access to stored data, anticipating certain patterns of use by applications. Often, however, applications require a different data model than the one that is stored. A view provides a mapping of the intrinsic stored-data model to the data model needed by the application.

The data model created by a view can be considered as a *virtual MIB*. A virtual MIB is computed by an agent and may be accessed by SNMP managers like any other MIB. The examples illustrate various applications of virtual MIBs.

An application may wish to correlate data in multiple related tables. In a relational databases such correlation is accomplished by computations of a join. Consider for example the MIBs used in a terminal server device. The physical layer of the terminal server, described by the RS232 MIB [RFC1317], includes a table (`rs232SyncPortTable`) describing the physical ports. The logical layer of the terminal server, described by the point-to-point protocol (PPP) MIB [RFC1471], includes a table (`pppLqrTable`) describing logical link objects. These tables are depicted in Figure 1(a) and (b). Suppose now that a management application wishes to correlate the status of logical links and of the physical ports which they use. This is important in isolating faults that are manifested by managed objects associated with both layers. To accomplish such correlation one would want to compute a join of the respective tables (Figure 1(c)).

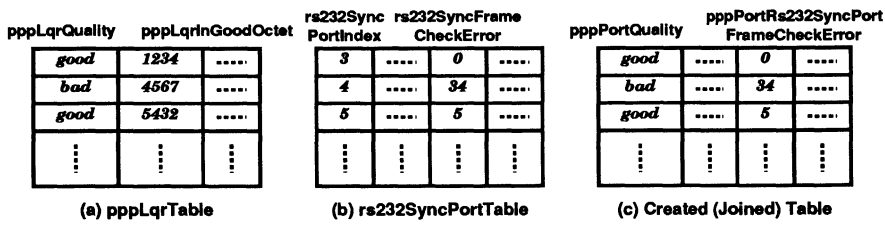

Figure 1: Example of "join" Tables.

In contrast with databases, neither SNMP nor CMIP provide a mechanism to correlate data by computing joins. In the example, the fault analysis application will have to retrieve both tables from the terminal server agent and compute the join. This computation of a join is very inefficient as much more data than needed will be retrieved and processed by the application. Moreover, it can lead to serious errors. Retrievals of tables by SNMP is not an atomic operation. Each GET-NEXT access will retrieve the current data in the respective tables. If attributes stored in the table change during retrieval the table images at the application side will reflect multiple versions of the respective MIB tables. The fault analysis routine may be mislead by the data to identify the wrong faults. Problem management could exacerbate the problems rather than resolve them. The problem of computing a join of table as an atomic action commonly occur in other network management scenarios. For example, resolution of routing problems typically involves correlation of routing, address translation and other configuration tables. It would be thus very useful to support effective computations of atomic joins.

Views can be used to perform such computations efficiently. A view computation could obtain an atomic (or a very good approximation of it) snapshot of the respective tables and then join them at the agent side. The joined table is a part of a virtual view MIB. It could be accessed by applications for retrievals via GET-NEXT (or GET-BULK) as any other MIB table. Atomic retrievals, of course, can be important even when tables are not joined. A view could be used to generate an atomic snapshot of a MIB table in the virtual MIB which could then be retrieved by managers.

Views may be used, similarly, to select objects that meet certain filtering criteria of interest. Selective retrievals are provided by CMIP via filters passed to agents as part of queries. In contrast, SNMP does not permit filtering of data at the source. Consider the terminal server example. Suppose one wishes to retrieve logical link data for all troubled links (defined by some filtering conditions on link status). At present, it is necessary to retrieve the entire logical links table and perform the filtering at the manager. This is inefficient and presents great difficulty in searching large tables (e.g., of thousands of virtual circuit objects in AtomMIB [ATOMMIB]). A view could be defined over the logical link table to perform the filtering required by the manager. A GET-NEXT access to this view will retrieve the next logical link that meets the filtering criteria. This can be used to augment SNMP with selective retrievals without any changes to the protocol. Furthermore, this method of filtering could be more efficient than the one pursued by CMIP since the filters are delegated ahead of access and require no parsing and interpretation during access time.

Views may be used to support participatory management of complex multi-domain networks. Consider for example a collection of private virtual networks (PVN) sharing a common underlying physical network. Such PVN are commonly used by telecommunication service providers as means to partition bandwidth among multiple organizations. NMS responsible to manage the various PVN must share access to agents of the underlying network elements. At the same time, their access should be limited to monitor and control the resources in their respective PVN. It is thus necessary to provide each PVN with a view of the actual MIBs. SNMPv2 [RFC1442] provides a "context" mechanism to support projection view of a MIB. A party may be authorized to access a subset of the MIB. Views significantly extend this mechanism to support not only projections but also computed data. The virtual MIBs accessed by PVN may hide some of the underlying network features to prevent PVN from compromising sensitive resource data.

Views may be used to support atomic actions in a multi-manager environment. In the multi-manger environment, it is difficult to ensure atomicity of actions invoked from several managers. With SNMP architecture, a side-effect of a SET operation is used to invoke an action. This operation may take one or more parameters which control a behavior of the action. When an action invoked by setting an value to an object (trigger object), agent may treat one or more other objects as parameters related to the action (parameter objects). But a parameter object set by one NMS may be modified by other managers before the previous manager invokes the action by setting trigger object. This can lead to incorrect behaviors. A view can define the action trigger and its parameters as an atomic group. This will associate with the group a queue of action requests. Each SET invoked by a manager to any object in the group will be queued. When all object SET requests by a given

manager have been received in the queue the action is invoked atomically. Should two managers access the action concurrently, their actions are serialized by the queuing mechanism.

Views could also provide a beneficial mechanism to protect access to data. A view can be used to define the data model and access rights available to certain applications. This is routinely used in databases to secure data access. SNMPv2 has this capability but view could provide it even with SNMPv1. However, a full discussion of view applications to secure management is beyond the scope of this preliminary paper.

Finally, views may be used to simulate abstraction/inheritance relations among SNMP objects, similarly to the object model provided by CMIP. For example, a view could define a port object and its properties as a common abstraction of various port objects in different MIBs. The abstract port properties could be mapped by the view (simulating inheritance) to properties of the specific port objects in the MIB. Similarly, one can use views to model containment relations among objects. These features, however, is beyond the scope of this paper.

In summary, a view could be used to support extensive computations over MIBs (correlations and filtering of data), atomicity of data access and actions, access control and object abstractions/inheritance and containment. These capabilities are summarized in Figure 2.

FEATURE	DESCRIPTION
CORRELATION	Join tables to create new table which contains correlated data.
ATOMIC RETIEVE	Generate atomic snapshot of a MIB table which can be retieved atomicly.
FILTERING	Select data which meet a filtering condition at agent side.
SELECT PARTIAL MIB	Provide partial access to each manager in multi-manager environment.
ATOMIC ACTION	Garantee atomic invocation of actions in multi-manager environment.
SECURE ACCESS	Define access rights to each management application.
OBJECT-ORIENTED MODEL	Simulate data abstructions and representation of containment relationships.

Figure 2: Summary of View Features.

3 MIB View Language

This section introduces the *MIB View Language (MVL)*. The goal of MVL is to provide a minimal extension of SNMP's SMI [RFC1155, RFC1442] that supports:

- definitions of the structure of view objects
- conversion of data from real MIB objects to compute view objects.

Traditional database systems use SQL, the data manipulation language, to define views. For example,

```
CREATE VIEW View1
AS SELECT      T.Attr1, T.Attr2
   FROM        Real_Table T
   WHERE       T.Attr3=0 AND T.Attr4=1
```

These SQL expressions accomplish definitions of the structure of view objects and their computations from real objects simultaneously, using a `SELECT-FROM-WHERE` construct. MVL develops a similar approach to view definitions, adapted to the SMI.

View definitions in MVL are compiled by a MVL compiler into appropriate agent computations and MIB structures for the view MIB. Access to a view MIB by a manager is indistinguishable from access to any other MIB.

An important consideration in implementing views is the organization of a view MIB and access within a complex multi-MIB agent environment. There are a few issues that an implementation architecture must address.

1. how does a manager query of a view MIB get processed
2. how are computations of a view MIB executed
3. how do view computations access real MIB objects
4. how are views delegated to an agent environment

A comprehensive discussion of the architectural options to address these questions is beyond the scope of this paper. We provide here a brief summary of one possible solution. View computations are encapsulated in a view agent. A view agent can function as a subagent within a multi-agent environment. An SNMP query of a view will be communicated by the master agent to the subagent (e.g., using one of a number of mechanisms currently available such as SMUX, WINSNMP, or other extensible agent mechanisms). The view agent is entirely responsible to compute the views. Views can be delegated to the view agent using the management by delegation mechanisms [YGY91]. Figure 3 depicts the overall organization of the different components of a typical SNMP management environment extended with view mechanisms.

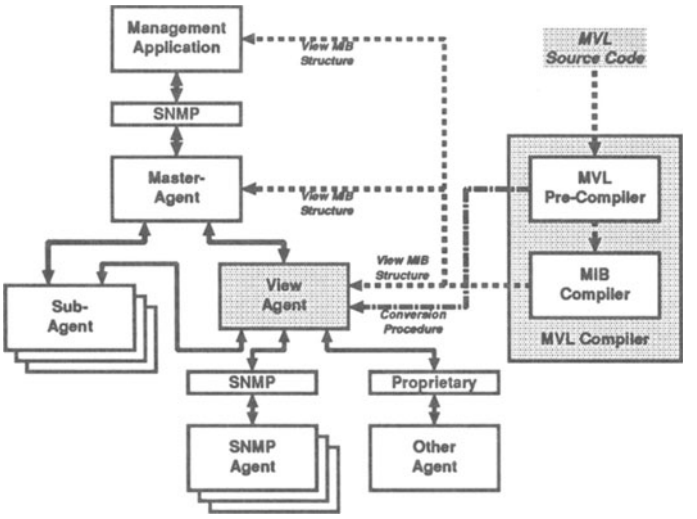

Figure 3: View Agent and MVL compiler.

Notice that a view agent may act as a manager and use proprietary or standard protocols to access remote agents and retrieve data needed in computing views. This may be accomplished by functions, invoked through view computations, to access and retrieve remote data.

3.1 The VIEW-TYPE Macro and View Function

The data structure of view object can be defined by a modified **OBJECT-TYPE** macro of SNMP. We called it a **VIEW-TYPE** macro. The only difference between view objects and real objects is that the value of the view object is computed from existing managed objects. Therefore we introduce **VIEW-FUNCTION** block to specify how to compute the value of a view object. **COMPUTED-BY** clause is used to bind view object and view function. Figure 4 illustrates a view definition using a **VIEW-TYPE** macro and a **VIEW-FUNCTION** block.[2]

```
viewObject1 VIEW-TYPE
    SYNTAX        INTEGER
    MAX-ACCESS    read-write
    STATUS        current
    DESCRIPTION   "Example of view
                   definitions"
    COMPUTED-BY   function1
    ::= {view 1}

function1 VIEW-FUNCTION
    SELECT realObject1
    WHERE  realObject1 <= 100
```

Figure 4: VIEW-TYPE and VIEW-FUNCTION.

3.1.1 COMPUTED-BY Clause

The **COMPUTED-BY** clause declares the function name which computes the value of the object by accessing existing objects. If no **COMPUTED-BY** clause appeared in **VIEW-TYPE** macro, value of the view object is persistently stored into the view MIB instead of being computed from existing real objects. In this case the value of the object should be initialized by a manager application or through a **DEFVAL** clause.

3.1.2 VIEW-FUNCTION Block

Each function declared by a **COMPUTED-BY** clause has to be defined using a **VIEW-FUNCTION** block. This defines conversion procedures of data. A **VIEW-FUNCTION** block is based on SQL. It consists of two clauses, "**SELECT**" and "**WHERE**".

SELECT Clause The SELECT clause defines how to access values of existing objects in computing a view object. Note that, the existing objects specified here may be other view objects. The following operators are available for computing selection:

[ ] operator Indexing column object in table structure (See Section 3.2)

− > operator To access a column object from an object identifier of conceptual row object.

+, −, *, / operator Arithmetic operators for calculation

WHERE Clause The **WHERE** clause specifies a condition that filters the instances of objects accessed by the **SELECT** clause. The following conditional operators are available:

AND operator Logical AND operator

OR operator Logical OR operator

[2] In this paper, we use syntax of SNMPv2 SMI as an example. But MVL is also applicable to SNMPv1 with few modification.

NOT operator Logical NOT operator

IN operator Compare two object identifiers (OIDs) whether right hand OID is included in the left hand OID's subtree

=, <>, <, >, <=, >= operator Compare magnitude of two expressions

[ ], − >, +, −, *, / operator All these operators described with `SELECT` clause can be used in `WHERE` clause

The key word "`SELF-INDEX`" is used as an index value of `[ ]` operator to specify index value of the view object its self. (See Section 3.2)

3.2 Computing Join and Selection View Tables

Specifying the computations of view tables from real tables is particularly challenging. One must identify how conceptual rows in the view table relate to respective rows in the real tables. Of particular significance is the computation of the index of a view table. The simplest case is when a view table uses a column from a real table as its index. This is illustrated by the following example.

```
viewIfIndex VIEW-TYPE
    SYNTAX INTEGER
      :
    COMPUTED-BY func_ifIndex
      :

func_ifIndex VIEW-FUNCTION
    SELECT ifIndex[SELF-INDEX]
```

In here `viewIfIndex` is the index column of the view table and `ifIndex` is a column of a real MIB table. The notation `[SELF-INDEX]` is used to specify the index of the real MIB table containing `ifIndex`. Of course, one must ensure that the values in `ifIndex` can be suitably used as an index (i.e., they are key for the view table).

Consider now the case where the view table is created by selecting a subset of conceptual rows from the real table. This may be used to filter row entities using appropriate filtering condition. For example, `ifOperStatus` represents the operational status of interface objects and a value 1 represents that an interface is operational. [RFC1213, RFC1573] The following example creates a view table that includes index values for all operational interfaces. A manager accessing this view table via GET-NEXT could retrieve index values for operational interfaces only.

```
func_column1 VIEW-FUNCTION
    SELECT ifIndex[SELF-INDEX]
    WHERE  ifOperStatus[SELF-INDEX] = 1
```

We now illustrate how to specify join views using MVL expressions. Consider two tables, `ifTable` [RFC1213, RFC1573] and `atmInterfaceConfTable` [ATOMMIB] whose index column is `ifIndex`. We wish to create a view table that join the two tables using their common index values and containing the common index column, followed by `ifSpeed` of `ifTable` and then the `atmInterfaceMaxVpcs` and `atmInterfaceMaxVccs` of `atmInterfaceConnTable`. This is depicted in Figure 5 and is accomplished by the MVL specification in Figure6.

3.3 Computing Atomic Operations in MVL

Supporting invocation by managers of actions at agents is of great significance in management. Remote actions can be used to control configuration (e.g., partition hub ports, establish permanent virtual circuits through a switch or configure collection of statistics by a remote monitor) or invoke

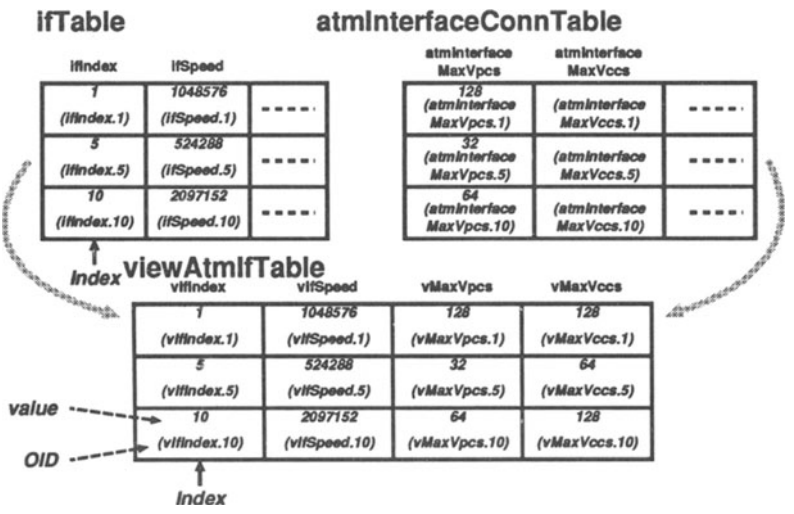

Figure 5: Join Tables.

diagnostic procedures. CMIP, therefore, supports explicit invocation of remote procedure calls. In contrast, SNMP utilizes side-effects of SET to invoke agent procedures. This implicit invocation of remote procedures is seriously limited in passing parameters to actions. A manager would have to ensure that parameters are set prior to triggering the execution of an action that uses them. In a multi-manager environment interference among managers trying to invoke a parameterized actions could lead to erroneous actions. One manager could reset the values of parameters just set by another manager who issues an action triggering request.

The parameterized action model of SNMP may be best viewed as a form of supporting transactions among managers and agents. The problem is, accordingly, that of supporting concurrency control of such transactions to assure their serializability. Interference among managers can lead to non-serial execution schedules.

Currently, there are several approaches to realize the parameterized atomic action. For example, using "lock" variable to control write access to parameters of action is the most popular method for the concurrency control.

Managers must check the lock variable before modifying the parameters and if it is not set, the manager set the variable to lockout access from other managers. The agent keeps an ID of the manager which set the variable, and does not accept SET access from other managers. This method still have a chance of conflict to access lock variable itself.

Another example to realize the parameterized atomic action is introduced by using row creation of SNMPv2. With this method, each manager which invokes an action creates a new row which contains parameters of the action. Since the other managers does not know the identifier of the new row, this manager can set the parameters without conflict from other managers. This method is fit for an action like create a new virtual circuit. But it may not be appropriate for changing parameters of some services. And this method cannot be applied to current SNMP.

MVL provides a simple and generic mechanism to support concurrency control of SET transactions by multiple managers. MVL uses the `ATOMIC-GROUP` construct to accomplish this. We use the example in Figure 7 to illustrate the atomic execution mechanisms of MVL.[3]

[3] This example is based on an action of virtual path (VP) cross-connect establishment described in [ATOMMIB],

```
viewAtmIfTable VIEW-TYPE
    SYNTAX SEQUENCE OF
              vAtmIfTableEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION "ATM Interface Table"
    INDEX {vIfIndex}
    ::= {view 1}

vAtmIfTableEntry VIEW-TYPE
    SYNTAX VAtmIfTableEntry
    MAX-ACCESS not-accessible
    STATUS current
    DESCRIPTION "Conceptual row"
    ::= {viewAtmIfTable 1}

VAtmIfTableEntry ::= SEQUENCE {
    vIfIndex INTEGER,
    vIfSpeed Gauge,
    vMaxVpcs INTEGER,
    vMaxVccs INTEGER
    }

vIfIndex VIEW-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
      "ifIndex from ifTable"
    COMPUTED-BY func_vIfIndex
    ::= {vAtmIfTableEntry 1}

func_vIfIndex VIEW-FUNCTION
    SELECT ifIndex[SELF-INDEX]
    WHERE  ifType[SELF-INDEX] = 37

vIfSpeed VIEW-TYPE
    SYNTAX Gauge
    MAX-ACCESS read-only
    STATUS current
    DESCRIPTION
      "Interface speed"
    COMPUTED-BY func_vIfSpeed
    ::= {vAtmIfTableEntry 2}

func_vIfSpeed VIEW-FUNCTION
    SELECT ifSpeed[SELF-INDEX]
    WHERE  ifType[SELF-INDEX] = 37

vMaxVpcs VIEW-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-write
    STATUS current
    DESCRIPTION
      "Max. number of VPCs."
    COMPUTED-BY func_vMaxVpcs
    ::= {vAtmIfTableEntry 3}

func_vMaxVpcs VIEW-FUNCTION
    SELECT atmInterfaceMaxVpcs
                        [SELF-INDEX]
    WHERE  ifType[SELF-INDEX] = 37

vMaxVccs VIEW-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-write
    STATUS current
    DESCRIPTION
      "Max. number of VCCs"
    COMPUTED-BY func_vMaxVccs
    ::={vAtmIfTableEntry 4}

func_vMaxVccs VIEW-FUNCTION
    SELECT atmInterfaceMaxVccs
                          [SELF-INDEX]
    WHERE  ifType[SELF-INDEX] = 37
```

Figure 6: Example of MVL definition to join tables.

```
vVpConnConf VIEW-TYPE
    SYNTAX TimeTicks
    MAX-ACCESS read-write
      :
    ATOMIC-GROUP { vVpConnLowIfIndex,
                   vVpConnLowVpi,
                   vVpConnHighIfIndex,
                   vVpConnHighVpi
                   vVpConnAdminStatus,
                   vVpL2HOperStatus,
                   vVpL2LOperStatus }

vVpConnLowIfIndex VIEW-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-write
      :
    COMPUTED-BY func_LowIfIndex

func_LowIfIndex VIEW-FUNCTION
    SELECT
      atmVpCrossConnectIndex[SELF-INDEX]
      :

vVpConnAdminStatus VIEW-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-write
      :
    COMPUTED-BY func_AdminStatus

func_AdminStatus VIEW-FUNCTION
    SELECT
      atmVpCrossConnectAdminStatus
                          [SELF-INDEX]
      :

vVpL2HOperStatus VIEW-TYPE
    SYNTAX INTEGER
    MAX-ACCESS read-only
      :
    COMPUTED-BY func_L2HOperStatus

func_L2HOperStatus VIEW-FUNCTION
    SELECT
      atmVpCrossConnectL2HOperStatus
                          [SELF-INDEX]
      :
```

Figure 7: Example of Atomic Group.

The view object **vVpConnConf** is defined as an *atomic object* with a value of **TimeTicks**. The **ATOMIC-GROUP** declaration binds a group of view objects to an action (transaction) associated with **vVpConnConf**. The group of view objects is called *atomic group*. When a manager starts to invoke the atomic action, it would first SET the **vVpConnConf** with a value of time-out by which time the atomic action is canceled. Once the atomic object is SET, all subsequent SET accesses to any objects in the atomic group by the same manager are queued until the view agent has obtained a SET for all objects in the atomic group whose access is **read-write** (In this example, **vVpConnLowIfIndex**, **vVpConnLowVpi**, **vVpConnHighIfIndex**, **vVpConnHighVpi** and **vVpConnAdminStatus**). At that time, the view agent executes all these SETs in the order defined by their request-id. When the last SET request is executed, the action is invoked at real agent (In this example, virtual path is connected). The other SET requests are used to set parameters of the action.

After finishing all SET request executions, the view agent will execute GET requests to all read-only objects in the atomic group (In this example, **vVpL2HOperStatus** and **vVpL2LOperStatus**). These read-only objects are used to return results of the atomic action. The view agent takes an atomic snapshot of these values for subsequent GET and GET-NEXT accesses by the manager. The snapshots will be deleted either through another SET request by the same manager or through a timer (**vVpConnConf**) expiration on the time-out. If the timer expires before all objects are SET, the atomic action will be canceled and the agent dose not execute any SET request issued by the manager.

If all the variables in the **ATOMIC-GROUP** are **read-only**, then the agent interprets the SET request to the atomic object as an atomic retrieval initiation. It takes a snapshot of all these objects in the **ATOMIC-GROUP** and stores them. All subsequent GET or GET-NEXT access to these variables by

but it is slightly modified from actual MIB definitions. Because, original definitions have their own way to provide atomic action with row creation techniques described above.

the manager that issued the SET retrieves this atomic snapshot.

MVL also provides another capability of atomic retrieval which is called *asynchronous update*. Suppose that there are two or more counters defined in a MIB (a real MIB). And they are being updated concurrently. There is no guarantee that two counter values which are retrieved by manager are consistent. Because, the values may be updated by agent after the manager retrieves one counter value and before retrieves another one. Retrieving two values on different cycles of updating may make inconsistency between these values.

MVL uses **UPDATE-GROUP** construct to prevent this inconsistency. Example in Figure 8 is used to illustrate the asynchronous update mechanism.

```
updateErrorCounts VIEW-TYPE
   SYNTAX TimeTicks
   MAX-ACCESS read-write
      :
   UPDATE-GROUP { vIfInErrors[SELF-INDEX],
                  vIfOutErrors[SELF-INDEX] }
   UPDATE-CONDITION
     IF ifOperStatus[SELF-INDEX] = 1
      :

vIfInErrors VIEW-TYPE
   SYNTAX Counter
   MAX-ACCESS read-only
    :
   COMPUTED-BY func_inErrors

func_inErrors VIEW-FUNCTION
   SELECT ifInErrors
                [SELF-INDEX]
```

Figure 8: Example of Update Group.

An **updateErrorCounts** is declared as an object controls asynchronous update with a value of **TimeTicks**. The **UPDATE-GROUP** declares a group of view objects which are updated simultaneously. The view agent updates a snapshot of the view objects in the group by getting values through view functions of each view objects. By default, interval time of update is given as a value of object defined with **UPDATE-GROUP** declaration (In this example, value of **updateErrorCounts**).

Since all member objects of **UPDATE-GROUP** are updated simultaneously, managers always retrieve values with in a same update cycle by GET access. If agent receives GET request during it is updating values, the request is queued and responded after finishing update process. On the other hand, if update timing is reached during process of responding GET access, the update process is delayed after completion of the response.

UPDATE-CONDITION clause specifies conditions of updating members of the **UPDATE-GROUP**. In an example of Figure 8, the agent updates the member objects only if a value of **ifOperStatus** is equal to 1 (UP). Any other condition identical to **WHERE** clause can be specified with **IF** clause. **TIMING** clause can also be specified to determine more complex update schedule. For example: **TIMING { 10:00, 12:00, 14:00, 16:00 }** declares that the member objects are updated at 10:00, 12:00 and so on. With this declaration, the value of **updateErrorCounts** is ignored.

4 Conclusion

Introducing views concept of database management systems into managed object definitions of network management systems provides capability of restructuring network models. This capability makes it is possible that network models can be modified to be best for each management application. Especially, views provides "select" and "join" features of database management systems and they make development of network management application easier and they can also reduce traffics between manager and agent nodes.

We introduced "MIB View Language (MVL)" for SNMP architecture which can be used without changing any protocols between manager and agent. With MVL compiler, we can produce MIB structure for view agent and view functions which convert existing data models to view models.

MVL and view agent also provide atomic operation features. With these features atomic invocation of actions and asynchronous update of view objects, which is not available with current SNMP architecture, can be achieved without changing SNMP protocol itself.

References

[ATOMMIB] M.Ahmed, K.Tesink, Editor, "Definitions of Managed Objects for ATM Management, Version 7.0", *Internet Draft*, 1994

[EN89] R.Elmasri, S.Navathe "Fundamentals of Database Systems", The Benjamin/Cummings Publishing Company, Inc., 1989

[GY93] G.Goldszmidt, Y.Yemini, "Evaluating Management Decisions via Delegation", *Integrated Network Management, III*, Elsevier Science Publishers B.V. (North-Holland), 1993

[RFC1155] M.Rose, K.McCloghrie, "Structure and Identification of Management Information for TCP/IP-based Internets", *RFC-1155*, 1990

[RFC1213] K.McCloghrie, M.Rose, "Management Information Base for Network Management of TCP/IP-based Internets: MIB-II", *RFC-1213*, 1991

[RFC1317] B.Stewart, Editor, "Definitions of Management Objects for RS-232-like Hardware Devices", *RFC-1317*, 1992

[RFC1351] J.Davin, J.Galvin, and K.McCloghrie, "SNMP Administrative Model", *RFC-1351*, 1992

[RFC1442] J.Case, K.McCloghrie, M.Rose, S.Waldbusser, "Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2)", *RFC-1442*, 1993

[RFC1471] F.Kastenholz, "The Definitions of Managed Objects for the Link Control Protocol of the Point-to-Point Protocol", *RFC-1471*, 1993

[RFC1573] K.McCloghrie, F.Kastenholz, "Evolution of the Interfaces Group of MIB-II", *RFC-1573*, 1994

[SB93] S.Bapat, "Richer Modeling Semantics for Management Information", *Integrated Network Management, III*, Elsevier Science Publishers B.V. (North-Holland), IFIP 1993.

[YGY91] Y.Yemini, G.Goldszmidt, S.Yemini, "Network Management by Delegation", *Integrated Network Management, II*, Elsevier Science Publishers B.V. (North-Holland), IFIP 1991