A Framework for SNMPv2 in ERLANG

Martin Björklund
Klas Eriksson
Computer Science Laboratory
Ellemtel Telecommunications Systems Laboratory

May 31, 1995

Master's Thesis in Computer Science (12 credits)
at the School of Computer Science and Engineering
Department of Numerical Analysis and Computing Science (NADA)
Royal Institute of Technology (KTH)
S-100 44 Stockholm, Sweden

Examiner: Lars Kjelldahl, NADA, KTH

Supervisors: Claes Wikström, Ellemtel AB and
Joacim Halén, NADA, KTH

Abstract

The Simple Network Management Protocol (SNMP) has mainly been used for managing IP (Internet Protocol) networks. Until recently, OSI (Open Systems Interconnection) management has been the choice for managing non-IP-networks as well as larger systems where management by delegation is necessary. With version 2, SNMP became more suitable for these systems.

One of the drawbacks of SNMP has been that it focuses more on implementation than on design. When implementing large network management systems, the design phase is essential. This report presents a framework for SNMPv2 where iterations in the design phase have a low turn-around time. Prototyping of MIBs (Management Information Bases) is simplified by automated implementation of instrumentation functions for scalar variables as well as tables.

The purpose of the framework is to facilitate MIB design and agent implementation by providing a user-friendly environment. For this, the ERLANG programming language has shown to be an appropriate choice.

The report also discusses design tools that could be built on top of the framework.

Keywords: Network Management, SNMP, MIB, OSI, ERLANG.

Ett programmeringsverktyg för SNMPv2 i ERLANG

Referat

Tidigare har "The Simple Network Management Protocol" (SNMP) först och främst använts för att övervaka IP-nätverk (Internet Protocol), men i och med version 2 har SNMP blivit mer lämpat även för andra nätverk. För större system har OSI (Open Systems Interconnection) management varit det naturliga valet, men nu börjar SNMP bli ett realistiskt alternativ.

SNMP har kritiserats för att det är mer inriktat på implementationsfasen än designfasen. Om det skall vara möjligt att bygga större nätverksövervakningssystem med SNMP, behövs hjälpmedel för design och prototypning. I denna rapport beskrivs ett programmeringsverktyg i ERLANG som är ett första steg i denna riktning.

Med hjälp av verktyget är det enkelt att implementera en prototyp för en SNMP-agent, d.v.s. en agent där MIB:en (Management Information Base) inte har någon koppling till den faktiska enheten som den skall styra. Agenten kan besvara förfrågningar genom att själv lagra tabeller och variabler i en databas. Verktyget underlättar också vid implementation av en riktig agent.

I rapporten behandlas dessutom nätverksövervakning i allmänhet och SNMP i synnerhet. Förslag och skisser på ännu kraftfullare MIB-utvecklingsverktyg presenteras också.

Preface

This Master's project was performed at the Computer Science Laboratory at Ellemtel. We would like to thank our supervisor Claes Wikström and the rest of the laboratory for their help and support.

Section 2.2 on Enterprise Management is inspired by our discussion with Richard Bruvik who is head of the Network Operations Center at Ellemtel.

For more information about this Master's project, send e-mail to
d90-mbj@nada.kth.se (Martin Björklund) or
d90-ker@nada.kth.se (Klas Eriksson).

Contents

1 Introduction

In the open network management world, two main approaches are taken. We will refer to these as OSI management and Simple Network Management Protocol (SNMP).

Since the need for open network management protocols arose, the telecommunications community has favored the OSI network management protocol and specification language. Recently, interest of SNMP has aroused. One purpose of this project was to analyze SNMP's scalability properties, to investigate how SNMP can be used in larger systems, such as telecommunications software. Another purpose was to design and implement a framework for SNMP agents in ERLANG.

SNMP was developed in the Internet world, for managing IP-networks. It is not obvious that SNMP is suitable for more general purposes. In fact, version 1 is not suitable for anything else than managing IP-networks. The protocol definition [RFC1157] only considers certain IP related tables and explicitly states how to handle these. This problem is taken care of in SNMP version 2. Consequently we have only considered SNMPv2 in our work. We will use the term SNMP for SNMPv2 throughout this report.

Both SNMP and OSI management are described since the latter has been the natural choice for large networks. We present an architectural model and describe the approaches according to it.

We are *not* comparing SNMP to OSI management. Such comparisons have been done before ([HEGE], [SLOM] and [RUTT]). OSI management is based on an object-oriented approach, whereas SNMP is based on a hierarchy containing simple variables and tables. Therefore a comparison can not evaluate 'SNMP object-oriented features' [SLOM]. Most comparisons have evaluated SNMPv1, which means that they are out of date, as SNMP has undergone major enhancements. Some people claim that 'S' in SNMP now stands for Sophisticated rather than Simple [BERK].

Even though this report is not a comparison, it can be read simultaneously to one. In this way you will get a fair picture of SNMP's profits and limitations. It is important to bear in mind that the two management approaches are suited for different problems. Hopefully, this report in combination with existing literature will give you a clue of which approach to use.

1.1 Design Goals

The design goals for our SNMP framework implementation are:

- *Ease of use.* The user should have to specify as little as possible to get a running agent. All details of SNMP interaction should be hidden from the user and automatically taken care of. The programming interface should to be simple, in order to support for rapid prototyping.

- *Flexibility of functionality.* It must be possible for the user to override the default behavior whenever necessary. As well as making prototypes and perform testing, constructing a real product should be possible.
- *Flexibility of code.* Our code should be designed to facilitate the construction of any SNMP program, for example an SNMP manager.
- *Extensible.* It should be possible for our environment to serve as a basis for implementing other tools, such as tools for MIB design, simulation and testing.
- *Compactness.* Our source code should be small and easy to understand.

1.2 Intended Audience

This document is intended for readers interested in network management in general and SNMP in particular. We have focused on the agent side of SNMP, so we hope MIB designers and implementors will find this report interesting. We propose a simple way of setting up an SNMP-agent. Those with experience of other network management tools will hopefully notice the difference in complexity.

We assume that our readers are familiar with some basic concepts of networking, protocols, data modeling and concurrent programming.

1.3 Outline of this Report

Our work has been divided into two phases. Firstly, we wanted to gain experience with SNMP and networking are. To do this we implemented an SNMP-agent in the ERLANG programming language [ARMS]. Secondly, we studied the theoretical aspects of networking, mainly design of MIBs for SNMP.

The first part of the report concerns the theoretical aspects of our work. The most technical aspects, our implementation, is in the second half. For a more detailed description of the framework, please consult the User's Manual.

Chapter 2 concerns network management in general.
Chapter 3 introduces an architectural model for network management frameworks, and describes SNMP and OSI management according to this model.
Chapter 4 gives an introduction to MIB design, and contains guidelines for structuring management information.
Chapter 5 introduces the technical basics of SNMP.
Chapter 6 describes our generic SNMP agent framework and how it is used.
Chapter 7 gives some examples on how it is possible to use the design principles from chapter 4 to build higher levels tools upon our framework.

2 This is Network Management

2.1 The need for Network Management

Network management could be defined as all procedures and products for planning, configuring, controlling and monitoring computer networks. The intention is to ensure efficient use of all resources. From the user's point of view, the best network is the invisible network where you can communicate without bothering about what is in between. To achieve this, it is important to understand why network management is a complex problem.

Networks of today are populated by a large and increasing number of resources from different suppliers. Common components are hosts, routers and printers but as new services are being added, they are becoming even more varied. All these heterogeneous components should be managed in some standardized manner.

Most networks of any considerable size have a network control center to control and monitor the computer communications. The computer dedicated to this task is called the *management station*. It is responsible for handling the enormous amount of information from all network components.

When a fault occurs in the network, the challenge for the network operator is often not how to correct the error, but to find out where the fault occurred. Therefore the management station must help the operator to efficiently sieve information from the components. Sieving should be so efficient that the management station is completely quiet most of the time. If a small fault occurs in a resource somewhere, this is automatically reported by the resource to the management station which is to take appropriate action. Since most faults come from erroneous configuration, they can be corrected easily or even automatically by the system itself. The vision is a computer network which is quiet until something goes wrong, then it sends a message to the operator describing the error and how to correct it. The situation today is far from this ideal.

2.2 Enterprise Management

Network management could be divided into three sub-areas. These are shown in figure 1. Even the most primitive computer systems of today can monitor connectivity, usually with a command similar to the Unix command `ping` to find out whether or not computer A is connected to computer B. In addition to pure connectivity monitoring, larger systems collect statistics such as response times, throughput, loading, error rates and availability. Either this is done manually or automated, that is, the management station polls selected components for interesting information. The third sub-area of network management slowly gaining ground is configuration management. In our experience, only a very few and simple configuration tasks can normally be performed remotely today.

These three sub-areas capture what is commonly meant by network management. The aim is to keep the network running smoothly. In a larger perspective, every

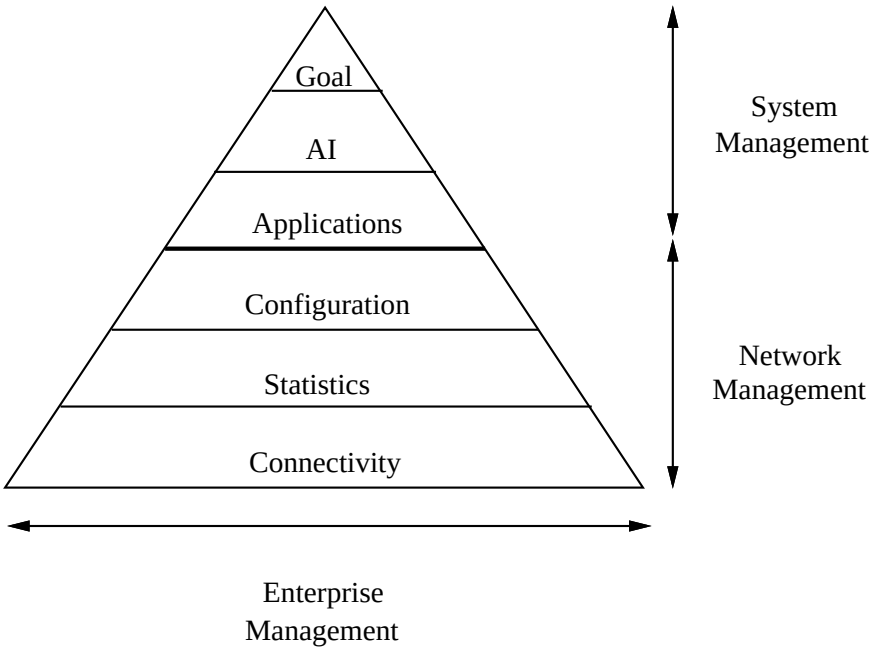

Figure 1: *The Pyramid of Enterprise Management*

enterprise has a global goal. Thus, networks should be managed with that goal in mind. When taking on this more holistic perspective, we are talking of *enterprise management*. This concept is pictured in the pyramid in figure 1.

When management has become so mature that configuration can be done fully remote, we are ready to begin managing the system as a whole. This is called *system management*. At this stage, applications important for the goal of the enterprise can be monitored and controlled. One scenario is a heavy batch process running at night, perhaps a backup procedure. If this is not finished until morning, the ordinary work will be negatively affected. The management station should detect this, and signal an alarm *Operator: Backup not finished in time*.

On top of this is the layer termed 'Artificial Intelligence'. This is a management system taking own initiatives. If the enterprise gets a large order, the management system would be able to predict that the existing computer resources is not enough, and send an order for more computing power.

Today, the network management world is entering the system management level, where we are able to manage the system as a whole. This is necessary since it would be inconvenient trying to manage large systems "by hand", in other words, staying at level 2-3 in the pyramid. As networks become larger and more complex, even more must be automated, hence we have to continue to higher levels in the pyramid.

2.3 Management in Telecom

Apart from managing true computer networks, there is also a need to manage telecommunications networks. This is essentially the same problem as in computer networks. The aim of management in Telecom is to achieve interconnection among the operating systems and telecommunications systems that must exchange management information to assure the smooth and continuous operation of the telecommunications services. Telephone networks, mobile radio networks and data networks should be managed in a uniform way. These networks consists of more complex components than an ordinary computer network, furthermore the number of components is large.

Management in Telecom is meaningful first when entering the system management level. Configuration, that is, subscribing new customers, changing telephone numbers etc. is the business concept. Therefore this level must be completely developed before management of these networks can succeed.

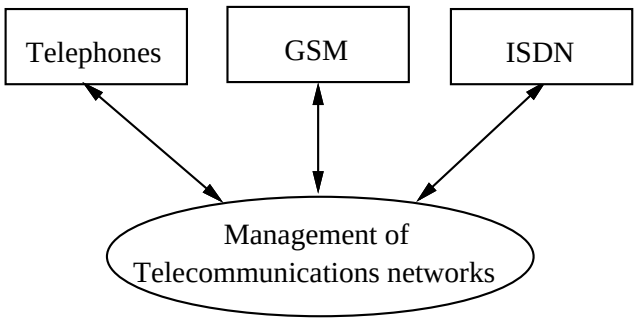

Figure 2: *Management of Telecommunications Networks*

3 Network Management Concepts

This chapter describes the fundamental concepts of network management. The concepts are essential to understand the various aspects of networking. To find out what is needed for design of complex network management systems, the scalability properties of the system have to be investigated. If the system grows, what parts will be affected and how?

The network management world is divided into two competing halves, at least when considering open systems. Within the Internet world the Simple Network Management Protocol (SNMP) evolved and in parallel ISO developed OSI management. Today SNMP is market leader.

SNMP was designed with the two most important philosophies of the Internet community in mind: the importance of *implementation experience*, and the importance of *lean design* with absence of extraneous features and misfeatures. It focuses more on implementation and test phases, than on specification and design phases. This allows the MIB-designer to do 'quick and dirty' implementations, although structured MIB-design is indeed possible.

For larger systems, OSI management has been the natural choice. OSI management was designed with a completely different approach: it should be "the ultimate solution" to all network management problems.

3.1 Architectural Model

To deal with a heterogeneous network management environment we need a general framework, an *architectural model*. Within this framework, specific management systems are implemented.

The system requirements come from different sources. First, the user should be guaranteed good service and maximal availability. Second, the system manager wants a user-friendly application where it is easy to monitor the network as a whole. Third, it should be simple enough for network element providers to implement. Finally, the architecture should be scalable in order to accomplish enterprise management.

The fundamental network management system consists of:

- several managed nodes (for example, a bridge, modem, router or host), each containing an *agent*.
- at least one *manager*, human or automated, that can perform management activities.

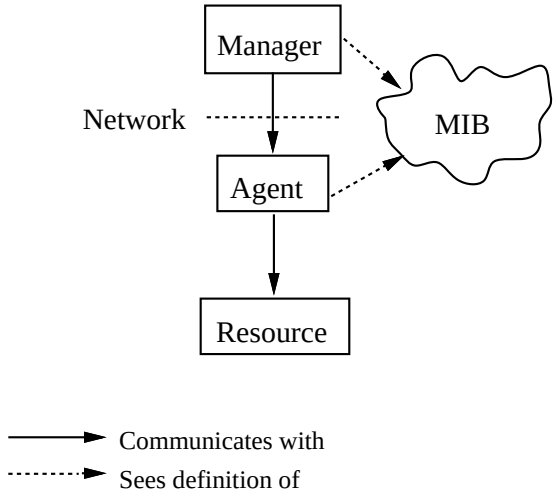

Figure 3: *The Manager-Agent model*

The Manager-Agent model is based on the client-server principle, see figure 3. One difference from the ordinary client-server model is that there is one client (manager) and many servers (agents).

To be able to clearly identify differences and similarities between SNMP and OSI management, we have chosen to describe the two architectures according to four submodels from [HEGE]. They are also helpful when identifying aspects affected by a growing system. The four submodels are:

- Information model. 'How is management information modeled?'
- Organizational model. 'Who are the participants in the management system and what are their roles?'
- Communication model. 'How does the protocol work?'
- Functional model. 'What functionality is provided?'

In the following subsections these submodels are presented in detail.

3.1.1 Information Model

There is a lot of information in the resources that should be managed and there are several theories of how to model the information. Well known models are the relational database model, the hierarchical model (used in for example file systems) and the object-oriented (OO) model.

The conceptual repository for management information is called the *Management Information Base* (MIB). It is conceptual since it does not hold any data, merely

a definition of what data which can be accessed. The MIB is a fundamental concept in network management because it defines the interface between a resource and the management system. More specific, a resource is represented by a *managed object* residing in the MIB which describes how to manage, monitor and control the resource. A description of a MIB is simply a description of a collection of managed objects. A managed object typically has at least the following properties:

- What it is (type).
- Who it is (name).
- How it is assembled (attributes, what data it carries).
- How it can be manipulated (methods).

For example, a modem could be represented as:

```
type: Modem
name: "IBM modem 12"
data: {speed: Integer, off-hook: Boolean}
methods: {hang-up(), reset(), set-speed(Integer),
          get-speed()}
```

Managed objects have different complexity in different network architectures. A managed object can be described as a class in OO, as a plain variable or as a table.

It is important to note that the managed object is a logical object, that is, it does not necessarily have to correspond to an object in the physical world. From the manager's point of view, it is not important if a value is physically stored in memory or needs to be computed, to answer a manager request. From the outside, it is the same logical object.

The normal procedure when constructing an agent, is to define the MIB in a specification language, along with the implementation of managed object methods, sometimes called the *instrumentation functions*. A well-designed specification language is concise and helpful for the information structuring process. In addition, it should be flexible and easy to revise, since the underlying resource often is in a process of change. Consequently, the language should support information encapsulation.

What makes the MIB concept so useful is that it defines a standardized interface for managing a resource. It is then up to each provider of network components to implement the MIB. Assume that we have a modem MIB. Then every modem manufacturer must implement the managed objects in the MIB. For example, they must implement the `hangup()` method for their particular modem.

In the design phase of large network management systems, the information model is extremely important. Chapter 4 is devoted to this.

3.1.2 Organizational Model

The organizational model defines the participants in the network architecture, their roles and how to distribute the management work. The simplest and most straightforward model is the Manager-Agent model, presented in the beginning of this chapter. In practice, this model is often too naïve, simply because of the size of the network being managed. Instead of having a flat organization with only one or a few managers and many agents, larger systems could be built using 'management by delegation'. This means that middle-managers are introduced to be responsible for interaction with a subset of all agents. The middle-manager plays a dual role as an agent for some higher level manager and as a manager of a subordinate agent. Suppose that a company has many agents for managing modems from different manufacturers, then a middle-manager can keep count of the total online time.

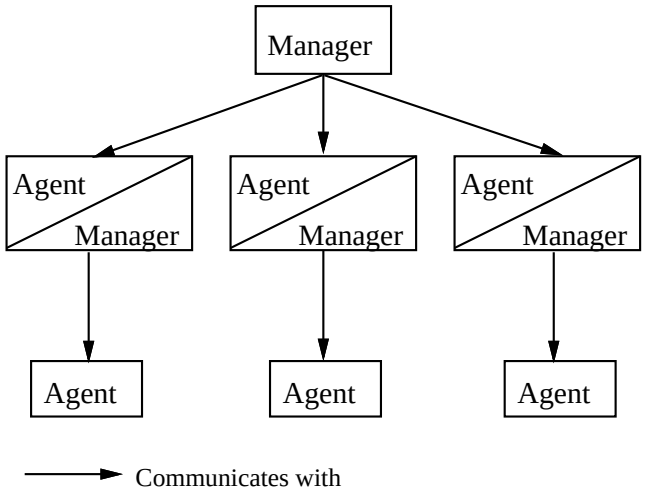

Figure 4: *A hierarchy of agents and managers*

We get a hierarchy (figure 4) of abstraction which has the property of being scalable when the network grows and enterprise management is getting closer.

3.1.3 Communication Model

The communication model describes the schemes for exchanging information between the actors within the network management system. It specifies the actors and what kind of messages they can process. The messages can be of different nature. For example, a modem pool which suddenly gets a malfunction may send a *trap message* to the manager station, simply saying 'Replace modem 14 immediately due to serious malfunction'. Other types of messages are sent from the manager to a agent in managed resource, such as: 'Give me your uptime' (a

status query), or 'Your new IP-address is 1.2.3.4' (an action). It should also be defined in the communication model whether these messages are synchronous or asynchronous.

Apart from the semantics of the messages, the syntax must be defined. The syntax describes how to put the message into a protocol data unit (PDU), and uniquely encode it into the bits which are to be transmitted over the network.

Another important aspect is to know what is required from the underlying protocol. Is an unreliable protocol such as UDP (User Datagram Protocol) sufficient, or do we need a protocol which is both reliable and connection-oriented (TCP)? Few requirements will make protocol embedding an easy task. A flexible network management protocol should be able to run on top of anything, no matter if it is UDP, TCP/IP or Appletalk.

3.1.4 Functional Model

The functional model divides management activities into various functional areas. Examples of functional areas are:

- *Configuration management.* Detecting and controlling the state of the network.
- *Performance management.* Controlling, analyzing and logging of throughput and error rate.
- *Fault management.* Detecting, isolating and controlling abnormal behavior, such as excessive line outages.
- *Accounting management.* Collecting and processing data related to resource consumption.
- *Security management.* Controlling access to network resources.

The reason for specifying distinct functional areas, is that the management station needs a generic way to handle management within an area. An area defines these generic mechanisms and it is up to every resource to follow them. For example, collecting statistical data, such as `sysUpTime`, should be done in a uniform way on different resources. The aim is to be able to manage the network as a whole. Hopefully it will also help the MIB-designer to structure information in the resource.

There are two schools of thought on how to implement a functional model. One is to include functional area definitions directly in the network architecture. The other is to use the information model to build generic MIBs for various functional areas.

3.2 Concepts of SNMP

"Perfection is not achieved when there is nothing left to add, but when there is nothing left to take away."

— Antoine de St. Exupéry (pilot and writer)

This section introduces the most fundamental and most important concepts of SNMP. In chapter 5, a more detailed description is given.

3.2.1 Information Model

In an SNMP MIB, the managed objects are either scalar variables (single valued, not multi valued as vectors or structures) which have only one instance, or tables that can grow dynamically. Every managed object is given a globally unique name using a universal naming tree. A node could for example be named 1.12.4.9.2.3. For convenience, mnemonic names such as `sysUpTime` exist for each object.

Tables are two-dimensional, that is, all elements in a table must be scalar variables. Every column has to be accessed separately, hence the only processable unit in SNMP is a scalar variable. This means that there is no protocol support for handling entire rows in a table, although the SNMP framework defines conventions for this. Actually, there is no support at all in the protocol for table operations. All table operations (foe example, looking up a row with a key or deleting a row) are emulated by simple variables, using a trick in the global naming tree (see section 5.3.1). Hereafter the term *variable* will be used both for ordinary scalar variables and for elements in a table.

The MIB specification language is a subset of the ASN.1 language [X208]. ASN.1 stands for Abstract Syntax Notation One and is an internationally standardized language for defining syntaxes, more on this in section 5.3. A specific MIB is described by all its variables and tables along with their global names. For example, a definition of the managed object `sysLocation` looks like this in ASN.1 (there is no need to understand the details):

```
-- (comment) Example of how the managed object 'sysLocation'
-- is defined in ASN.1.

sysLocation OBJECT-TYPE
    SYNTAX      DisplayString (SIZE (0..255)) -- its data type
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION
           "The physical location of this node (e.g.,
            'telephone closet, 3rd floor')."
    ::= { system 6 }  -- sysLocation's global name is defined
                      -- as the 6th node under the system subtree.
```

In SNMP, managed objects do not have methods. The only operations available are *set variable value* and *get variable value*. The implementation of these operations is called the instrumentation functions.

It is important to note that the variables are conceptual variables, that is, it is not required that a variable has a one-to-one correspondence to a real resource. This approach obviates the need for imperative commands, because any command can be realized by setting a variable which has been specially defined for this purpose. For example, to implement a command to reboot a device, one could provide an integer valued variable `rebootDevice`, which when set to 1 by a manager reboots the device.

Since the only way to represent information dynamically is in two-dimensional tables, information modeling in SNMP is essentially the same as relational database modeling.

3.2.2 Organizational Model

SNMP's organizational model is based on the Manager-Agent model. Scalability is achieved by forming hierarchies.

In addition to the manager and agent roles, there is a third role, called *proxy agent*. An agent which has to communicate with another remote agent in order to access the management information, is called a proxy agent (see figure 5). The benefit is that the manager only knows about the proxy agent, which in turn can distribute its work freely.

Figure 5: *A Proxy Agent*

There are several reasons for using a proxy relationship, for example, when the remote device does not support the transport protocol, or when can be useful to let the proxy agent take care of the administration in order not to burden an already busy device.

3.2.3 Communication Model

The principle for exchanging information in SNMP is by getting or setting variables in the MIB. This is the only way a manager can control the agent. There is also a way to retrieve large quantities of data, for example, a subset of the data in the global naming tree or large parts of a table. This is done by traversing the tree in depth-first order with the `get-bulk` or `get-next` request. This is a very

simple but powerful method. The get-next operation is fundamental in SNMP and is often referred to as *the powerful get-next.*

The complete listing of messages that can be sent in an SNMP PDU:

- *Set-request.* Setting a variable.
- *Get-request.* Getting a variable.
- *Get-next-request.* Given a variable, find next instance in the global naming tree and return its name and value.
- *Get-bulk-request.* Retrieve large amounts of data by depth-first search in tree until a given amount of data is collected. Get-bulk was introduced in SNMPv2 in order to minimize network traffic.
- *Response.* Responds to the requests above.
- *Inform-request.* For sending messages between managers. This request was also introduced in SNMPv2 making it possible to build hierarchies of agent/managers.
- *Trap.* The only way for an agent to initiate a communication. If some extraordinary event occurs in the resource, this is signaled to the manager with a trap.

One PDU can contain multiple requests of the same kind, but different requests must be sent in distinct PDUs.

SNMP has no requirements on the underlying protocol. Usually it runs on top of UDP, but TCP/IP, Appletalk or even the OSI stack is indeed possible. It is up to the SNMP entities to make sure that all messages sent reaches the destination. A deficiency is that traps have 'maybe'-semantics, there is no guarantee that it reaches the manager. This could be solved by using TCP/IP (which is both reliable and connection-oriented) or a specially designed MIB for handling this.

3.2.4 Functional Model

SNMP uses the approach of defining special MIBs for each functional area. For example, one notable deficiency in SNMP was previously the difficulty of monitoring networks as a whole, as opposed to nodes on networks. A quite substantial functional enhancement was achieved by the definition of RMON, the Remote Network Monitoring MIB, which consists of a set of standardized managed objects for collecting different kinds of information from the resources.

3.3 Concepts of OSI management

The alternative framework is a set of standards being developed for use in environments based on Open Systems Interconnection (OSI), known as OSI Systems Management. OSI environments consist of the 7 layer protocol stack.[1] Our description of OSI management is based on the [PSCG] presentation; more detailed descriptions can be found in [HEGE], [SLOM] or [STAL].

3.3.1 Information Model

The information model uses an object-oriented approach to model the resources relevant to management. Hence the managed objects can inherit from each other unlike SNMP's simple objects. Along with the object-oriented model comes support for reuse and information encapsulation as well as the method concept to perform actions on objects.

The managed objects are defined in the object-oriented MIB specification language called Guidelines for Definitions of Managed Objects (GDMO). This language has all conventional object-oriented constructs plus a concept called packages. Packages are a collection of characteristics (that is, attributes, attribute groups, notifications, actions and behavior) and are similar to uninstantiated object classes but cannot participate in inheritance relationships. Furthermore, packages can be conditional, that is, a package is only present in a particular class if certain conditions are met.

3.3.2 Organizational Model

The Manager-Agent model is applicable to OSI management too, but here the roles may be assigned dynamically, in other words, they could in principle change during runtime. As in SNMP, a participant could take on both roles simultaneously.

The organizational model of OSI management has a concept called *domain* (although it is not yet specified in detail). A domain is defined as a grouping of resources formed for executive or structural organizational reasons. For example, a collection of resources may be responsible for the security domain. Normally domains are formed after the functional grouping (see section 3.1.4). If a resource handles functionality from different groups, it will be a member of several domains at the same time.

[1] Physical, link, network, transport, session, presentation and application layer.

3.3.3 Communication Model

In addition to the object methods, there is a higher level service to perform meta-operations on multiple objects. This is called the Common Management Information Service (CMIS). CMIS defines operations to create or delete managed object instances, retrieve or modify attributes on managed object instances and mechanisms for scoping and filtering. CMIS is a connection-oriented service implemented by the Common Management Information Protocol (CMIP) in OSI layer 7.

Scoping selects a set of managed object instances on which operations are performed. It is done by specifying a base object or all sub-trees beneath it in the managed object containment tree. This allows many operations to be requested in one CMIP transfer, as means of improving protocol efficiency.

In combination with scoping, filtering can be used. Filtering permits testing of attribute values for $=$, $<$, $>$, substring or presence and can be combined with logical operators (and, or, not). If an object matches the filter criteria, some operation will be performed on it.

3.3.4 Functional Model

The functional areas identified in OSI management are the same five as mentioned in section 3.1.4 namely: fault, performance, configuration, accounting and security management. These areas are implemented by 15 System Management Functions (SMF[2]). One example from the Configuration Management area is the State management function. This SMF provides general operations for state management of managed objects; in other words, a general state model is specified and a set of operations for controlling the state transitions is defined. Some of the SMF are quite complex functions, which are mostly defined generically for high flexibility [HEGE].

3.4 Discussion

3.4.1 The four Models

In our opinion, this subdivision into four models is adequate since it provides a clear way of describing distinct parts of a network architecture. The division *within* the submodels can sometimes be questioned, especially for the functional model. Earlier we have described why networks of today are of such heterogeneous shape. This implies that it is almost impossible to find all aspects of network management and group them in a concise and logical way.

One problem is the fact that many interactions between the functional areas are conceivable, and therefore exact delimitation is not always possible. The need for interfaces between the functional areas arises, for example when the manager gets

[2]SMF actually refers to *functionality area.*

an error report from the fault management area, and wants to do a configuration to automatically correct it. We see a risk that these interfaces grow too complex, indicating an erroneous functional grouping.

Another pitfall is object-oriented modeling, which can lead to unnecessarily complex solutions if you use inheritance when it is not appropriate. Inheritance should be used to simplify problems, but sometimes a generic base class only makes complicated problems even more complicated. This always happens when the similarities are too small.

In OSI management there is a functional area for configuration management consisting of for example the State management functions (figure 6, from [HEGE] page 124). It consists of a set of functions and a model of how to use them. We think that this model is suspectly complicated. Configuration *can* be this complicated but mostly it is not. It looks like the model was designed with the following thought in mind 'What state model can handle all configuration problems?'. Perhaps this is a too general formulation. This state model might be good if we have a huge switchboard to configure, but for most problems it seems too complex. The use of inheritance or generalization should have stopped at an earlier stage.

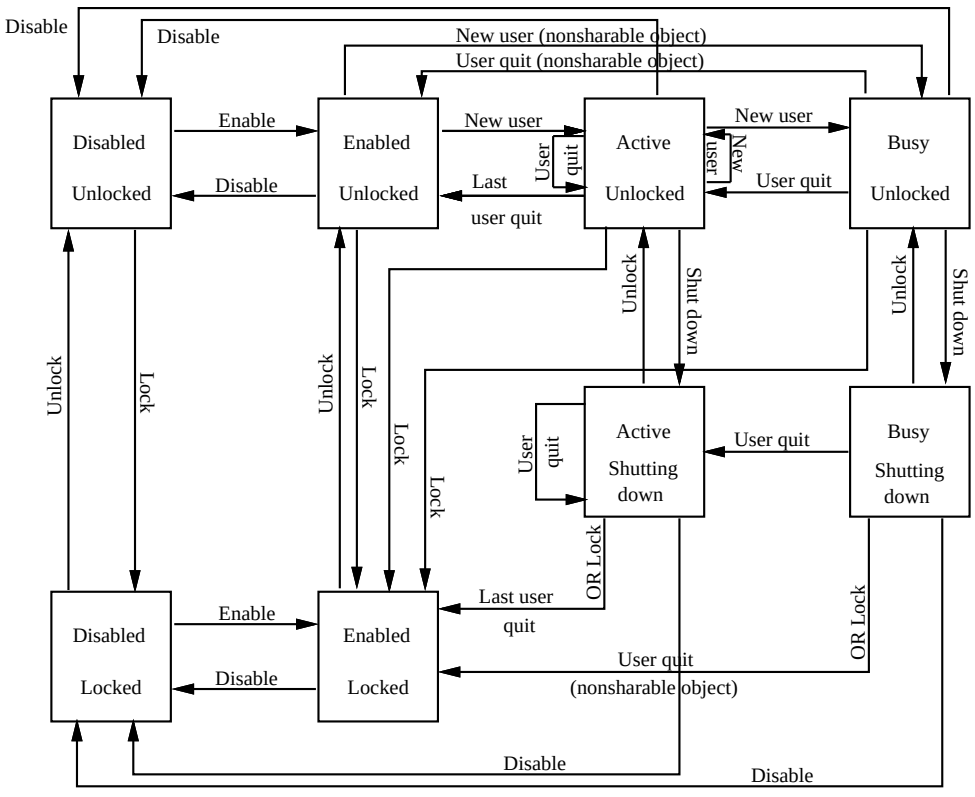

Figure 6: *State Model of OSI Configuration Management*

3.4.2 Requirements on the Information Model

In section 3.1.1 we explained what is required from an information model. When designing a MIB you need help with structuring the information. We must say that OO is more powerful than SNMP's MIB specification language based on ASN.1 macros. An important feature missing in SNMP is the function (or method) concept. Executing a function in SNMP is emulated by setting a variable. This is sufficient if no parameters are needed, but if they are, there is no well-defined solution. One approach would be to define scalar variables that represent the parameters, and let the manager set these first, and then set an 'execute variable'. This introduces the problem of mutual exclusion.[3]

A clear example where the lack of a function concept leads to spaghetti conventions, is SNMP's handling of tables. Of course this is a trade-off between simplicity and functionality. In the implementation phase you want simplicity, but in the design phase you might need a little more than tables and simple variables. It would be nice if the MIB specification language was closer to the MIB design language.[4]

3.4.3 The Future

The two architectures of OSI management and SNMP are designed with completely different approaches. The intention of OSI management is to design a complete solution in a longer perspective, whereas SNMP leans on simplicity. For providers of simple network components SNMP is definitely more attractive, since it is easy to get a low-cost product finished fast. A reason for this is that SNMP's information model with tables and simple variables is closer to hardware than the more complex, object-oriented model of OSI management. Today SNMP forms the basis for the majority of manufacturer-independent solutions, but as networks become larger and OSI more mature, many people ([ROSE1], [HEGE]) think that the network management world will turn to OSI. The reasons for this is its "infinite" scalability and structured support for enterprise management. We ourself, believe that SNMP will gradually evolve as network management grows in complexity (upwards in the pyramid of figure 1). This solution might not be quite as well-structured as OSI management, but it will be the solution to the correct problem. Today we do not yet know what problems management will face in the future.

[3]This can also be solved in SNMP. It is trivial to implement a variable with atomic 'test and set' semantics on a single set-request.

[4]Any kind of data modeling technique.

4 MIB design

There are two common approaches to MIB design, *bottom-up* and *top-down*. The bottom-up approach primarily answers the question "what information is available?". The top-down approach answers the question "what information is needed?". A provider of computer network components usually uses bottom-up MIB design. He simply has some new hardware and maps a managed object to each hardware element, and does not have to worry about the manager application. The top-down approach on the other hand, takes into account how the information in the MIB should be used from within a manager application. The different approaches are appropriate in different situations. For example, if a network whose functionality often is extended or changed,[5] all of the hardware must be available to the manager. This could be accomplished by specifying in the MIB what resources actually are available. In the case of added functionality, only the management application needs to be rewritten, whereas in the case of a top-down designed MIB, the MIB probably would have to be redesigned.

When designing a MIB, it is possible to use the standard modeling techniques such as object-orientation or entity-relationship diagrams (ER-diagrams). Such a model is always possible to translate into a MIB, although the work needed depends on the model used and the MIB specification language. For example, an OO-model fits easily into OSI management, as GDMO is object-oriented, and an ER-diagram fits quite easily into SNMP, as SNMP essentially is a relation database. We will only consider MIB design in SNMP here, but it is worth mentioning that given a description of a MIB in GDMO, there exists an automated procedure to translate it into an SNMP MIB [RFC1442]. Today there are no commercially available high level SNMP MIB design tools.

4.1 Design Example

The support for reuse is important during MIB design. There are three reuse aspects to consider. First, reuse of semantics from previously designed MIBs. SNMP supports this with a construct[6] that allows a user to define new types that maps to one of the simple types, as well as describing the semantics of the new type. The second aspect is direct reuse of managed objects. The support for this is quite limited in SNMP. Mainly this is achieved by using tables. New tables can of course index into old ones. It is also possible to define extensions to a table in SNMP[7] without changing the table being extended. The third aspect is reuse of instrumentation. This has to be taken care of by the programmer.

We will now look at how a MIB is modeled and described directly in SNMP's information model, not using a higher level design technique, although when designing large complicated systems, this would be appropriate.

[5] for example, services in a telecommunications network.

[6] The ASN.1 macro `TEXTUAL CONVENTION`.

[7] The ASN.1 macro `AUGMENTS` allows this.

Let us look at a simple example of how to model a specific situation in SNMP. The example is to enlighten the reuse aspects of MIB design. A well-designed MIB can easily be reused, whereas an improper design rules out all chances of reuse.

Consider a standardized MIB for modems. To keep things simple, suppose the modem has two single-valued attributes, `online` and `speed`, and a table of the phone numbers it knows of. The straightforward solution would be to define the MIB as shown in figure 7.

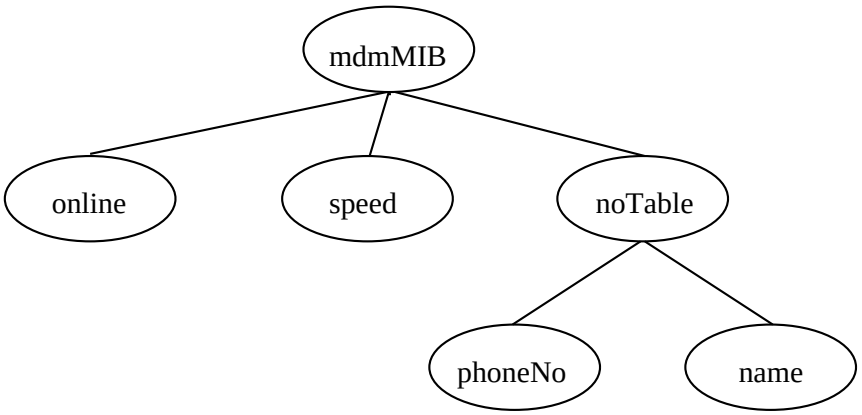

Figure 7: *Simple MIB for modems*

With a MIB like this, we can have an SNMP agent for each modem, and a manager can manage all of them in the same way, by connecting to a specific agent, and everything works fine.

Suppose now that we want to collect many modems in racks, and that we want to have one agent for each rack. Of course we want to reuse the standard MIB for modems shown above. We now have to design a MIB for the rack. In the same way as before, we would say that a rack has one single-valued attribute, `location`, and a table of modems. So each entry in the modem table should have a reference to one modem. But the modem MIB will not allow this, because there can only exist one instance of each scalar variable in the system, as the MIB is static and only can represent one modem. In the example above this was handled by letting one agent control one modem, but now we want one agent to control many modems. So the modem MIB in figure 7 is not appropriate for this situation. The problem resides in the design of the modem MIB. When designing a MIB, you must consider if there possibly could exist multiple instances of the MIB in one agent (normally this is the case). If so, the MIB must be designed for this, using tables, as the only way to dynamically create management information in SNMP is by adding rows to tables[8]. We redesign the modem MIB to provide

[8]The object-oriented approach to this is to allow multiple instances of each managed object class. In this example, the modem would be a managed object class, of which it

a table of modems, instead of just one single modem. The result is shown in figure 8. (In this figure and the following, there is a table within another table. This is not possible to achieve in SNMP directly, but will be implemented as two separate tables, with the first table containing an index into the second table.) In the figure, we have added a field `mdmIndex`. This is used to uniquely identify each modem. We will not consider how the values for the index actually is computed.

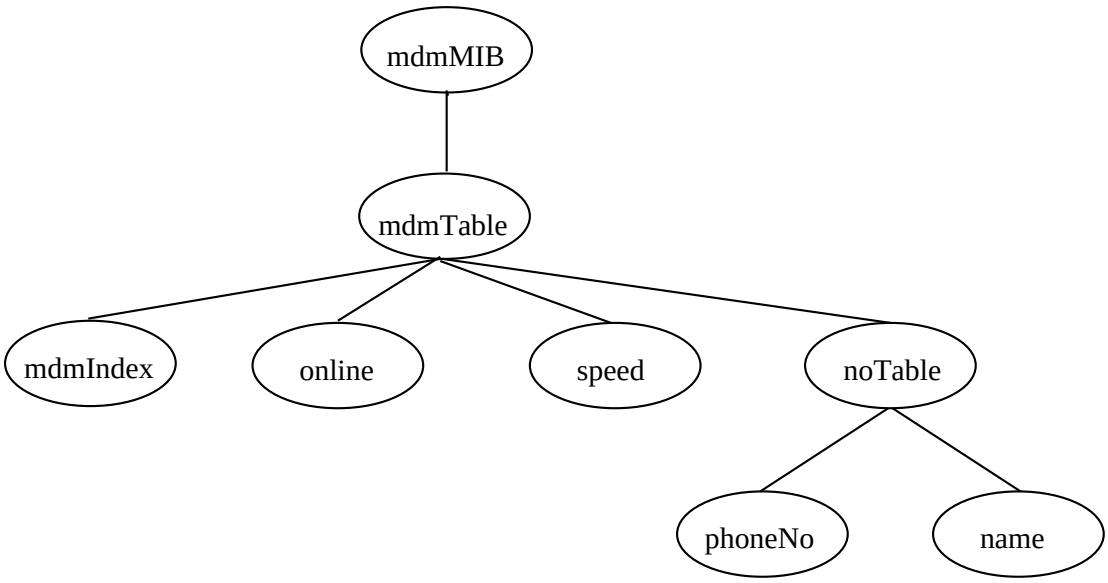

Figure 8: *Improved MIB for modems*

Now we can design our rack MIB, using the standard modem MIB given in figure 8. First, we realize that there could probably exist more than one rack in each system controlled by an agent, so we should make our MIB as a table of racks. Then, each rack has a `rackIndex` (used to identify the rack), a `location` variable, and a table of indexes into the modem table. The rack MIB is shown in figure 9.

With these MIBs, an SNMP manager can control the modems in a rack by first looking in the modem table of the rack, and then using the indexes found to retrieve entries from the modem MIB. This is a flexible solution, because even if there exists only one rack right now, the situation could change, without us having to redesign our MIB.

Another advantage of specifying the MIB as a table, is that it is possible to extend a table in SNMP, without changing the original table. This makes it possible to define a standard MIB for modems, and later define MIBs for enterprise specific modems as extensions to the standard MIB. For example, IBM could define a MIB for their modems, simply saying that an IBM modem is a standard modem,

is possible to create many instances.

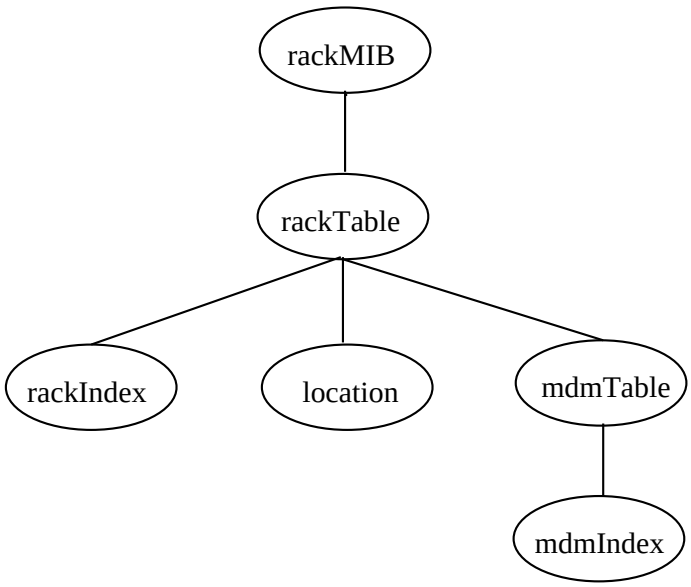

Figure 9: *Simple MIB for racks*

plus some additional features. A manager who is aware of the standard MIB only, could still be able to manage IBM-modems. It is not possible, however, for one agent to implement both the standard modem MIB and more than one extension to it.

5 Description of SNMPv2

This section is intended as a brief introduction to SNMPv2, needed to understand our agent prototype. For total documentation, see [ROSE2].

SNMP is designed to be small and simple, so it can be implemented on simple managed nodes. The protocol only supports basic operations; it is up to the manager application (or the agent) to provide intelligence. Every design decision that seems restrictive falls back on this principle.

5.1 Organizational Model

An SNMP *entity* can act in two distinct roles; a *manager role* and an *agent role*. This makes it is possible to build hierarchical relations between entities. For example, one could have a system with many local networks, each controlled by a manager which reports to a central manager. This central manager will then have a view of the important aspects of the whole system.

There are three central concepts in SNMP. The first one is the *party* concept. This refers to entities communicating via a management protocol and a transport service using authentication and encryption facilities. The second concept is the *context* concept. This refers to a subset of management information. The last concept is the *access policy* concept, which determines the operations that may be performed when one party asks another party to perform some operation on objects in a specific context. All this SNMP-related information is in the *party MIB* defined in [RFC1447], in the form of tables, which each SNMP entity has to implement. This means that SNMP is configured and administrated within SNMP.

5.1.1 Party Concept

All communication in SNMP takes place between two parties. An SNMP party is a logical process in an SNMP entity. Each entity can contain various parties. Each party is associated with three kinds of attributes: *transport* attributes, which define the transport service and transport address being used, *authentication* attributes, which define the authentication protocol and the corresponding data being used, and *privacy* attributes, which define the encryption protocol and the corresponding data being used. The entity must keep information of all local and remote parties known to it, in order to process SNMP messages. This information is held in the party table in the party MIB.

5.1.2 Context Concept

An SNMP context is a collection of management information accessible by a party, held in the context table. A context can either be local or remote. If the context is local, it refers to a *MIB view*. Each MIB view is a collection of subtrees of the MIB, defined in the view table. If it is remote, it defines a *proxy relationship*. Consequently the party has to communicate with a remote party in order to access the management information connected to this context (see section 3.2.2).

5.1.3 Access Policy Concept

An SNMP access policy defines the operations that are allowed on a context, when a party communicates with another party. This information is held in the acl table.

5.1.4 Operations Example

To understand how the party MIB is used, let us look at an example. When a manager wishes to perform some operation (for example, get or set) on some objects, it searches its party table for a party at the agent which meets its requirements of authentication and encryption. This party is called the *destination party*. When found, it determines in which context the objects are visible. Given this, it sends to the agent a possibly encrypted message containing the names of the manager party and the destination party,[9] the context and the objects. When the agent gets this message, it searches the party table for the destination party, and determines if encryption and/or authentication is in use. Next, it consults the access table to find out which operations the manager party is allowed to perform when talking to the destination party, requesting objects in the current context.

5.2 Traps

When an extraordinary event occurs, the agent takes initiative and sends traps to one or more managers to make them aware of the event. When a trap is defined, it is decided which variables in the MIB that will be sent in the trap PDU to the managers. The managers use these variables to diagnose the event. Each trap is given a unique identifier, which is an ASN.1 `OBJECT IDENTIFIER`, see section 5.3. In this way, the trap is viewed as a member of the MIB, so it can be contained within a MIB view.

When the agent decides to send a trap, it looks for a local context that refers to a MIB view which contains the trap. Then it searches the acl table to find the

[9] The destination party is not encrypted, in order for the agent to be able to determine whether to use decryption or not.

entries for this context allowing traps to be sent from a local party to another party. The agent will send traps to all these parties. The agent will then lookup each value for the variables included in the trap definition, and include those values in the trap message being sent.

5.3 ASN.1

SNMP uses ASN.1 for two different purposes: defining the format of the messages being sent, and defining the management information. SNMP uses the *Basic Encoding Rules* (BER) [X209] for a well-defined encoding of the defined types into a stream of bits, in a machine-independent way.

SNMP uses a subset of ASN.1, for example it uses only the four simple types `INTEGER`, `OCTET STRING`, `BIT STRING` and `OBJECT IDENTIFIER`, along with the two constructed types `SEQUENCE`, which is like a "record" or a "structure" in a conventional programming language, and `SEQUENCE OF`, which is a list of another ASN.1 type. `OBJECT IDENTIFIER` is an important type, which has to be understood. An `OBJECT IDENTIFIER` is a sequence of non-negative integers, resulting from traversing a global tree. Each node in the tree has an integer valued label, as well as a symbolic label.

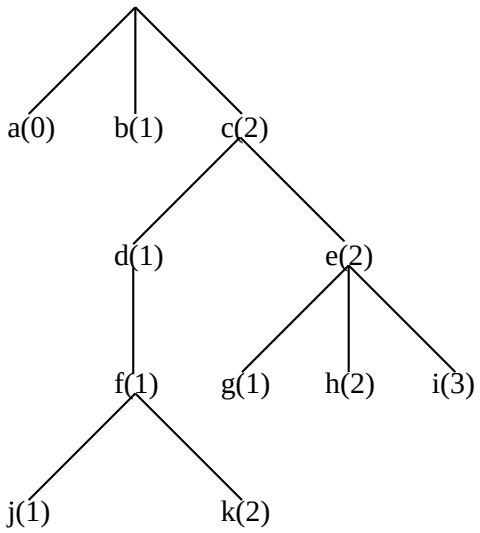

Figure 10: *Naming tree*

For example, in figure 10, the object `g` has `OBJECT IDENTIFIER 2.2.1`. It is also possible to use the symbolic names in the `OBJECT IDENTIFIER`, or to mix symbolic names and integers, for example, `g` could also be named as `2.e.g`. An `OBJECT IDENTIFIER` is used to give a unique name to an object. The object can have any semantics, for example, it could be a document, a managed object or a trap. SNMP uses `OBJECT IDENTIFIER`s to identify variables.

5.3.1 Instance Identification

Recall that there can only be one instance of each scalar variable in SNMP. To identify the instance of a variable, for example, in a get operation, a zero is appended to the `OBJECT IDENTIFIER` defining the variable. For example, suppose the object `g` in figure 10 is a scalar variable, and that we want to refer to the one and only instance of this variable. The `OBJECT IDENTIFIER 2.2.1.0` identifies this instance.

A table has one or more columns as keys, or *indexes*. This means that given values for the keys in a table, the row is uniquely identified. As SNMP only allows manipulation per object (column) in a row, the following trick is used to identify a column in a specific row: First, to the `OBJECT IDENTIFIER` identifying the table, the column number is appended. Second, the values of the keys are appended in order. Consider the following situation. Suppose `f` in figure 10 is a table entry, and that each row in the table consists of the `INTEGER` valued columns `j` and `k`, indexed by `j`. To identify the column `k` in the row in which `j` has value 4, we would specify the `OBJECT IDENTIFIER 2.1.1.2.4`, as the table is `2.1.1`, we want column 2, and the index of the row is 4.

SNMP imposes a *lexicographic ordering* over all object instances (that is, over the `OBJECT IDENTIFIER`s identifying them). With this ordering, it is possible to get all values in a table without knowing the keys for each row. This is taken care of by the `get-next` operator. Given an `OBJECT IDENTIFIER`, `get-next` returns the next (in the lexicographic ordering) object instance, as well as its value. So to retrieve the first column in the first row in the table above, we would issue a `get-next` command, with the `OBJECT IDENTIFIER 2.1.1`. If the row specified above is the only row in the table, we would get `2.1.1.2.4` along with the value of `k` in this row.

6 Our Framework

6.1 General

Our framework provides an environment which supports rapid prototyping of MIBs and testing of MIBs which lacks instrumentation functions (that is, functions for get or set of specific variables). It is also possible to implement a complete SNMP agent. Further, the framework could serve as a basis for building high level tools.

In order to use our program, it is necessary to understand the basics of SNMP, its operations and their semantics, and how MIBs are defined using ASN.1.

To verify that our agent really understands SNMPv2, we have tested it using CMU's SNMPv2 manager.[10]

6.2 Description of the Program

Given a MIB description in ASN.1, and names of instrumentation functions written in ERLANG for the managed objects, our program sets up a running SNMP agent.

A complete example of a MIB in ASN.1 and the corresponding instrumentation functions is given in Appendix A.

6.2.1 MIB Description

For our program to run, the user must create a MIB in ASN.1 format in a text file, and run it through our MIB compiler. The MIB compiler checks the syntax, and produces a file with the MIB in an internal format. This compiled file is read by the agent on startup.

Given a compiled MIB, it is possible to load it into a running agent without restarting the agent. It is also possible to unload a MIB from a running agent.

[10] It can be found at Carnegie-Mellon University, ftp://lancaster.andrew.cmu.edu/

6.2.2 Instrumentation Functions

To actually attach the managed objects with real resources, a user-defined instrumentation function for each variable is needed. This function will be called by the agent on a get or set operation. Such a function could for example read a register on some hardware, do some calculation, or whatever is necessary to implement the semantics associated with the conceptual variable. These functions must be written both for scalar variables and for tables. They are specified in a text file where the `OBJECT IDENTIFIER` for each managed object is associated with an ERLANG tuple `{Module, Function, ExtraArgument}`. When a managed object is referenced in an SNMP operation, the associated `{Module, Function, ExtraArgument}` is looked up, and the function will be applied to some standard arguments (for example, the operation type), and the extra argument supplied by the user.

In order to understand how this works, let us look at how the instrumentation functions should be defined in ERLANG for the different operations. In the following, `RowIndex` is a list of key values for this table, and `Column` is a column number.

Get operation
: For scalar variables:
`variable_access(get, ExtraArg)`
For tables:
`table_access(get, ExtraArg, RowIndex, Column)`
These functions must return the current value of the associated variable.

Set operation
: For scalar variables:
`variable_access(set, ExtraArg, NewValue)`
For tables:
`table_access(set, ExtraArg, RowIndex, <columns>)`
where `<columns>` is a list of tuples `{Column, NewValue}`.
These functions returns `noError` if the assignment was successful, otherwise an error code.

Next operation
: This should only be defined for tables.
`table_access(next, ExtraArg, RestOfOid)`
`RestOfOid` is a (possibly empty) list of integers. It is a list representation of the `OBJECT IDENTIFIER` specified, minus the `OBJECT IDENTIFIER` for the table itself. So if the list is non-empty, the first integer is the column, and the rest is values for the keys. This function should return the lexicographically next instance of a managed object in the table, in the same format as `RestOfOid` (that is, as a list with the first element being the column number, and the rest being the keys for the row).

Note: normally the functions described above behave exactly like this, but they are free to do anything else too. For example, a get-request may have side effects such as setting some other variable, perhaps a global `lastAccessed` variable.

The first two functions, `get` and `set`, have a one to one correspondence to SNMP requests, but the third has not. If the agent gets a `get-next-request`, it will first call the `next` function and then the `get` function. It would be inconvenient for the programmer to have to implement a `get-next` operation when `get` is already implemented. Further, with the choice of these three basic operations, the agent will be able to handle the `get-bulk-request` as well.

In addition to these functions, it is possible to specify a test function, which has the same syntax as the set operation above, except that the first argument is `is_set_ok` instead of `set`. This function will be called before the variable is set, to ensure that it is permissible to set the variable to the new value. For a full description of this function, see section 6.2.3.

The `ExtraArgument` can be used to write generic functions. Consider two read-only variables for a device, `ipAdr` and `name` with object identifiers 1.1.23.4 and 1.1.7. To access these variables, one could implement the two ERLANG functions, `ip_access` and `name_access`, which will be in the MIB. The functions could be specified in a text file as follows:

```
ipAdr = {my_module, ip_access, []}.
-- (comment) Or using the object identifier syntax for 'name':
1.1.7 = {my_module, name_access, []}.
```

Here, the `ExtraArgument`-parameter is the empty list. For example, when the agent receives a get-request for the `ipAdr` variable, a call will be made to `ip_access(get, [])`. The value returned by this function is the answer to the get-request.

If `ip_access` and `name_access` are implemented similar, we could write a `generic_access` function using the `ExtraArgument`:

```
ipAdr = {my_module, generic_access, 'IPADR'}.
-- Using the mnemonic 'name' is more convenient than 1.1.7
name  = {my_module, generic_access, 'NAME'}.
```

When the agent receives the same get-request as above, a call will be made to `generic_access(get, 'IPADR')`.

Yet another possibility, closer to the hardware, could be:

```
ipAdr = {my_module, generic_access, 16#2543}.
name  = {my_module, generic_access, 16#A2B3}.
```

If there is a managed object, scalar variable or table, in the MIB, which does not have a function associated with it, we provide a default function for that object. This default function will store a value for the object in a local database, or if the object is a table, store all data in the table in the database. The object can then be used in all SNMP operations, including the next operation for tables. It is also possible for other ERLANG functions to access this database; for an example of how this can be used, see section 6.2.4. This mechanism is useful for MIB testing and rapid prototyping, as it is not necessary to write all instrumentation functions for the MIB in order to test it with a manager.

6.2.3 Atomic Set

In SNMP, the set operation is atomic. This means that either all variables specified in a set operation are changed, or none. To implement this, the set operation is divided into two phases. The first phase will check that the values supplied for all variables is of the correct type, and within ranges etc. The second phase will then set the values. This approach does not take care of the case when the value supplied is of the right type and within ranges, but still cannot be set at this time, because some other resource depends on this value, or the variable depends on another resource. For example, if a row in a table represents a physical connection, it should not be possible to delete this row if the connection is in use. To be able to handle these cases, it is possible for the user to define an `is_set_ok`-function, which will be called during phase one. Phase two will only be run if all `is_set_ok`-functions returned true.

Still, there could be situations (though rare), where this is not sufficient either, for example, if there are complex relations between many variables, and it won't be sufficient to check each value sequentially. Suppose that we have the variables `month` and `day`. A set request containing both these variables must not allow the combination 'February' and '31'. For these situations, it is possible to define a *consistency-check* function, which will be called with all variable-bindings in the operation. (Actually, the `is_set_ok`-phase described above, is taken care of by a default consistency check function, which will be called if the user doesn't specify his own. So if there is a consistency check function, the `is_set_ok`-phase will not be run, unless the user explicitly calls the `default_consistency_check` function, or the `is_set_ok` functions.)

6.2.4 Default Instrumentation Functions

Sometimes it is useful to store management information in a database. We provide a simple mechanism that can be used for this. For example, suppose a MIB has an integer valued counter, which should be incremented by one each time some external event occurs. If no instrumentation function is provided for this counter, we will use a default instrumentation function, which can handle all requests. The default functions are part of a library with generic functions for

accessing the database. The user can then write a function, which when the external event occurs, reads the variable using the default instrumentation function, increments the value, and use the same function to store the new value. This is all that is needed to make the counter available for the managers. The general situation is shown in figure 11.

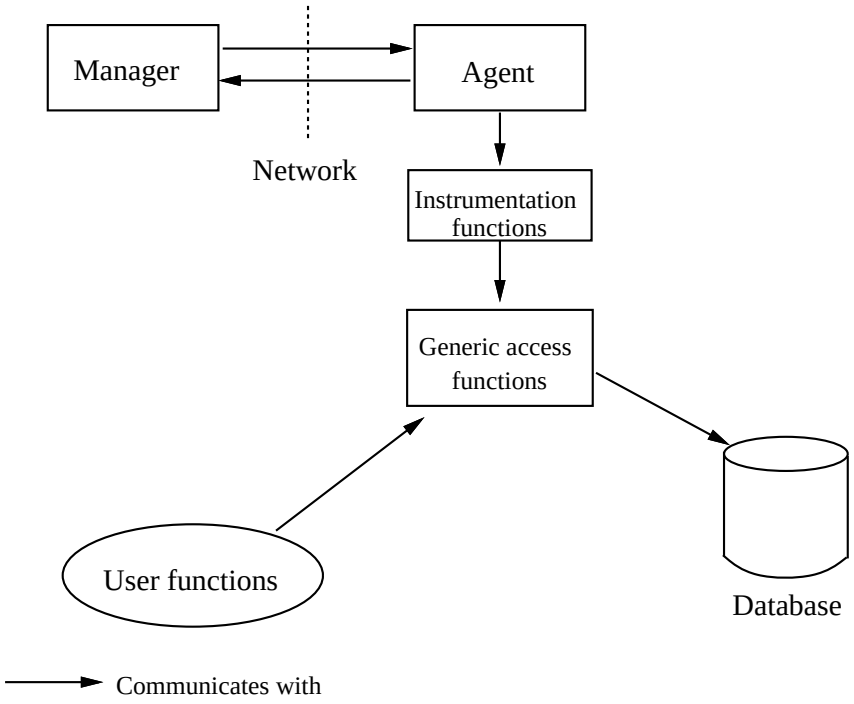

Figure 11: *Communication with local database*

Let us look at a more useful example on how to use the default functions. Suppose that there is a table of cards in a MIB for a rack, where each card is a row in the table. We want to mirror the rack in a database table. The row consists of the following fields:

- `index`, which is used to identify the cards. This corresponds to the slot in the rack where the card resides.
- `rowStatus`. This variable reflects whether the card is active or out of service. It is not possible for a manager to create or delete rows in this table.
- possibly more fields.

Further, suppose that there is some ERLANG process which receives messages from the rack when a card is taken away or placed in the rack. The table should

contain those cards that are actually placed in the rack. In order to change a card's configuration, it must first be taken out of service. This is done by setting the `rowStatus` field to `notInService`. When the manager wishes to put a card into service again, the `rowStatus` variable is set to `active`.

In this situation, it is not possible to use the default functions as is, because an `is_set_ok` function must be written, to check that an active row is not modified. The solution is to write an instrumentation function (and put it in the MIB), which communicates with the generic database access functions. When this instrumentation function is called with the `is_set_ok` parameter, is checks the necessary status and variables. Whenever it is called with the `next` parameter, it just passes the arguments to the default function.

When a card is placed in the rack, an ERLANG process notices this as described above. This process creates a row in the database, using the library functions. Similarly, when a card is taken away from the rack, the ERLANG process notices this, and deletes the row from the database, using the library functions.

The only necessary steps here are to write functions that actually communicate with the hardware. The library functions will take care of the tedious `next`-operator.

Other important usages of the library and the default functions are when making prototypes of large MIBs. It is possible to make a prototype of a MIB without implementing all the instrumentation functions. With the default functions, prototyping can be accomplished in the following way. Firstly, the MIB is written, but no instrumentation functions are implemented. The default functions will be used for all variables and tables. In order to be able to test the agent from the manager, the manager needs useful data. This data could be added with the library functions at the agent. Secondly, instrumentation function are written and tested for one table or variable at a time.

In summary, the mechanism with default and generic library functions provides:

- Effective retrieval of the data.
- Safe storage of the data. This is actually not yet implemented in our framework.
- Functions to create and delete rows in the tables.
- A function that handles next.
- A function that could handle get and/or set. (As long as it is not required to communicate with the resource to implement this.)

6.2.5 Traps

Recall that a trap in SNMP is defined by an `OBJECT IDENTIFIER`, and that when the agent decides that a trap is to be sent, the managers that will receive the trap are deduced from the party MIB. This means that deciding which managers to send a trap to is a configuration issue, but which traps that will be sent is a MIB design issue.

We provide an ERLANG server, called the *trap server*, that takes care of the distribution of the traps. The user must write some ERLANG code that detects that a trap should be sent, and then calls a function that sends a message to the trap server, for example

```
snmp_trap:send_trap(TrapOid)
```

will distribute the trap `TrapOid` to all managers that are currently configured to receive this trap.

6.2.6 Default Configuration

To be able to get a running agent, the party MIB must be initialized. The user have to specify the initial party information in four configuration files, one for each table in the party MIB. To change the party MIB after initialization, at least one manager must have rights to write in the party MIB.

6.2.7 Fault-tolerance

Our program gets input from three different sources; UDP packets from the network, return values from the user-defined instrumentation functions and the MIB in ASN.1 syntax. The first two are fault-tolerant, but the MIB compiler is not. It can handle syntactical errors, but not all semantical. This means that there are some incorrect inputs that the MIB compiler will treat as correct, and this will make the agent process behave strangely. If the MIB compiler is presented with a semantically correct MIB, the agent process is input fault-tolerant. By this we mean that the agent will not crash even if the user defined instrumentation functions crashes or return erroneous values.

6.3 Future Extensions

Before using our implementation of the framework in a real product, there are a few things that should be improved:

- The party data store is entirely in memory. This means that the information in the agent does not survive crashes. It should be backed up on disc.
- Authentication (and maybe encryption) should be implemented.
- The MIB compiler should have better error-handling.
- Some parts of the agent are quite inefficiently implemented, for example PDU encoding and looking up variables in the MIB.
- The definition of SNMPv2 is not yet finished. When SNMPv2 becomes a full standard, there are probably a few minor changes that will have to be done.

7 High Level Tools

We believe that our framework serves as a good basis for building higher level tools. Such tools are necessary when implementing management systems for large and complex networks. In this chapter, different ideas of tools that can be constructed on top of our framework, will be presented.

The framework does not give any help in structuring the design of a new MIB. It simply assumes that there exists a MIB in ASN.1 format. As described in chapter 4, it is convenient if there is some means of abstraction, for example when designing a MIB it is helpful if the tool makes structuring and reuse easier. This could be achieved by defining a better MIB specification language or a graphical tool.

A tool that is built upon our framework, must generate the MIBs in ASN.1 format, as well as the ERLANG instrumentation functions. The latter is not an easy task, because there can be complex relations[11] between tables, and the instrumentation functions must keep the data in the tables consistent. The MIB description will most likely be read by the manager application builder, so it is important that the ASN.1 file is understandable. Particularly, the description fields of each object must be informative.

7.1 Entity Relationship

One example of a higher lever model is the ER-diagram. This technique is often used to model relational databases, so it is probably also suitable for designing MIBs for SNMP. Here we will present a strategy for translating ER to a MIB. This could be implemented as a tool, or the ER technique could be used only during the design phase, and then translated "by hand" into ASN.1. It should also be possible to generate instrumentation functions for a MIB prototype from the ER-diagram.

To exemplify the following arguments, consider the situation in figure 12. The entity `Modem` can exist in multiple instances, which implies that entity `Owner` also will do that. Each entity has one key, the property marked with an asterisk, as well as one other property. Each entity will be translated to a managed object, either a group of scalar variables if the entity exists only in one copy, or a table if it is possible to have multiple instances of the entity. Each 1–1 containment relation is modeled as an extra field in the table or group, which is either a `VariablePointer` or a `RowPointer`[12] that refers directly to the other entity. But if it is a 1–N relation, the contained entities are rows in a table. It is not possible to reference all these rows in one field, so one solution is to define another table, indexed by the container entity's unique key and a `RowPointer` which points to

[11] For example, a row in one table may not be deleted because of information in another table.

[12] These types are `TEXTUAL CONVENTION`s which resolve to `OBJECT IDENTIFIER`s. They are used to reference other managed objects.

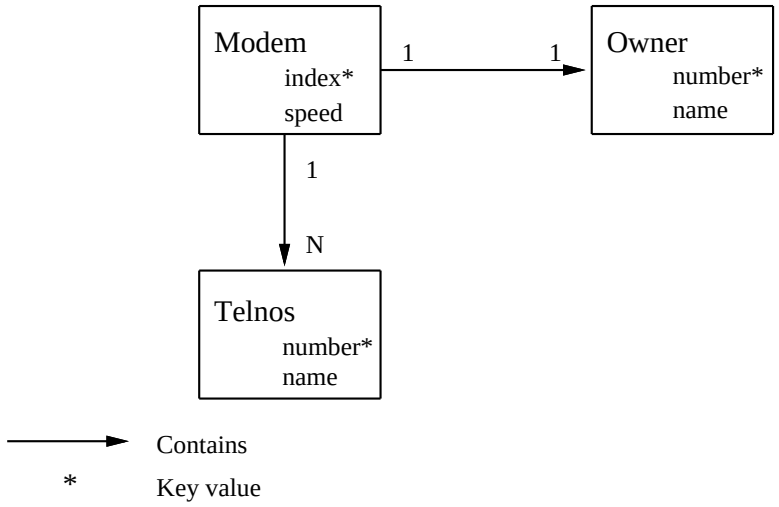

Figure 12: *Example Entity-Relationship diagram*

the contained entity. This means that there will be one row per contained entity in the table. The use of a `RowPointer` (instead of the explicit keys) makes it possible to change the definition of the contained entity without changing the container entity. The resulting MIB is shown in figure 13. The `Modem-Telno` table is used as the extra table for the 1–N relationship between `Modem` and `Telno`. A manager that wishes to get all contained entities from a container entity, can issue a request to get all rows in the extra table, with the first key equal to the container entity's key. In our example, if a manager wishes to retrieve all information about the `Telno` entities contained in the `Modem` entity with `index` 2, it would issue a request to get all fields `TelnoP` in the `Modem-Telno table` with `index` equal to 2. For each such value, it can issue a request to retrieve the corresponding row in the `Telno-table`.

7.2 Object-Orientation

We will briefly consider object-orientation. The reason for wanting OO during design is primarily the support for reuse, encapsulation and the method concept to perform actions on objects. During runtime, polymorphism is an important concept.

These design concepts are possible to translate into an SNMP MIB. A class would be translated into an object group or a table, just as an entity in ER. If a subclass inherits a superclass, all attributes from the superclass would be copied into the subclass. Relations to other classes would be handled as in the ER case described above. Encapsulation comes for free with a table or an object group. Methods would be translated into scalar variables, but as usual, there are problems with parameter passing (see section 3.4.2). Polymorphism is not possible to achieve.

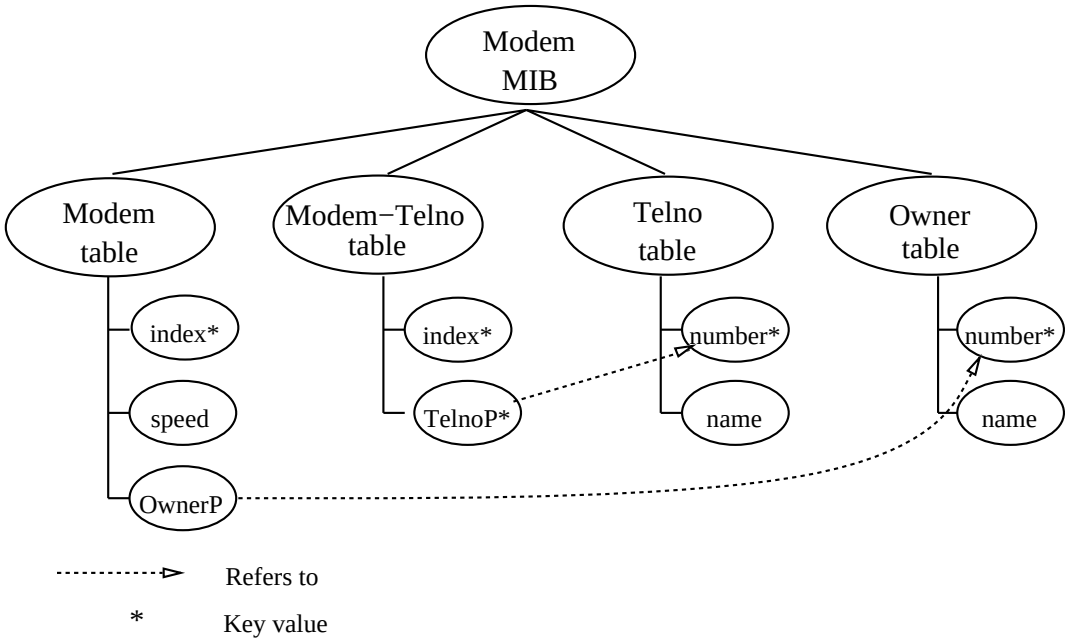

Figure 13: *The MIB corresponding to figure 12*

We do not think that it is a good idea to use OO on top of SNMP, simply because the semantic gap between SNMP and OO is too big.

7.3 Syntactic Extensions

Another useful approach to MIB definition, might be to define syntactic shortcuts for the ASN.1 macros. For example, instead of having to care about how to specify tables in SNMP, it would be nice with a macro `create-table`, which generates the necessary ASN.1 constructs. It might also be nice to provide means of syntactic reuse, in order to lighten the burden of the MIB writer.

8 Conclusions

SNMP's capability of managing large, complex networks such as Telecom networks, depends on how easy is it so divide the problem into subtasks.[13] The following two aspects are important: the degree to which management can be distributed and the support for design of MIBs. In SNMPv2, the `inform-request` is for interchanging information between managers, consequently SNMP fulfills the first aspect. As far as we know, there are no design tools for MIBs today, but our framework is a step in that direction. Since SNMP is based on tables, modeling techniques for relational databases could be adopted. Implementing the framework in ERLANG made it possible to raise the level of abstraction, so that such tools can be implemented.

Moving focus towards design does not imply that implementation experience should be forgotten, but that details should be hidden as long as possible when constructing an agent. Apart from SNMP specific details, the instrumentation functions should be removed during early stages of MIB design. The agent implementor should be able to write them incrementally during prototyping. This will allow the application on the manager side to be developed and tested simultaneously with the agent. The framework implemented can, given only a MIB in ASN.1, generate instrumentation functions for variables as well as tables. Consequently, you get a running prototype agent that can handle set, get, next and table operations without any programming.

It is trivial for a program to generate instrumentation functions for get or set of scalar variables, but it is non-trivial to generate functions for tables. In addition to information available in the MIB (indexes for tables etc.), the program must be given information about relations between the tables. This could be achieved by using Entity Relationship diagrams from which instrumentation functions for a prototype agent are generated. Another possibility is a higher level MIB specification language, where you explicitly specify the tables and their relations.

The framework provides not only prototyping functionality, but also a user friendly environment when implementing a real product. Details, such as type-checking, access rights, PDU encoding/decoding and trap distribution are taken care of by our framework. Left to the agent implementor is the writing of the instrumentation functions, and this can not be avoided. The tedious `get-next` function only has to be implemented for tables and not every variable in the global naming tree. Actually, when mirroring tables in the resource into the software tables of the agent, the `get-next` is handled automatically too.

We succeeded in keeping our source code short. The framework consists of about 7000 lines of ERLANG code. This includes the agent, MIB-compiler, SNMPv2 Party MIB, library functions for handling table operations and finally a simple manager.

[13] Security is another important aspect. This is not discussed here but taken care of in SNMPv2.

Future work is to implement a high level tool for MIB design on top of our framework. In addition, the performance must probably be increased before using it a real product.

9 Abbreviations

ASN.1 Abstract Syntax Notation One
An internationally standardized language for defining syntaxes.

BER Basic Encoding Rules
An internationally standardized mapping of ASN.1 to bits.

CMIP Common Management Information Protocol
An OSI application layer protocol designed to support management.

CMIS Common Management Information Service
The services implemented by CMIP.

ER Entity Relationship
A common technique for information modeling.

GDMO Guidelines for Definitions of Managed Objects
MIB specification language in OSI management.

GSM Global System for Mobile communications

IP Internet Protocol

ISDN Integrated Services Digital Networks

ISO International Standardization Organization

MIB Management Information Base
The conceptual repository for management information.

OO Object-orientation
A paradigm for information modeling as well as programming.

OSI Open Systems Interconnection

PDU Protocol Data Unit

SMF System Management Functions

SNMPv1 Simple Network Management Protocol version 1

SNMPv2 Simple Network Management Protocol version 2

TCP Transmission Control Protocol
A reliable, flow-controlled, in order, two-way transmission of data. Layered above IP.

UDP User Datagram Protocol
A simple, unreliable datagram protocol layered directly above IP.

References

[ARMS] Armstrong J., Virding R., Williams M. 1993. *Concurrent programming in* ERLANG. Prentice Hall.

[BERK] Berkhout V. 1994. *SNMPv2 Simple or Sophisticated?* University of Twente (see [SWEB]).

[EWEB] ERLANG *WWW-page*. http://www-cslab.ericsson.se:5000/

[HEGE] Hegering H-G, Abeck S. 1994. *Integrated Network and System Management*. Addison-Wesley.

[PSCG] The PSC Group. 1993. *OSI Management: Managed Object Modelling & Definition*. 2430 Don Reid Drive, Ottawa, Ontario, K1H 8P5.

[RFC1157] Case, J., M. Fedor, M. Schoffstall, and J. Davin. 1990. *The Simple Network Management Protocol*, RFC 1157. University of Tennessee at Knoxville, Performance Systems International, Performance Systems International, and the MIT Laboratory for Computer Science.

[RFC1442] Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. 1993. *Structure of Management Information for version 2 of the Simple Network Management Protocol (SNMPv2)*. RFC 1442. SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University.

[RFC1443] Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. 1993. *Textual Conventions for version 2 of the the Simple Network Management Protocol (SNMPv2)*. RFC 1443. SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University.

[RFC1444] Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. 1993. *Conformance Statements for version 2 of the the Simple Network Management Protocol (SNMPv2)*. RFC 1444. SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University.

[RFC1445] Galvin, J., and McCloghrie, K. 1993. *Administrative Model for version 2 of the Simple Network Management Protocol (SNMPv2)*. RFC 1445. Trusted Information Systems, Hughes LAN Systems.

[RFC1446] Galvin, J., and McCloghrie, K. 1993. *Security Protocols for version 2 of the Simple Network Management Protocol (SNMPv2)*. RFC 1446. Trusted Information Systems, Hughes LAN Systems.

[RFC1447] McCloghrie, K., and Galvin, J. 1993. *Party MIB for version 2 of the Simple Network Management Protocol (SNMPv2).* RFC 1447. Hughes LAN Systems, Trusted Information Systems.

[RFC1448] Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. 1993. *Protocol Operations for version 2 of the Simple Network Management Protocol (SNMPv2).* RFC 1448. SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University.

[RFC1449] Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. 1993. *Transport Mappings for version 2 of the Simple Network Management Protocol (SNMPv2).* RFC 1449. SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University.

[RFC1450] Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. 1993. *Management Information Base for version 2 of the Simple Network Management Protocol (SNMPv2).* RFC 1450. SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University.

[RFC1451] Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. 1993. *Manager-to-Manager Management Information Base.* RFC 1451. SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University.

[RFC1452] Case, J., McCloghrie, K., Rose, M., and Waldbusser, S. 1993. *Coexistence between version 1 and version 2 of the Internet-standard Network Management Framework.* RFC 1452. SNMP Research, Inc., Hughes LAN Systems, Dover Beach Consulting, Inc., Carnegie Mellon University.

[ROSE1] Rose, M.T. 1991. *The Simple Book - An Introduction to Internet Management*, Prentice-Hall.

[ROSE2] Rose, M.T. 1994. *The Simple Book - An Introduction to Internet Management*, Prentice-Hall.

[RUTT] Rutt, T. 1994. *Comparison of the OSI management, OMG and Internet management Object Models.* A report of the Joint XOpen/NM Forum Inter-Domain Management Task force. Email: t.rutt@att.com

[SCHE] Schekkerman E.J. 1993. *An Analysis of the Simple Network Management Protocol version 2* M.Sc. Thesis (see [SWEB]).

[SLOM] Sloman, M. 1994. *Network and Distributed Systems Management.* Addison-Wesley.

[STAL] Stallings, W. 1993. *Network Management.* IEEE Computer Society Press.

[STEE] Steedman, D. 1990. *Abstract Syntax Notation One (ASN.1): The Tutorial and Reference.* Technology Appraisals.

[SWEB] *The Simple Web - snmp info site.* http://snmp.cs.utwente.nl/

[X208] *Specification of Abstract Syntax Notation One.* 1987. CCITT recommendation X.208. Geneva, Switzerland.

[X209] *Specification of basic encoding rules (BER) for Abstract Syntax Notation One.* 1987. CCITT recommendation X.209. Geneva, Switzerland.

A Example MODEM-MIB

In this Appendix, we show a complete example of a MIB and its instrumentation functions written in ERLANG. The MIB is the modem MIB shown in figure 8 in chapter 4.

For a detailed description of the instrumentation functions, see the user-manual for our framework.

A.1 The MIB in ASN.1

```
MODEM-MIB DEFINITIONS ::= BEGIN

IMPORTS
      MODULE-IDENTITY, OBJECT-TYPE,
      snmpModules, UInteger32
   FROM SNMPv2-SMI
      TEXTUAL-CONVENTION, RowStatus,
      DisplayString, TruthValue
   FROM SNMPv2-TC;

modemMIB MODULE-IDENTITY
   LAST-UPDATED "9505040000Z"
   ORGANIZATION "SU"
   CONTACT-INFO
     "(d90-mbj,d90-ker)@nada.kth.se "
   DESCRIPTION
     "Example of modem-MIB."
   ::= { iso 12 }

-- textual conventions (that is, abstract datatypes)

ModemSpeed ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
              "300/1200/2400 or 9600 baud."
    SYNTAX       INTEGER

PhoneNbr ::= TEXTUAL-CONVENTION
    STATUS       current
    DESCRIPTION
              "A telephone number."
    SYNTAX       OCTET STRING

modemMibObjects
               OBJECT IDENTIFIER ::= { modemMIB 1 }
```

```
modemContact OBJECT-TYPE
    SYNTAX      DisplayString
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION
          "Contact info."
    ::= {modemMibObjects 1}

modemTable OBJECT-TYPE
    SYNTAX      SEQUENCE OF ModemEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
            "Modem Table"
    ::= { modemMibObjects 2 }

modemEntry OBJECT-TYPE
    SYNTAX      ModemEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
            "The entry in the table"
    INDEX      { modemPort }
    ::= { modemTable 1 }

-- The modemTable consists of the following columns
ModemEntry ::=
    SEQUENCE {
        modemPort            INTEGER,
        modemOnline          TruthValue,
        modemSpeed           ModemSpeed,
        modemStatus          RowStatus
    }

modemPort OBJECT-TYPE
    SYNTAX      INTEGER
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
         "The physical port where the modem hangs."
    ::= { modemEntry 1 }

modemOnline OBJECT-TYPE
    SYNTAX      TruthValue
    MAX-ACCESS  read-write
```

```
    STATUS      current
    DESCRIPTION
      "true = online, false = offline."
    DEFVAL      { true }
    ::= { modemEntry 2 }

modemSpeed OBJECT-TYPE
    SYNTAX      ModemSpeed
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION
         "The speed of this modem."
    ::= { modemEntry 3 }

modemStatus OBJECT-TYPE
    SYNTAX      RowStatus
    MAX-ACCESS  read-create
    STATUS      current
    DESCRIPTION
         "The special variable to emulate table
          operations, createRow, deleteRow, ..."
    ::= { modemEntry 4 }


-- The Modem-Telno-Table:

modemTelnoTable OBJECT-TYPE
    SYNTAX      SEQUENCE OF ModemTelnoEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
     "Modem-Telno Table."
    ::= { modemMibObjects 3 }

modemTelnoEntry OBJECT-TYPE
    SYNTAX      ModemTelnoEntry
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
            "The entry in the table"
    INDEX      { modemTelnoModemPort,
                 modemTelnoPhoneNbr }
    ::= { modemTelnoTable 1 }

ModemTelnoEntry ::=
    SEQUENCE {
```

```
        modemTelnoModemPort     INTEGER,
        modemTelnoPhoneNbr      PhoneNbr,
        modemTelnoPhoneName     DisplayString,
        modemTelnoStatus        RowStatus
    }

modemTelnoModemPort OBJECT-TYPE
    SYNTAX      INTEGER
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
         "The modem port uniquely defining a modem."
    ::= { modemTelnoEntry 1 }

modemTelnoPhoneNbr OBJECT-TYPE
    SYNTAX      PhoneNbr
    MAX-ACCESS  not-accessible
    STATUS      current
    DESCRIPTION
         "A phoneNbr in the phonebook for
          this modem."
    ::= { modemTelnoEntry 2 }

modemTelnoPhoneName OBJECT-TYPE
    SYNTAX      DisplayString
    MAX-ACCESS  read-write
    STATUS      current
    DESCRIPTION
         "The name of the phone number."
    ::= { modemTelnoEntry 3 }

modemTelnoStatus OBJECT-TYPE
    SYNTAX      RowStatus
    MAX-ACCESS  read-create
    STATUS      current
    DESCRIPTION
          "Only operations 'createAndGo' and 'destroy'
          are implemented. The only valid readable value
          is 'active'."
    ::= { modemTelnoEntry 4 }

END
```

A.2 MIB implementation

Here we present instrumentation functions for the modem MIB. These functions use the generic library functions in module `snmp_gfsd` to implement the tables in agent software. The module `modem` contains functions for communicating with the modems.

A.2.1 Prototype implementation

First, we present a *prototype* implementation of the MIB, that is, all data entirely exists in software, and there is no connection at all to the hardware (the physical modems). We cannot use the default functions directly, because there are relations between the two tables that must be maintained.

When a manager tries to add a row in the modem table, the default function `snmp_gfsd:table_func(is_set_ok,...)` will check that the row does not already exist. If the manager tries to delete a row, we must make sure that it does not exist any telephone numbers in the modemTelno table. If the `is_set_ok`-phase succeeded, the default function `snmp_gfsd:table_func(set,...)` will change the software table.

When a manager tries to add a row in the modemTelno table, we must check that the modem exists.

```
-module(modemmib).
-author('(d90-mbj,d90-ker)@nada.kth.se').
-export([init/0,
         modem_table/4, modem_table/3,
         modem_telno_table/4, modem_telno_table/3,
         try_change_modem_status/4,
         try_change_modem_telno_status/4]).
-include("snmp_party.h").

%%%-------------------------------------------------
%%% This file contains the instrumentation functions
%%% for the modemmib.
%%% PROTOTYPE implementation.
%%%-------------------------------------------------
%% Defines useful columns.
-define(modem_port_col, 1).
-define(modem_status_col, 4).
-define(modem_telno_modem_port_col, 1).
-define(modem_telno_status_col, 4).

init() ->
    snmp_gfsd:table_create(modemTelnoTable),
    snmp_gfsd:table_create(modemTable).
```

```
%%-------------------------------------------------
%% The modem_table
%%-------------------------------------------------
modem_table(get, [], RowIndex, Col) ->
    snmp_gfsd:table_func(get, modemTable, RowIndex, Col);

modem_table(is_set_ok, [], RowIndex, Cols) ->
    snmp_gfsd:table_try_row(modemTable,
    {modemmib, try_change_modem_status},
    RowIndex, Cols);

modem_table(set, [], RowIndex, Cols) ->
    snmp_gfsd:table_set_row(modemTable, nofunc, nofunc, RowIndex, Cols).

modem_table(next, [], RestOid) ->
    snmp_gfsd:table_func(next, modemTable, RestOid).

%%-------------------------------------------------
%% Called by is_set_ok if RowStatus is changed.
%%-------------------------------------------------
%% If status is 'destroy', we must check to see
%% that there don't exist any any telnos.
%% Cols      is a list of {ColumnNumber, NewValue}
%%-------------------------------------------------
try_change_modem_status(_, ?destroy, [ModemPort], _Cols) ->
    case snmp_gfsd:table_find(modemTelnoTable,
      ?modem_telno_modem_port_col,
      ModemPort) of
        false ->     {noError, 0};
        _FoundRow -> {inconsistentValue, ?modem_status_col}
    end;

try_change_modem_status(_,_,_,_) -> {noError, 0}.

%%-------------------------------------------------
%% The modem_telno_table.
%%-------------------------------------------------
modem_telno_table(get, [], RowIndex, Col) ->
    snmp_gfsd:table_func(get, modemTelnoTable, RowIndex, Col);

modem_telno_table(is_set_ok, [], RowIndex, Cols) ->
    snmp_gfsd:table_try_row(modemTelnoTable,
    {modemmib, try_change_modem_telno_status},
    RowIndex, Cols);
```

```
modem_telno_table(set, [], RowIndex, Cols) ->
    snmp_gfsd:table_set_row(modemTelnoTable, nofunc, nofunc, RowIndex, Cols).

modem_telno_table(next, [], RestOid) ->
    snmp_gfsd:table_func(next, modemTelnoTable, RestOid).

%%-------------------------------------------------
%% Called by is_set_ok if RowStatus is changed.
%%-------------------------------------------------
%% If status is 'createAndGo', we must check to see
%% that the modem exists.
%%-------------------------------------------------
try_change_modem_telno_status(_,?createAndGo,
                              [ModemPort | PhoneNbr],
                              _Cols) ->
    case snmp_gfsd:table_find(modemTable, ?modem_port_col, ModemPort) of
        false ->     {inconsistentValue, ?modem_telno_status_col};
        _FoundRow -> {noError, 0}
    end;

try_change_modem_telno_status(_,_,_,_,_) -> {noError, 0}.
```

A.2.2 Real implementation

When we have implemented and tested the prototype, we can start to implement the "real" instrumentation functions. We will show two different ways of doing this, one for each table. The modem table will be a software table which mirrors the actual hardware, but the modemTelno (which probably will contain more data) is not mirrored, and will only exist in the modems.

When a manager tries to add a row in the modem table, we use the default function `snmp_gfsd:table_func(is_set_ok,...)` to check that the row does not already exist. Also, we check that there is a modem connected to the specified port. If the `is_set_ok`-phase succeeded, the default function `snmp_gfsd:table_func(set,...)` is used to change the software table. When a row is added to the modem table, the modem must be initialized, and when it is deleted, it must be shut down.

As the modemTelno table does not exist in software, we cannot use the default functions as with the modem table. However, there are useful library functions for other purposes, and we will use one of these which handles the list of columns sent to the `is_set_ok`- and `set`-functions.

When a manager tries to add a row in the modemTelno table, we must check that the modem exists. If the `is_set_ok`-phase succeeded, we must tell the modem of the new number, and if a number is deleted, we must inform the modem.

```
-module(modemmib).
-author('(d90-mbj,d90-ker)@nada.kth.se').
-export([init/0,
 modem_contact/2, modem_contact/3,
         modem_table/4, modem_table/3,
         modem_telno_table/4, modem_telno_table/3,
         try_change_modem_status/4,
 set_modem_attributes/2,
         changed_modem_status/4]).
-include("snmp_party.h").

%%%-----------------------------------------------
%%% This file contains the instrumentation functions
%%% for the modemmib.
%%%-----------------------------------------------
%% Defines useful columns.
-define(modem_port_col, 1).
-define(modem_online_col, 2).
-define(modem_speed_col, 3).
-define(modem_status_col, 4).
-define(modem_telno_modem_port_col, 1).
-define(modem_telno_name_col, 3).
-define(modem_telno_status_col, 4).

init() ->
    snmp_gfsd:table_create(modemTable).

%%-------------------------------------------------
%% The modemContact variable.
%%-------------------------------------------------
modem_contact(get, []) ->
    modem:get_contact().

modem_contact(set, [], NewVal) ->
    modem:set_contact(NewVal).

%%-------------------------------------------------
%% The modem_table
%%-------------------------------------------------
modem_table(get, [], RowIndex, Col) ->
    snmp_gfsd:table_func(get, modemTable, RowIndex, Col);

modem_table(is_set_ok, [], RowIndex, Cols) ->
    snmp_gfsd:table_try_row(modemTable,
    {modemmib, try_change_modem_status},
    RowIndex, Cols);
```

```
modem_table(set, [], RowIndex, Cols) ->
    case snmp_gfsd:table_set_row(modemTable,
    {modemmib, changed_modem_status},
    {snmp_gfsd, table_try_make_consistent},
    RowIndex, Cols) of
{noError, 0} ->
    set_modem_attributes(RowIndex, Cols);
Error -> Error
    end.

modem_table(next, [], RestOid) ->
   snmp_gfsd:table_func(next, modemTable, RestOid).

%%-------------------------------------------------
%% Called by is_set_ok if RowStatus is changed.
%%-------------------------------------------------
%% If status is 'destroy', we must check to see
%% that there don't exist any telnos.
%% Cols is a list of {ColumnNumber, NewValue}
%%-------------------------------------------------
try_change_modem_status(_, ?destroy, [ModemPort], _Cols) ->
    case modem:get_all_numbers(ModemPort) of
        [] ->        {noError, 0};
        _FoundRow -> {inconsistentValue, ?modem_status_col}
    end;

%%-------------------------------------------------
%% If status is 'createAndGo' we must check that
%% there is a modem connected to the specified port.
%%-------------------------------------------------
try_change_modem_status(_, ?createAndGo, [ModemPort], _Cols) ->
    case modem:is_modem_connected(ModemPort) of
        true ->  {noError, 0};
        false -> {inconsistentValue, ?modem_status_col}
    end;

try_change_modem_status(_,_,_,_) -> {noError, 0}.

%%-------------------------------------------------
%% Called by set if RowStatus is changed.
%%-------------------------------------------------
changed_modem_status(_, ?destroy, [ModemPort], _Cols) ->
    modem:shut_down_modem(ModemPort),
    {noError, 0};
```

```
changed_modem_status(_, ?createAndGo, [ModemPort], _Cols) ->
    modem:initiate_modem(ModemPort),
    {noError, 0};

changed_modem_status(_, _,_,_) -> {noError, 0}.

%%-------------------------------------------------
%% Called when all values are set.
%% We must 'write through' to the modems, if the
%% attributes (Online, Speed) are changed.
%%-------------------------------------------------
set_modem_attributes([ModemPort], Cols) ->
    case snmp_gfsd:find_col(?modem_telno_status_col, Cols) of
{value, ?destroy} -> {noError, 0};
Else ->
    {value, Online} =
modem_table(get, [], [ModemPort], ?modem_online_col),
    {value, Speed} =
modem_table(get, [], [ModemPort], ?modem_speed_col),
    modem:set_attribute(online, ModemPort, Online),
    modem:set_attribute(speed, ModemPort, Speed),
    {noError, 0}
    end.

%%-------------------------------------------------
%% The modem_telno_table.
%%-------------------------------------------------
%% A get on the status col returns 'active' if the
%% row exists.
%%-------------------------------------------------
modem_telno_table(get, [], [ModemPort | PhoneNbr], Col) ->
    case modem:is_modem_connected(ModemPort) of
        false -> {noValue, noSuchInstance};
        true ->
            case modem:get_phone_nbr_entry(ModemPort, PhoneNbr) of
                {value, Entry} when Col == ?modem_telno_status_col ->
                    {value, ?active};
                {value, Entry} -> {value, element(Col, Entry)};
                Error -> {noValue, Error}
            end
    end;

%%-------------------------------------------------
%% If status is 'createAndGo' or 'destroy', we must
%% check that the modem is connected to the port, and
%% that the number doesn't resp. does exist.
```

```
%% We only implement 'createAndGo' and 'destroy'.
%%----------------------------------------------
modem_telno_table(is_set_ok, [], [ModemPort|PhoneNbr], Cols) ->
    case snmp_gfsd:find_col(?modem_telno_status_col, Cols) of
        {value, ?createAndGo} -> nbr_not_exists(ModemPort, PhoneNbr);
        {value, ?destroy} -> nbr_exists(ModemPort, PhoneNbr);
        {value, _Val} -> {inconsistentValue, ?modem_telno_status_col};
        _Else -> {noError, 0}
    end;

%%----------------------------------------------
%% If status is 'createAndGo', we must add the number
%% to the modem. If it is 'destroy', we must delete
%% the number.
%% If status is not modified, the name must be changed,
%% as it is the only accesible column, except for status.
%%----------------------------------------------
modem_telno_table(set, [], [ModemPort | PhoneNbr], Cols) ->
    case snmp_gfsd:find_col(?modem_telno_status_col, Cols) of
        {value, ?createAndGo} ->
    {value, NewName} =
snmp_gfsd:find_col(?modem_telno_name_col, Cols),
            modem:add_number(ModemPort, PhoneNbr, NewName);
        {value, ?destroy} ->
            modem:delete_number(ModemPort, PhoneNbr);
        Else ->
            change_name(Cols, ModemPort, PhoneNbr),
            {noError, 0}
    end.

%%----------------------------------------------
%% Rather naive implementation of next.
%% modem:get_all_numbers() returns a sorted list of
%% tuples {ModemPort, PhoneNbr, Name}.
%%----------------------------------------------
modem_telno_table(next, [], []) ->
    modem_telno_table(next, [], [?modem_telno_name_col]);

modem_telno_table(next, [], [Col | Index]) ->
    Nums = modem:get_all_numbers(),
    find_next(Nums, Col, Index).

find_next([], _Col, _Index) -> endOfTable;
find_next(Nums, Col, Index) ->
    case find_next_number(Nums, Index) of
        endOfList when Col == ?modem_telno_status_col ->
```

```
            endOfTable;
        endOfList ->
            {FirstModemPort, FirstPhoneNbr, _Name}  = hd(Nums),
            FirstCol = max(Col+1, ?modem_telno_name_col),
            [FirstCol, FirstModemPort | FirstPhoneNbr];
        {ModemPort, PhoneNbr} -> [Col, ModemPort |  PhoneNbr]
    end.

find_next_number([], _Index) -> endOfList;
find_next_number([{Port, PhoneNbr, _Name} | Nums], Index)
       when [Port | PhoneNbr] > Index ->
    {Port, PhoneNbr};
find_next_number([_Num | Nums], Index) ->
    find_next_number(Nums, Index).

change_name([{?modem_telno_name_col, NewName}],
              ModemPort, PhoneNbr) ->
   modem:change_phone_nbr_entry(ModemPort, PhoneNbr, name, NewName).

%% noError if row does not exist.
nbr_not_exists(ModemPort, PhoneNbr) ->
    case modem:is_modem_connected(ModemPort) of
        false -> {inconsistentValue, ?modem_telno_status_col};
        true  ->
             case modem:get_phone_nbr_entry(ModemPort, PhoneNbr) of
                 {value, Entry} -> {inconsistentValue, ?modem_telno_status_col};
                 Error -> {noError, 0}
             end
    end.

%% noError if row does exist.
nbr_exists(ModemPort, PhoneNbr) ->
    case modem:is_modem_connected(ModemPort) of
        false -> {inconsistentValue, ?modem_telno_status_col};
        true  ->
             case modem:get_phone_nbr_entry(ModemPort, PhoneNbr) of
                 {value, Entry} -> {noError, 0};
                 Error -> {inconsistentValue, ?modem_telno_status_col}
             end
    end.

max(X,Y) when X > Y -> X;
max(X,Y) -> Y.
```

A.3 Association file

This is the association file for the prototype implementation:

```
-- Defines instrumentation functions for the tables
-- in the MODEM-MIB.

modemTable      = {modemmib, modem_table,[]}.
modemTelnoTable = {modemmib, modem_telno_table,[]}.
```

This is for the real implementation:

```
-- Defines instrumentation functions for the tables
-- in the MODEM-MIB.

modemTable      = {modemmib, modem_table,[]}.
modemTelnoTable = {modemmib, modem_telno_table,[]}.
modemContact    = {modemmib, modem_contact, []}.
```