Automated Testing of SNMP Controlled Equipment

by Martin Gunnarsson

a Master's project report
at KTH Department of Teleinformatics,
performed at Ericsson Telecom AB

Abstract

This report describes the work I have performed during my Master's project at Ericsson Telecom AB, Älvsjö. The purpose of the project, entitled "Automated Testing of SNMP Controlled Equipment", has been to investigate the possibilities of performing automated tests on network devices using SNMP (Simple Network Management Protocol). The particular device of interest was the AXD301, an ATM switch developed by Ericsson Telecom. The tasks included the design of a testing methodology, i.e. a model of an SNMP testing platform, and the implementation of a test tool prototype.

The first part of the report provides the reader with a theoretical background to network management in general, and the SNMP framework in particular. I also briefly describe testing procedures and automated testing issues. The second part describes the actual work I have done. A model for automated SNMP testing is presented, as well as an implementation of a SNMP test tool prototype.

Contents

1. Project description

This project was done at the Datacom Networks & IP Services unit at Ericsson Telecom in Älvsjö. The System Switches department is concerned with the development of software and hardware for ATM switches and IP switching equipment. The purpose of the project was to investigate the possibilities of performing automated tests of network units equipped with SNMP agents. The primary system to test was the Ericsson AXD301 ATM switching system.

1.1 Problem definition

Today it is possible to test the AXD301 using an SNMP interface, but this requires the engagement of a human operator. A software based tool that allows for automation of these tests and is able to compare the expected results to the actual ones, would certainly speed up the testing process. As for now, any modification of the system requires the same tests to be run once more, a tedious task that takes several weeks. Many of these tests should be suitable for automation, reducing the testing time to a few days.

Several of the tests requires certain kinds of stimuli being injected into the system. For example, one might want to study the performance of the system under pressure, generating simulated traffic while using SNMP to monitor the system. Other possible stimuli could be things like accessing the system through a device processor interface, configuring the hardware in order to generate alarms which, if the system behaves as expected, should be sent as trap messages via SNMP.

1.2 Goals

The primary goals of this project was to:

- Develop a test methodology for automated SNMP testing.
- Implement a prototype of an automated SNMP test tool.

The focus of the project should be at the first of these two parts.

The test methodology and the prototype should support automated tests on a system under test through an SNMP interface, while concurrently generating different kinds of stimuli. Also, functionality for evaluating the test results, based on expected results, should be provided.

An important part in the design of the test methodology was the specification of the tests. A command language should be presented, describing variables to be monitored and configured, stimuli to provide and what result to expect.

2. Network management

2.1 Network management basics

The International Organization for Standardisation (ISO) defines five major functional areas of network management (see [1] and [13]):

- *Fault management*: Detection, isolation and correction of faults in network components.
- *Accounting management*: Administrating charging for use of managed objects.
- *Configuration management*: Maintaining relationships between network components and administrating initializing and shut-down of parts or entire network.
- *Performance management*: Monitoring net performance in terms of utilization, throughputs etc., and adjusting network resources to improve network performance.
- *Security management*: Administrating network access, encryption keys and information protection as well as logs.

Basically, the tasks of a network management system can be divided into two parts: network monitoring and network control. Each of these parts can be applied to the areas listed above, as will be described in the subsequent section. Before moving on though, a couple of concepts need to be introduced.

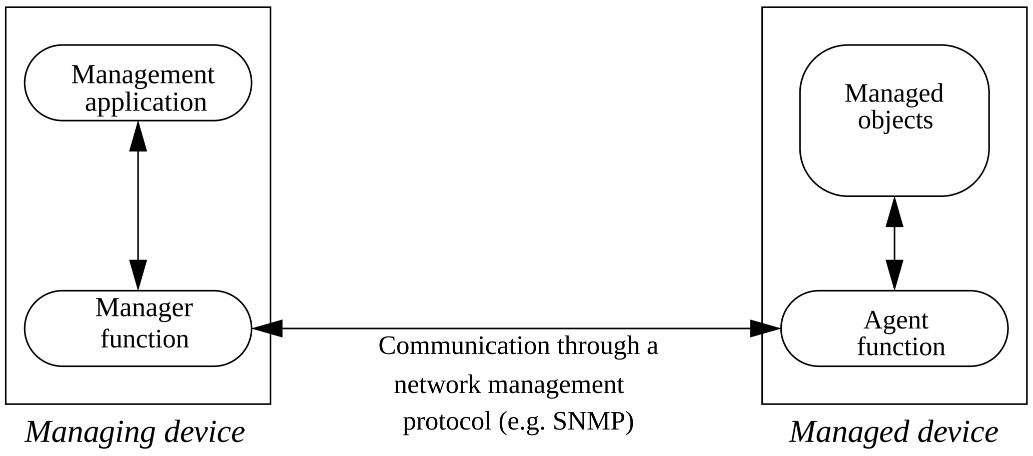


Fig. 2.1: Typical network management model

Fig. 2.1 shows a conceptual model of a network entity being managed and an entity managing it. The managed entity is equipped with an agent function (hereby referred to as the *agent*) and a number of objects being managed. The managing entity has a corresponding manager function (hereby referred to as the *manager*) and an application using the manager for monitoring and controlling of the managed network entities.

The information to be monitored by a manager can be categorized as follows [1]:

- Static: characteristic data for network entities (for instance, the MAC address of a router’s interface).
- Dynamic: information on the state of a system (for instance, that a board is being initialized).
- Statistical: information on what has happened to a system during a certain time interval (for instance, the average number of packets transmitted by an interface).

Information to be monitored can be supplied to the manager in one of two ways. A manager can produce an explicit request for information from an agent residing on a network entity, a technique called *polling*. Using *event reporting*, an agent can initiate contact with the manager to send information, for instance when an unusual event or a fault has occurred. Either one of these techniques is applicable and they can be combined. Which one to use depends on the type of application, as well as factors such as demand for low management traffic, demand for reliability and robustness.

2.2 Network management areas

Fault management

The primary requirements for a fault management system is detection and reporting of faults, as well as maintaining logs of errors and unusual events. Especially in the case of a polling-based system, logs are essential sources of information to the manager. In the case of systems where agents provides fault information to the manager through event reporting, it is important to have reasonable high thresholds for fault definition. If the criteria for fault reporting are too generous, this can result in overloading the network with fault reports.

Another desirable functionality of a fault management system is the ability to anticipate faults. This can be achieved by setting thresholds for certain measured values at the agents, and generate a fault report when these thresholds are exceeded or fallen below.

After a fault has been detected, the manager should have means of isolating (i.e. deciding which part of the network is responsible for the fault) and diagnosing the fault.

Accounting management

The area of accounting management primarily covers measuring the usage (and computing the cost of usage) of network resources per user. Especially in the case of a public service system, there is a need for managing the billing of users. Examples of resources being used are:

- Communication facilities (such as lines and networks)
- Computer hardware
- Software
- Services

The accounting information collected can be things such as user identity, number of packets sent, resources used, etc.

Configuration management

The major responsibilities of configuration management is initializing, maintaining and shutting down components of a system or a subsystem, in short “setting up the network”. It should implement the ability to perform on-line modification of network resources, without having to take the entire network down.

Configuration management is also used as the reactive part of other management functions. For example, if a fault is detected using fault management, configuration management can be used to configure the network in order to bypass the troublesome part of the network.

Primary functions included in configuration management are:

- Define configuration information for network resources.
- Set / modify attribute values.
- Define / modify relationships between network entities.
- Initialize / terminate network operations.
- Examine values and relationships.

Performance management

One of the key functionality areas of network managing is that of performance management. By using indicators that measures the performance of a network, a manager can monitor and control the managed system by taking appropriate actions, using configuration management.

Indicators can be classified into two groups. Service-oriented indicators are used to confirm that the services levels that users expect are kept. Efficiency-oriented indicators are used to measure at what cost these services are provided. Primary indicators in the two categories are:

Availability	(service-oriented)
Response time	(service-oriented)
Accuracy (i.e. error frequency)	(service-oriented)
Throughput	(efficiency-oriented)
Utilization	(efficiency-oriented)

One possible problem of having agents residing in network devices gather information continuously is the amount of processing required at the node. On a shared network this can be avoided by using an external monitor whose sole (or major) function is to monitor the traffic on the network.

Another consideration on information gathering is whether exhaustive or statistical measurement is to be applied. If the traffic load is heavy, measuring every packet sent and received might prove an infeasible task for an agent. By obtaining statistical samples, it is possible to estimate the actual value of an indicator.

Security management

The functionality of security management can be divided into three separate groups:

- Maintaining security information.
- Access control.
- Encryption control.

The first part is concerned with monitoring and configuring data such as security keys and access right information at the agents. Another important task is that of keeping track of activity (and attempted activity) on the network.

The goal of access control is to prevent resources such as security codes, routing information, etc. from being accessed by unauthorized users. It is important to protect network resources from being damaged, either intentionally or unintentionally.

Within the concept of encryption control lies encryption of communication between manager and agents, as well as encryption at other entities on the network. Administration of encryption algorithms and distribution of encryption keys also falls into this category.

2.3 Network management protocols

Until the late 1970s there were no actual network management protocols available. The most useful tool for monitoring networks was ICMP (Internet Control Message Protocol), a protocol that can be used with all IP-systems. The primary function of ICMP, as far as network managers are concerned, is the *echo* / *echo-reply* messages, which can be used to test communication between network entities.

A more powerful means of network management was achieved with PING (Packet Internet Groper), a program that uses ICMP plus some options in the IP-header. PING supplies methods to:

- Test communication with nodes on a networks.
- Test communication with a network.
- Verify functionality of nodes on a network.
- Measure round-trip times.
- Measure loss of datagrams.

These basic management tools proved quite useful at the time, but they required operations from skilled network managers to provide the wanted functionality. As the tremendous growth of the Internet started in the late 1980s, it was apparent that the task of managing the interconnected networks with these tools simply was not feasible. The need for a capable and standardized protocol, not requiring expert network managers in the current extent, was growing. Three possible solutions was developed:

- *CMOT* (CMIP over TCP/IP) was the ISO standard for network management.
- *HEMS* (High-Level Entity Management System) was a generalization of an early management protocol - HMP (Host Monitoring Protocol).
- *SNMP* (Simple Network Management Protocol) was based on another early protocol - SGMP (Simple Gateway Monitoring Protocol).

Of these three, CMOT was chosen by the IAB (Internet Architecture Board) to be the long-term solution, as a transition to the OSI-based protocols was expected shortly. As a short-term solution, SNMP was chosen. It was not a more capable protocol than HEMS, but it was simple and the IAB saw no reason to put too much work into a solution that was supposed to be temporary.

However, CMOT never had the chance to replace SNMP. After SNMP was introduced to the network community, it was quickly established as the de facto standard.

3. SNMP

3.1 Architecture

Within the conceptual model of network management for TCP/IP networks, four basic components can be identified:

- Management station
- Management agent
- Management Information Base (MIB)
- Network management protocol

A management station (referred to as ‘manager’) is responsible for monitoring, configuring and controlling a number of nodes on a network, each equipped with a management agent (referred to as ‘agent’). Typical entities that may be equipped with agents are bridges, hosts, routers and hubs. Basically any node on a network can be supervised by a manager through an agent, either residing on the node or acting remotely as a proxy (see section 3.4).

A MIB is a collection of objects representing resources of the node where the agent resides. Every message sent from the manager to the agent results in reading or configuring an object in the MIB. The MIB does not necessarily contain the information of interest. It should rather be thought of as a logical representation of the information. How a request for monitoring/configuration of an object in the MIB is handled in order to actually affect the resource is not stated in the SNMP specification, it is an implementation issue.

The manager and the agent communicates through a network management protocol, enabling the manager to access objects in the agent’s MIBs. In TCP/IP networks the standard protocol is SNMP. SNMP is an application-level protocol, using UDP as its underlying transport protocol. Note that the SNMP protocol, as described in [7], is not restricted to UDP as its transport protocol, although most (if not all) implementation uses UDP.

When referring to SNMP one often means the combination of these four components described above, rather then just the communication protocol. While the protocol itself is rather simple and straight-forward, the SNMP architecture as a whole is quite complex.

3.2 SMI and MIBs

The MIB (Management Information Base) is the standardized data structure, not only for SNMP, but for TCP/IP network management in general. It is a hierarchical structure, with each leaf being an object of interest to a manager. The data structure should not be thought of as an object-oriented structure, since there is no inheritance in the tree. Rather, it is a logical grouping of related objects, where each node that is not a leaf describes the category of its underlying nodes. Since simplicity is an important issue only very basic data structures can be stored in a MIB. To be more precise: scalars and columnar objects consisting of scalars. Columnar objects, which allows more than one instance of the object itself, have to be grouped in conceptual tables (see Fig. 3.1).

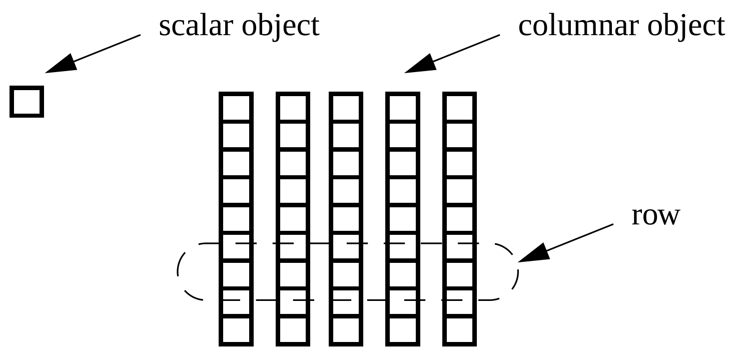


Fig. 3.1: To the left, a single instanced scalar object. To the right, a conceptual table consisting of columnar objects. One instance of each columnar object constitutes a row.

To uniquely identify a row in a conceptual table, one ore more columnar objects are carefully chosen to be indices of the table. More on identifying objects in subsection 3.2.1.

To make the concept of network management with MIBs functionable, two things need to be considered:

- The information structure for certain resources should be uniform within a system, i.e. objects describing the same information should have a standard representation on all network entities. This is accomplished by using standardized MIBs on all agents residing on nodes of similar types, holding the same type of information. Also, the manager needs to be aware of the structure of these MIBs.
- The data representation, i.e. the MIB structure, must be standardized.

The second point is addressed in [6] in which a structure of management information (SMI) is defined. The SMI describes a standardized detailed representation of MIBs, including syntax and techniques for object definition. The notation used to represent objects according to SMI is the Abstract Syntax Notation One (ASN.1).

Due to the complexity of the ASN.1 notation (and thus the formal SMI definition), I will not address the issue of representation of management information in this report.

That subject would require a separate report of its own. Note though that this in no way is an unimportant part of the area. On the contrary, it is indeed essential to anyone who wishes to understand MIB representation and construction in detail.

3.2.1 Object Identifiers

Each object within the SNMP management framework is identified with an *object identifier* (OID), which is a sequence of non-negative integers. One could think of the information supplied in a MIB as instances of objects (or variables) that can be viewed and configured with SNMP operations, identifying them by their OIDs. The OID itself describes the path to follow in the MIB to reach the leaf representing the object.

By *registering* the OID of an object one can make sure that that OID can never be registered for some other object. Also, the characteristics of the registered object can never be changed, and it can never be removed.

This might all seem a bit confusing, but an illustration of the tree structure generated by the OID scheme might clarify matters. The OID strategy was developed by ISO and CCITT (now ITU), which explains the structure of the top of the OID tree:

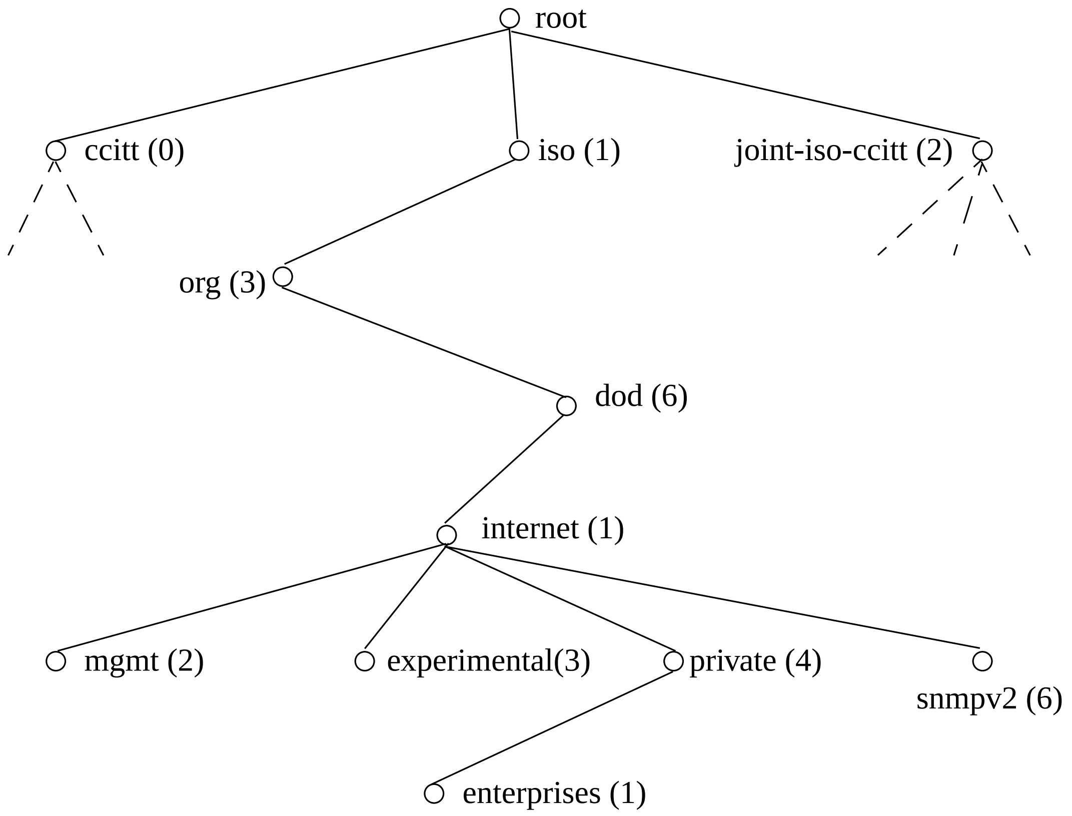


Fig. 3.2: A subset of the top levels of the OID tree.

There are only three values at the top level (0, 1 and 2), corresponding to the three categories ccitt, iso and joint-iso-ccitt. In the figure the path down to *mgmt* (with OID 1.3.6.1.2) is shown. Under *mgmt* lies the primary area for SNMP management information, i.e. the standardized MIBs. Under *private* (OID 1.3.6.1.4) one can register private MIBs by asking IANA (Internet Assigned Numbers Authority) for an *enterprise identifier*. This is suitable for vendors who need to implement specific MIBs for their products.

Any single instance of a scalar object has an OID consisting of the OID of the object appended by the suffix ‘0’. An instance of a columnar object has an OID consisting of the OID of that object appended by the concatenated values of the indices for the conceptual table to which the object belongs. To exemplify this, consider the following two cases:

- The scalar object class *sysLocation* has the OID 1.3.6.1.2.1.1.6. The single instance of *sysLocation* then has the OID 1.3.6.1.2.1.1.6.0.
- The columnar object class *ifSpeed* has the OID 1.3.6.1.2.1.2.2.1.5. The conceptual table in which *ifSpeed* resides has only one integer-valued index. An instance of *ifSpeed* then has the OID 1.3.6.1.2.1.1.5.n, where n is the single index (if two integer-valued indices had been used the OID would be in the form of 1.3.6.1.2.1.1.5.n.m).

3.3 The SNMP protocol

As mentioned in a previous section, the SNMP as a protocol is just as simple as you would suspect judging by its name. There are only three basic messages being sent between manager and agent:

- GET - retrieves the value of an object in an agent MIB.
- SET - configures the value of an object in an agent MIB.
- TRAP - enables the agent to alert a manager of an event.

The SET and GET messages are sent by the manager and are always followed by a responding message from the agent. The TRAP message is sent by the agent to the manager and is the agent’s only means of initiating contact with the manager. These messages are designed to support a management strategy called *trap-directed polling*. In a large network with many agents, having a monitoring manager polling the agents with GET messages results in an unnecessary amount of management traffic. Instead, by performing thorough pollings with long time intervals, such as once per day, and having the agents send TRAP messages as a result of unusual events, the traffic load can be greatly reduced. This strategy also reduces time-consuming processing both at the agents and at the manager. Notice that the SET / GET messages always spawn a response message from the agent, while a TRAP being sent to a manager will not result in a response.

The SNMP messages enables the manager to monitor as well as configure resources on the entities on which the agents reside. There are however a number of things a manager cannot do within the SNMP framework:

- It is not possible to change the structure of a MIB by adding or deleting objects in it. Only already existing objects can be modified.
- Only leaf objects are accessible to the manager, i.e. a table or a row of a table cannot be inspected in one atomic action.
- It is not possible to issue action commands to the agent.

The last of these restrictions can be solved implicitly by using an object instance in the MIB as a trigger. To invoke an action the manager can, for instance, set the value of a flag or set the value of an object representing a timer. This method is called *action invocation*.

3.3.1 SNMP messages

In Fig. 3.3 the general format of an SNMP message is shown. The size of SNMP messages is not defined and is limited only by the packet size of the underlying transport protocol (i.e. UDP).

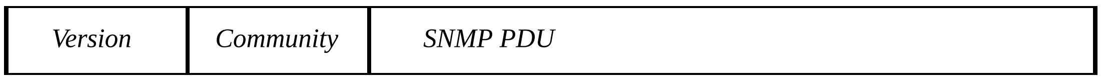


Fig. 3.3: General format of SNMP messages

- *Version* is the SNMP version used (SNMPv1, SNMPv2, SNMPv3).
- *Community* is the community name, a string acting as a simple kind of password to authenticate the message (see section 3.5).
- *SNMP PDU* is the protocol data unit containing the actual SNMP operation - one of GetRequest, GetNextRequest, SetRequest, GetResponse and Trap.

Any request PDU sent to an agent will result in a response PDU, while a trap PDU sent to a manager will be left without confirmation. The sequences of the different kinds of SNMP PDU exchange between manager and agent are shown in Fig. 3.4.

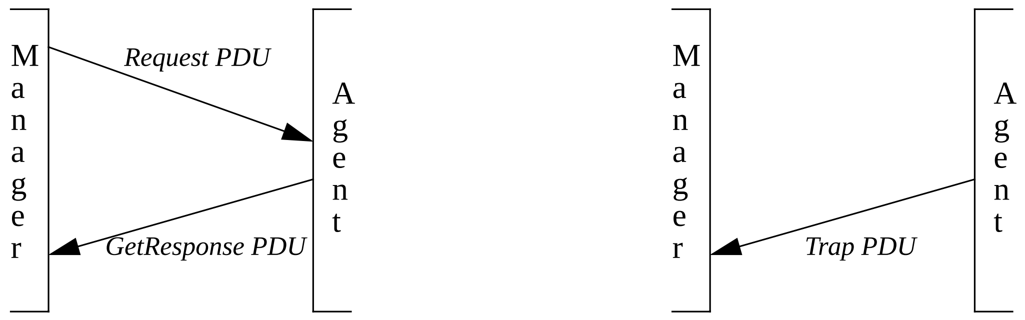


Fig. 3.4: SNMP PDU exchange sequences

The formats of all request PDUs sent from manager to agent are uniform:

PDU type	*request-id*	*error-status*	*error-index*	*variable-bindings*

Fig. 3.5: Format of SNMP GetRequest, GetNextRequest and SetRequest PDUs. The 'error-status' and 'error-index' fields are set to 0.

- *PDU type*: `GetRequest` / `GetNextRequest` / `SetRequest`
- *request-id*: message identifier
- *error-status*: set to 0
- *error-index*: set to 0
- *variable-bindings*: list of object instances whose values to retrieve / set

The request-id which is included in all SNMP request messages is used to distinguishing between different requests. The SNMP application can use the request-id to correlate incoming responses with outstanding request. It is also useful to detect duplicated messages.

The format of the GetResponse PDU is identical to those of the request PDUs, but allows for error information to be included:

PDU type	*request-id*	*error-status*	*error-index*	*variable-bindings*

Fig. 3.6: Format of SNMP GetResponse PDU

GetRequest PDU

A GetRequest PDU is sent by a manager in order to retrieve the value of one or more variables (object instances) from an agent.

When the agent receives the PDU it tries to find the object instances listed in *variable-*

bindings and retrieve their values, and then spawns a GetResponse PDU. The format of a GetResponse PDU is identical to that of a GetRequest PDU but with *PDU type* GetResponse PDU. If the agent is able to retrieve the requested value the *variable-bindings* of the returned GetResponse PDU includes a value for each of the object instances, and the *error-status* and *error-index* fields are set to 0.

If the operation is unsuccessful, the responding GetResponse PDU has the *error-index* set to the index in *variable-bindings* of the variable that caused the error, and error-status set to one of the following:

- `noSuchName`: the supplied object in the *variable-bindings* does not match any object identifier in the agent's MIB.
- `tooBig`: the size of the generated GetResponse PDU exceeds a local limitation.
- `genErr`: the agent is not able to retrieve the value for a requested object for some other reason.

Note that a GetRequest is an atomic operation, i.e. either all requested values are retrieved, or none. If at least one value of the requested object instances cannot be retrieved, the returned *variable-bindings* will be empty.

GetNextRequest PDU

The GetNextRequest PDU is identical to the GetRequest PDU with one exception: while the GetRequest *variable-bindings* lists the OIDs of the object instances to be retrieved, the GetNextRequest *variable-bindings* lists the OIDs of the objects *prior in lexicographical order* to the ones to be retrieved. What this means is most easily shown with an example:

> Suppose that the columnar object *ifMtu*, with OID 1.3.6.1.2.1.2.2.1.4, has instances with sequential OIDs in the range of *ifMtu*.1 ... *ifMtu*.4. A GetNextRequest PDU with the first requested variable in *variable-bindings* being *ifMtu*.2 will generate a GetResponse PDU with the OID *ifMtu*.3 as the first variable in the *variable-bindings* together with the retrieved value of *ifMtu.3*.
>
> If the third requested variable is *ifMtu*.4, the third variable and it's value in the responding *variable-bindings* will be the next object instance in lexicographical order. In this case it is 1.3.6.1.2.1.2.2.1.5.1 or *ifSpeed*.1, which is the instance of

ifSpeed at the first row of the conceptual table (see Fig. 3.7).

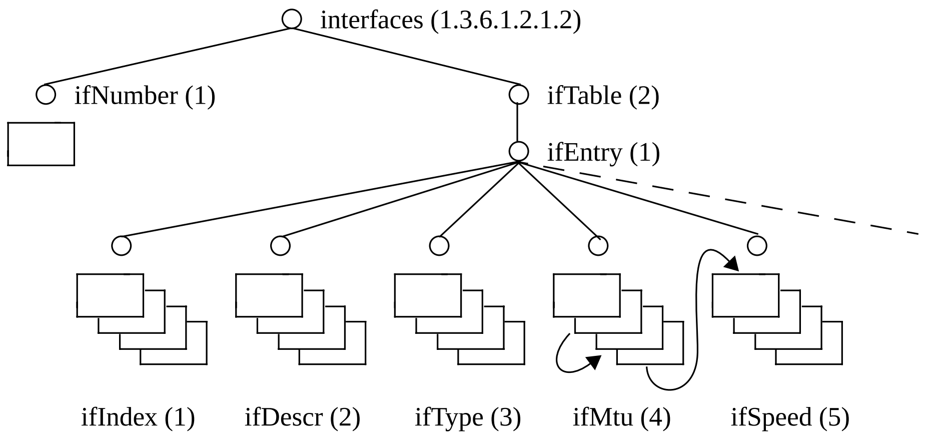


Fig. 3.7: A subset of the standardized MIB 'mib-2'. To the left, a scalar object 'ifNumbers' with a single instance. To the right, a conceptual table 'ifTable', consisting of a number of columnar objects, each with four instances.

If the GetNextRequest included the *ifNumber.0* scalar object instance, the corresponding returned variable would be *ifIndex.1*. Note that only scalar and columnar objects are accessible outside the MIB. Therefore the *ifTable* and *ifEntry* nodes will be skipped when finding the next object instance in lexicographical order after *ifNumber*.

This illustrates the usefulness of the GetNextRequest PDU. It enables the manager to dynamically explore the structure of the MIB view, without any prior knowledge, a technique called *MIB walking*. It also enables efficient retrieval of tables, even in the case where the number of rows are unknown. Note the GetNextRequest PDU, just as the GetRequest PDU, is an atomic operation.

SetRequest PDU

This request PDU is used by the manager to configure the values of object instances in the MIB view. The format is identical to that of the GetRequest and GetNextRequest PDUs, but the *variable-bindings* also includes a value for each supplied variable.

The SetRequest operation is atomic: either all of the object instances are updated, or none are. If the latter is the case, the returned GetResponse PDU has the *error-status* and *error-index* fields set. The types of error are the same as for the GetResponse PDU produced by a GetRequest, but with one addition:

- `badValue`: the object instance is supplied with an inconsistent value (e.g. bad type or length).

If the operation is successful, the returned *variable-bindings* will be identical to the one in the SetRequest PDU.

The SetRequest can also be used to create new rows in conceptual tables, though the SNMP definition [8] does not state that this functionality has to be implemented. If implemented, the agent would create the necessary instances (a new instance for every columnar object in the row) when it receives a variable name in the variable-bindings that does not match any OID. If not all object values in the new row are supplied in the SetRequest PDU, default values are used.

Trap PDU

Trap PDUs are sent from agents to one or more managers to report significant events. The format of the PDU differs from the request type PDUs:

PDU type	*enterprise*	*agent-addr*	*generic-trap*	*specific-trap*	*timestamp*	*variable-bindings*

Fig. 3.8: Format of SNMP Trap PDU

- *PDU type*: `Trap`
- *enterprise*: type of object (network entity) generating the trap
- *agent-addr*: address of the object generating the trap
- *generic-trap*: generic trap type
- *specific-trap*: specific trap code
- *time-stamp*: time elapsed between the last (re)initialization of the network entity and the generation of the trap

The *generic-trap* field specifies one of seven predefined trap types: coldStart, warmStart, linkDown, linkUp, authenticationFailure, egpNeighborLoss (signifies that an EGP neighbour for whom the sending protocol entity was an EGP peer has been marked down) and enterpriseSpecific.

If *generic-trap* = `enterpriseSpecific` an event has occurred that is specific to the enterprise. The *specific-trap* field contains the specific code for that event.

3.4 Proxy agents

To manage a network with the aid of SNMP, all the managed devices must support a common underlying protocol suite (typically IP / UDP). Now, some devices do not support the TCP/IP suite for some reason. Other devices might very well support TCP/IP but be to small to accommodate application entities such as an SNMP agent and a MIB.

To still be able to manage these type of devices through SNMP, a *proxy agent* can be used. A proxy agent is an SNMP agent that acts on behalf of one or more network devices. It has to support both SNMP (to communicate with the manager) and some device-specific means of communication. Fig. 3.9 illustrates the use of a proxy agent:

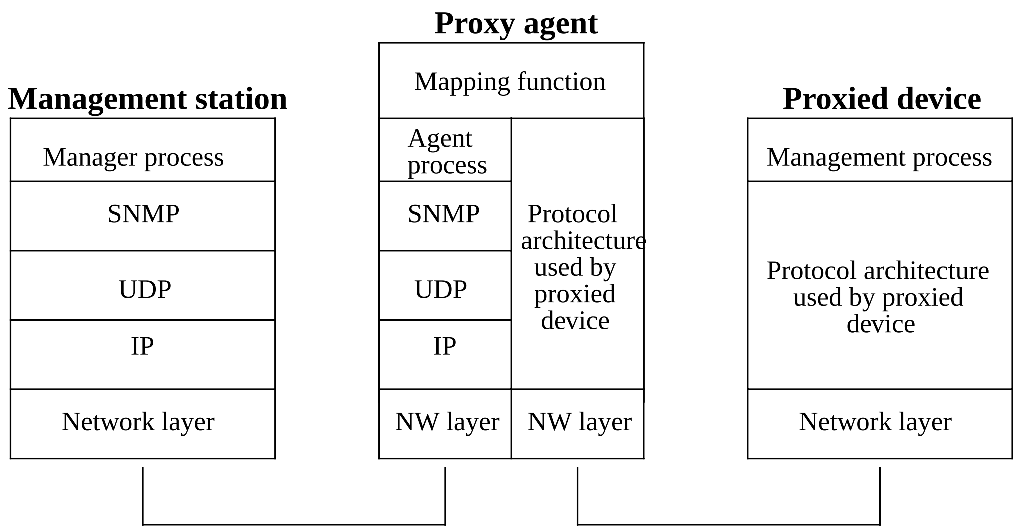


Fig. 3.9: Typical model of a proxy agent, showing the network management entities in the three interacting devices.

3.5 Security aspects

The definition of SNMP does not contain any major explicit security policies. There are some basic means of authentication and access control to the agent MIBs, but no secure network management environment is defined. That is an issue for higher-level applications. The basic concept that should be mentioned in this section is that of a *community.*

An SNMP community is created and controlled by the agent and describes a relationship between the agent and a manager in the form of:

- Authentication: Each community has a *community name*, a string supplied by the agent that is used in every SNMP message to identify the community.
- Access control: Each community has a corresponding *MIB view* which is a subset of a MIB, thus describing which part of the MIB that should be accessible to the manager. In addition a SNMP access mode is defined for each community. The access mode can be one of READ-ONLY and READ-WRITE and defines the overriding access restrictions to the objects within the MIB view.

Remember that an agent may be administered by more than one manager, which explains the need for different communities within the scope of one agent.

Of course, this strategy does not present a convincing security functionality. For instance, a message being sent from a manager to an agent containing a community string could easily be picked up by an eaves-dropper. The community name could then be re-used in any new SNMP message by the malicious source. Thus, the need for encryption at a higher level is apparent. However, it is not a concern for the SNMP standard.

4. SNMPv2

SNMP is at this time a rather old management framework, and although it is still sufficient for many systems it has a number of deficiencies, as listed in [1]:

- It is not suitable for very large networks, because of the performance limitation of polling.
- It has no functionality for retrieving large amounts of data, such as an entire table.
- Traps are unacknowledged.
- It has no real means of authentication.
- It does not support explicit action invocation.
- It does not support manager-to-manager communication.

With SNMPv2 some, but not all, of these weaknesses are addressed. For example, retrieval of large amounts of data can be done with a new PDU - the GetBulkRequest PDU. However, such an important issue as that of authentication is not a part in SNMPv2, as one would wish. Instead, that functionality together with other security-related issues is supposed to be included in the third version of SNMP.

Three primary areas of improvement can be identified when comparing SNMPv2 with the original SNMP framework (which to avoid confusion will be referred to as SNMPv1 from now on). It is the structure of management information (SMI), the protocol operations and the new concept of manager-manager communications.

4.1 SMI extensions

Of the additions of functionalities in the SMI for SNMPv2, the two most important to this report are: row creation / deletion and the concept of augmentation.

Row creation and deletion

Row creation by manager is not explicitly included in the SNMPv1 definition, although it is not prohibited. With SNMPv2, however, new functionalities are defined in order to facilitate manager control over conceptual tables in an agent MIB. All tables must explicitly state whether they support row creation / deletion by a manager or not. This is done by adding a new column, called *status column*, to the table. When a manager is to create a new row, the first thing to do would typically be to confirm that the row to create doesn't already exist by sending one of the possible GET PDUs (e.g. GetRequest). After that the typical way of creating the row is:

- Send a SetRequest PDU supplying values for those objects that don't have default values, setting the status column to `createAndGo`.
- If the operation is successful the agent changes the status column value to `active` and a GetResponse PDU is returned to the manager. If the agent fails to create the row for some reason an error code is set in the returned GetResponse PDU.

If the original SetRequest PDU doesn't supply values for all variables needed, the agent creates the new row but sets the status column to `notReady`, making the row inaccessible for value retrieval. If the manager at a later point supplies the missing values the agent will set the status column to `notInService`, and will not activate the row until the manager explicitly sets the column value to `active`.

In order to delete a row the manager sets the status column value of that row to `destroy`.

Augmentation

When designing an enterprise-specific MIB for a SNMP-enabled product, a vendor might want to use the tables contained in the standard MIBs but extend them with additional columns. In SNMPv1 there was no way this could be done without rewriting the table definition (which isn't allowed for a registered MIB) or defining an entire new table, containing the columns of the original table. None of these approaches seemed very practical and this problem has been addressed in SNMPv2.

SNMPv2 provides a method for extending the number of columns in a conceptual table without having to alter the definition of the table. This is done by using c*onceptual row augmentation*. By creating a new table that, instead of using indexing objects, uses *augmenting objects*, one can connect the rows in this table to rows in the original table. The result of this operation, as seen by the manager, is that the original table has been extended with additional columns. Note that every row in the original conceptual table must have exactly one corresponding row in the new augmenting table.

4.2 Protocol extensions

In SNMPv1 two major types of management communication are defined: manager-to-agent (consisting of requests) and agent-to-manager (traps). In SNMPv2 a new type is added - manager-to-manager communication. This type of communication is accomplished with the new InformRequest PDU, which is described below.

The PDUs used by SNMPv1 are kept in the definition of SNMPv2, but with some changes. In addition, three new PDUs are introduced: GetBulkRequest, InformRequest and Report.

GetRequest PDU

The GetRequest operation has undergone a small but significant change. In SNMPv1, the operation is atomic, i.e. if one of the values cannot be retrieved, then no values are returned. In SNMPv2 this operation is no longer atomic, every value that can be retrieved by the agent are included in the response to the manager. Those variables that cannot be identified by the agent are paired with an exception identifier in the returned *variable-bindings*. If a variable cannot be retrieved for some other reason the responding PDU has *error-status* set to `genErr`.

This new feature has big effect on the amount of management traffic on the network as well as the amount of processing at manager and agent nodes.

GetNextRequest PDU

Just like GetRequest the GetNextRequest PDU of SNMPv2 is no longer atomic, but retrieves the value of as many variables as possible. If there is no lexicographic successor to a specified variable the value in the *variable-bindings* in the GetResponse PDU will be `endOfMibView`.

GetBulkRequest PDU

The modifications of the GetRequest and GetNextRequest PDUs in SNMPv2 contributes to reduced amount of network management traffic and processing. These factors are even more reduced with the new GetBulkRequest PDU. GetBulkRequest is similar to GetNextRequest in that it specifies the object instances prior in lexicographical order to those to be retrieved. But where GetNextRequest retrieves only a single immediate successor, GetBulkRequest enables retrieval of multiple successors.

PDU type	*request-id*	*non-repeaters*	*max-repetitions*	*variable-bindings*

Fig. 3.10: Format of SNMPv2 GetBulkRequest PDU

- *PDU type*: `GetBulkRequest`
- *request-id*: message identifier
- *non-repeaters*: number of variables in *variables-bindings* for which only a single successor is to be returned

- *max-repetitions*: number of successors to be returned for the remaining variables in *variable-bindings*
- *variable-bindings*: list of object instances whose successor(s) value(s) to retrieve

Thus, for the first *non-repeaters* number of variables in *variable-bindings*, only a single variable name / value pair will be submitted in the responding *variable-bindings*. For the rest of the variables, *max-repetitions* number of pairs will be submitted. The number of returned pairs may be smaller than requested if:

- The size of the encapsulating message for the GetResponse PDU exceeds some size limitation.
- Some subsequent name / value pair has the value `endOfMibView.`
- The agent terminates the processing of the GetBulkRequest because of overload and returns a partial result.

SetRequest PDU

Just as in SNMPv1, the SNMPv2 SetRequest PDU is atomic - either all the variables are set, or none Actually, the only thing that differs between the two versions is response handling. The set of possible values for the *error-status* in the GetResponse PDU is in SNMPv2:

- `noAccess`: variable is not accessible
- `notWritable`: variable cannot be created or modified
- `wrongType`: value of wrong type supplied
- `wrongLength`: value of inconsistent length supplied
- `wrongEncoding`: value contains inconsistent ASN.1 encoding
- `wrongValue`: value cannot be assigned to variable
- `noCreation`: variable doesn’t exist and cannot be created
- `inconsistentName`: variable doesn’t exist and cannot be created under the present circumstances
- `inconsistentValue`: value cannot be assigned to variable under the present circumstances
- `resourceUnavailable`: value assignment requires allocation of a resource that is currently unavailable
- `genErr`: failure for other reason

SNMPv2-Trap-PDU

This PDU is very similar to the SNMPv1 Trap PDU. It is used in the same way, but has a different format. To simplify processing this PDU has the same format as the other SNMPv2 PDUs, with the exception of GetBulkRequest.

The first name / value pairs in the variable-bindings in SNMPv2-Trap-PDU contains:

- sysUpTime.0 (the time since the network management portion of the system was last reinitialized)
- snmpTrapOID.0 (trap identifier)
- variables names / values describing the event
- additional variables included by the agent

InformRequest-PDU

The InformRequest-PDUs are sent between managers, on behalf of the overlaying applications, in order to provide management information on a MIB view that is remote to the manager receiving the message. The format of the PDU is identical to the other SNMPv2 PDUs, with the exception of GetBulkRequest.

A successfully received InformRequest-PDU is passed on to the destination application, and a GetResponse PDU is returned with *request-id* and *variable-bindings* fields identical to those in the received InformRequest-PDU, and the *error-status* field set to `noError`.

If the incoming InformRequest-PDU size exceeds some limitation, the returned GetResponse PDU will have an empty *variable-bindings* field and *error-status* set to `tooBig`.

Report-PDU

In the SNMPv2 specification, a PDU called Report-PDU is included. However, no semantics or usage is yet defined for this PDU.

4.3 SNMP / SNMPv2 coexistence

When the SNMPv2 framework evolved, it was designed to be an extension av the original SNMPv1, thus making the transition to the newer version smoother. To avoid the difficulties of a absolute transition, strategies that allows coexistence between SNMPv1 and SNMPv2 entities are possible. The differences to be considered in such a strategy can be divided into two categories:

- Management information
- Protocol operations

Management information

The SMI for SNMPv2 is almost a proper superset of the SMI for SNPMv1, making MIBs defined in the SNMPv1 SMI rather easy to integrate in a SNMPv2 environment. To achieve interoperability, [10] lists a number of changes that need to be done in object definitions, trap definitions, compliance definitions and capabilities definitions. However, this will not be discussed in this report.

Protocol operations

The protocol operations in SNMPv1 and SNMPv2 are very similar. The only major difference is the addition of the GetBulkRequest and InformRequest PDUs in the later version. In [13] two important issues in a coexistence strategy are mentioned, which are explained in some detail in [1]:

- By using an SNMPv2 entity acting as a *proxy agent* between an SNMPv2 manager and an SNMPv1 agent, interoperability can be achieved. The PDUs sent from one of the entities to the other will be mapped to a suitable PDU compliant with the version of the receiving entity.
- One other possibility is to use a *bilingual manager* that is capable of communicating with SNMPv1 as well as SNMPv2, depending on which agent to communicate with.

5. SNMPv3

When SNMPv2 was presented it was incomplete in the sense that it didn't meet its original security related design goals. These functionalities, which will be addressed in the definition of the SNMPv3 framework [12], can be categorized as follows:

- Authentication: origin identification, message integrity and some aspects of replay protection.
- Privacy: confidentiality.
- Authorization and access control.
- Suitable remote configuration and administration capabilities for these features.

The third version of the SNMP framework will be derived from the previous versions, just like SNMPv2 was derived from SNMPv1.

The work of defining SNMPv3 is a procedure in progress. So far six RFCs have been presented, and the SNMPv3 Working Group is currently working on additional ones. The goal of the Working Group is to provide the documents (RFCs) necessary to provide a SNMPv3 standard, but they are not there yet.

6. Existing test tools

As SNMP has come to be the most wide-spread management framework on a continuously growing market, a large number of SNMP related software products, both commercial and public, have become available. Most of these products are SNMP management applications, designed to provide a means of administrating your network.

The kind of applications I am interested in for my work are, not surprisingly, applications used for SNMP testing. Of course, any management application where the user can send request PDUs to an agent of choice and then receive a response, can be used for manual testing. Applications specifically designed for some sort of automated SNMP testing are much more difficult to find.

I have found three commercial products designed for automated SNMP testing:

- “SNMP Test Suite” from InterWorking Labs
- “SNMP Tester” from Duet Technologies
- “SimpleTester” from Simplesoft

All of these products are capable of performing automated tests of both SNMPv1 and SNMPv2 agents. Since I have only been able to obtain a demo of one of these test tools (the “SNMP Test Suite”), I will in the following sections describe and compare the products according to their respective specification sheets. This survey should however prove useful for my future work, providing insight in what to expect from an SNMP test tool.

Note that the purpose of these tools is to test the implementation of the SNMP agents residing on the remote network entities, rather than testing the actual network entities *using* their SNMP agents. Thus, these commercial tools do not directly apply to this project, where the SNMP agent can be considered tested and reliable. It might however be possible to extend these tools to use them for the type of testing required within the scope of the project.

6.1 SNMP Test Suite

The SNMP Test Suite (revision 5.0) is a product developed by the California-based company InterWorking Labs. The tool, that can test standard MIBs as well as private ones, is used to:

- Verify correct lexicographical ordering.
- Test protocol compliance.
- Test error / exception handling.

- Test boundary conditions.

The tests are written as Tcl/Tk scripts, enabling the user to modify the tests to suit particular needs. There is a Developer's Guide available for interested customers.

The SNMP Test Suite is available for Solaris 2.4 and 2.5, HP/UX, Win95 and WinNT.

6.2 SNMP Tester

SNMP Tester from Duet Technologies (San Jose, California) can be used to test:

- Protocol compliance.
- MIBs (lexicographical ordering, access control, boundary control, etc.).
- Conformance testing (by checking agent responses and traps).

The tool can be used to test both standard and private MIBs, and can be used either in interactive or batch mode (for automatic testing). Also included in the tool is a MIB browser.

Just like the SNMP Test Suite, the SNMP Tester uses Tcl/Tk as a basis for tests, which allows for user defined test designs.

The SNMP Tester is available for Solaris 2.5.

6.3 SimpleTester

The last of the three tools is the SimpleTester, a product from Simplesoft, a California-based company. The tool tests agents for:

- Protocol compliance.
- MIB syntax errors.
- MIB compliance.

The data sheet of the product also states that it includes

> *"a command script generation and script execution capability that can be used for load testing, regression testing and user customizing"*

The SimpleTester is available for Win95 and WinNT.

7. System under test

In this chapter I will provide a brief description of the system that, primarily, should be the subject of test using an automated SNMP test tool - the AXD301 ATM switching system. In section 7.1 the functionalities and structure of the system will be discussed in brief. section 7.2 will address the subsystem developed by the department where this project is being done - the Switching Subsystem (SWS). The last section, 7.3, will address the current testing procedures and the extent of test automatization.

7.1 AXD301 switching system

The AXD301 is an ATM switching system, designed for different types of traffic (voice, data) on large networks. The switch scales from 10 GBit/s up to (and beyond) 160 GBit/s, making it useful in backbone networks, as well as in edge applications. The system supports charging based on both usage and duration, for both sides of the connection.

Management traffic between the AXD301 and remote operations management centres (OMCs) is carried inband by IP over ATM. There are three basic means of management:

- SNMP (for network management, both standardized MIBs and specific Ericsson MIBs are supported).
- FTP (for transfer of call details from a single system to a billing gateway system).
- HTTP / FTP (for element management, provided by a built-in web server - the AXD301 management system (AMS).

For installation and maintenance purposes, the AMS can be accessed at location, by connecting a work station to function as a local craft terminal (LCT):

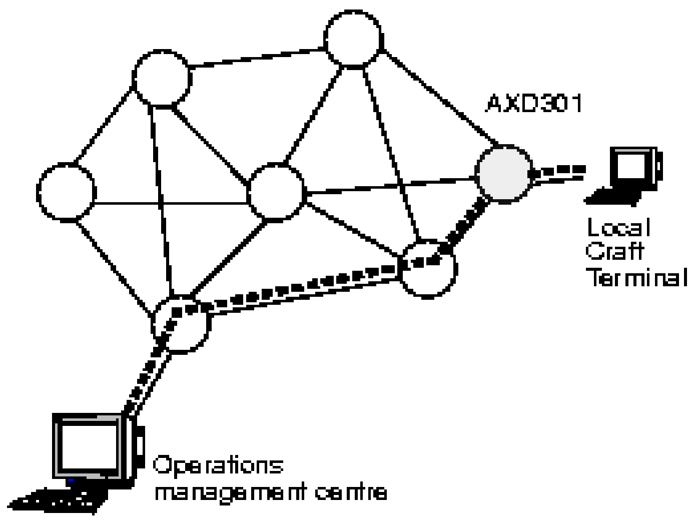


Fig. 7.1: Model of AXD301 management communication.

The two management interfaces for system monitoring / configuration - SNMP and AMS - allows access to the same MIBs:

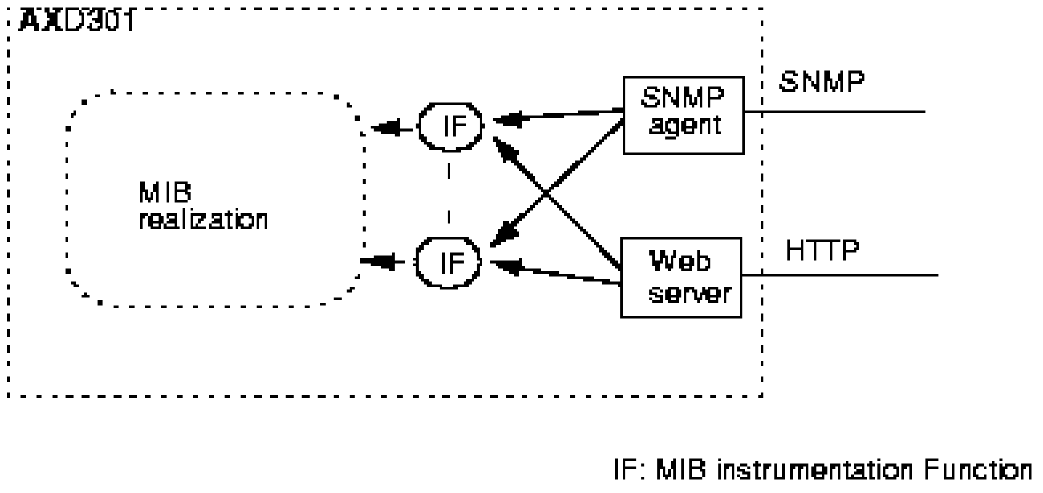


Fig. 7.2: Model of MIB access with two different protocols. The MIB instrumentation functions are used to map the objects in the MIBs to the actual system resources.

The AXD301 implementation is divided into five different subsystems, each developed and maintained separately:

- CPS - provides the basic execution environment with a database and support for load, start, restart, dual processor operation and failover/takeback.
- SWS - provides the basic ATM cell switching functions and the equipment management functions.
- ATS - provides call and connection control functions.

- OMS - gives support for a number of OMS (Operations, Administration, and Maintenance) functions such as alarm, logs, software management and charging support.
- AVS - provides interfaces between the AXD301 and the AXE system.

The department where this project is being done is responsible for the SWS implementation, which will be described in the following section.

7.2 SWS

The SWS (Switching Subsystem) provides the ATM switching fabric and the associated software (operation / maintenance / connection handling) of the AXD301. The subsystem also includes equipment management and device processor support blocks. However, the major blocks of interest in the SWS (interesting for automated testing within this project) are:

- Equipment Management (EQM) - provides the equipment management MIB for the switch fabric. Allows for monitoring and configuration of the boards installed in the AXD301.
- Fault Management (FTM) - handles alarm coordination for the switch fabric functions.
- Performance Management (PRM) - handles collection of performance data and allows for configuration of the performance measurements.
- Network Synchronization (NSY) - provides timing for traffic on the egress traffic links.
- Connection Handling (CNH) - manages connections through the switch fabric by allocating and releasing hardware resources.

Each of these blocks are subjects of *regression testing*, which will be discussed in the following section. Regression testing is a type of testing performed on a system after each iteration of development, in order to ensure that the system hasn’t regressed in terms of functionality since the last iteration.

7.3 Testing

Software testing is performed at every stage of the process of development, from testing small code modules to testing the entire AXD301 system. To illustrate this, consider the following test hierarchy:

- Module testing - a programmer tests his/hers Erlang or C module by using test tools and stubs for other modules.
- Block testing - a block, consisting of several modules, is tested with test tools, stubs and debugging tools.

- Subsystem testing in a simulated environment - a subsystem, consisting of several blocks, is tested by using simulated device processors, debugging tools and stubs for other subsystems.
- SSIT (Subsystem Integration and Test) - a subsystem is tested with real hardware. Other subsystems are often available, in the cases where they're not, stubs are used.
- SIT (System Integration and Test) - the entire system is updated by combining the different subsystem. Some regression testing is done.
- Function testing (FT) - exhaustive testing stage, where various functions are tested (Normally the main part of a function is implemented in one block).
- System testing - the system is tested for capacity, robustness, etc.

The major testing stage in this hierarchy is the function testing. In this stage, every block in the SWS are tested using test specifications. Examples of tests for the basic SWS software blocks may be:

- EQM - test that hardware devices (e.g. boards and links) can be blocked / deblocked, that parameters can be read and written, etc.
- FTM - generate abnormal traffic over the system and verify that the correct alarms are generated.
- PRM - generate traffic over the system and verify that various (performance measuring) counters are set to the correct values.
- NSY - the synchronization uses two clocks, having a master / slave relationship (the slave setting its time to the master's). Force a restart and verify that the clock that was master before restart still is.
- CNH - make certain VC connections and verify that they are properly reserved.

Current extent of automated testing

Today, some of the test cases in the SWS subsystem have already been automated (perhaps as much as 25%). These tests do not use the standard interfaces AMS and SNMP. Instead, the system is accessed 'from the inside' by accessing the MIB functions directly and using an interface to the device processors to configure the hardware (something that would normally require human interaction). A model of the setup for these SWS autotests is shown in Fig. 7.4.

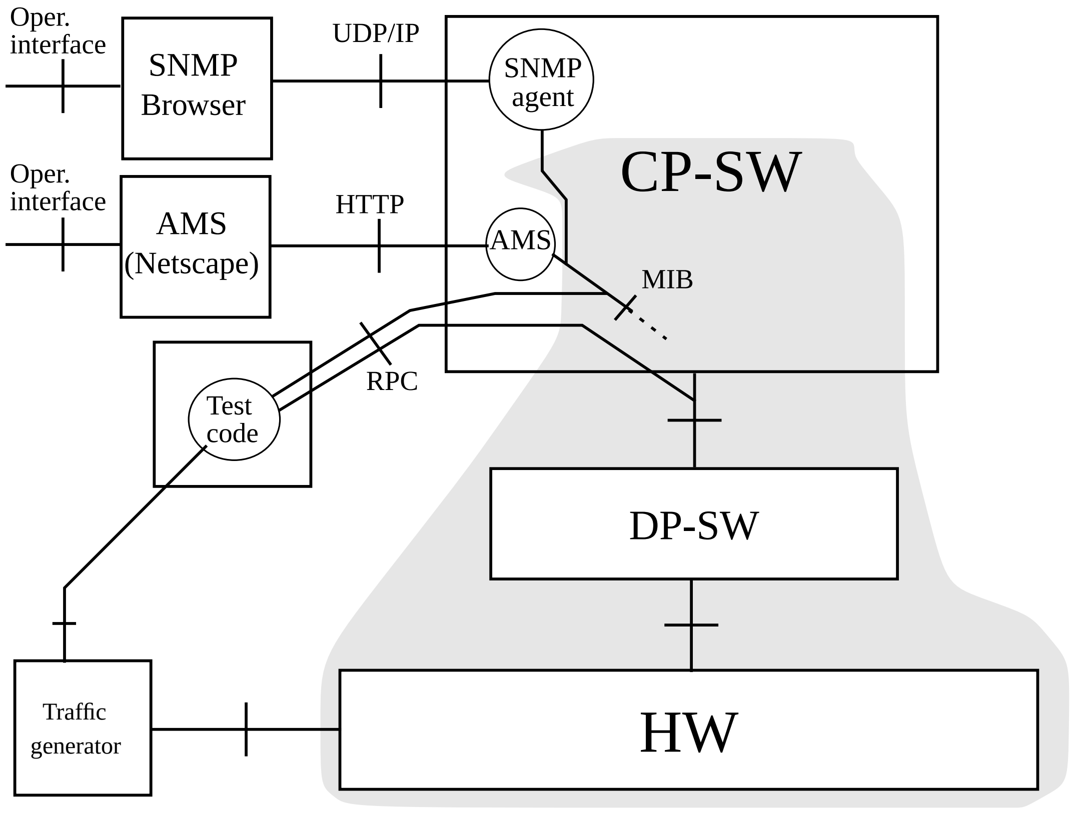


Fig. 7.4: Current autotesting procedure in the SWS subsystem. The shaded area marks the system under test (the AXD301). In the model the system is divided into three parts: the control processor software (CP-SW), the device processor software (DP-SW) and the hardware (HW). In this model the test tool also controls an external device, a traffic generator, to introduce stimuli to the system.

The reason that the present autotesting is performed in this manner is the lack of good test tools and also that the focus has been on testing the SWS subsystem only. If one would want to perform tests on the whole system using one of the standard management interfaces, AMS might not prove a suitable interface for autotesting, because of the graphical user interface. However, one should be able to use ordinary SNMP messages in an automated fashion, in order to perform tests on the system.

The goal of this project is to investigate the possibilities of using the SNMP interface to access the system with an automated test tool, and hopefully increasing the amount of automation in the testing process. Of course, there will always be tests that cannot be automated. For example, some tests require boards to be physically removed from the AXD301 rack, something that must be done by a human.

8. Automated testing

This chapter will address the basics of automated testing procedures; common testing concepts, test selection for automation process and desirable properties of an automated testing platform.

8.1 Testing terminology

Before moving on to test automation issues there are some testing procedure terms, frequently used in the rest of this report, that need to be introduced. They have rather obvious meanings, but I will address and describe them briefly anyway.

Test cases

A test case (often referred to simply as a ‘test’) is the key testing entity - a single test used to confirm a certain expected behaviour of a system. The general steps to process in an individual test case are the following:

1. Set up and confirm preconditions.
2. Perform some actions.
3. Confirm postconditions.

Steps 1 - 2 defines the actual test by first defining a test case specific state to reach by stating a set of preconditions to be fulfilled and then performing some actions. In the last step, the expected outcome is compared to that resulting from step 2.

The last two steps can be executed repeatedly. For example, one might want to define a test case where functions are called sequentially and the return values compared to the expected ones.

Test suites

Test cases are grouped together in test suites, based on the function area they are used to test. For example, Equipment Management (EQM) test cases are defined in a separate test suite, Fault Management (FTM) test cases in another, and so on.

Test execution

With test execution I mean performing a defined set of test cases from one or more test suites, in a defined order and under certain conditions.

8.2 Things to consider when automating tests

When performing a transition of testing from manually performed tests to tests running on an automated platform, there are a few things to consider. Both the issue of which test cases to automate and the issue of how to design an automated testing environment are worth considering. In this section I will address these issues, and try to pin-point the different problems and desirable properties when automating tests.

This section is based on documents written by people with experience of testing and test automation.

Which tests to automate

A common mistake is to try to automate the entire testing process, just for the sake of automating. Even if 100% of the tests could be automated (indeed an unlikely scenario), the work of automating these tests would probably be much more time consuming than to just settle for a reasonable part of the tests [15][21].

The process of automating, running and documenting a test takes longer than performing the test manually, the estimations are often set to 10 times longer [22]. Therefore, one should settle for automating only those test cases that are to be run 10 times or more. A preferable approach is to start with those groups of tests whose automation result in an obvious time gain, and settle for that, at least for the time being [14]. One should resist the temptation to automate tests just because they are suitable for automation. Instead, one should consider the total time gain of automating a particular group of tests.

Automated test tool design

Listed below are a number of issues to consider when designing an automated testing framework, some of them included in [19].

1. First of all, one should recognize that a test tool is a software system itself. The more advanced you try to make it, the less reliable it turns out to be as the number of bugs tend to increase. A safe strategy is to start off with a small test tool, providing only the functions necessary to automate the first set of tests.

2. Try to make a reasonably general test tool. It should not have to be recoded when extending the existing set of tests or for any small modification of the system under test. Also, it is generally not a good idea to automate testing of a system that is not stable, i.e. a system that is still under development.

3. Tests included in a test suite should be able to run individually and in arbi-

trary order, independent of each other. Therefore, it is important to consider how to reach the preconditional state of a test case before running it. For each test case there should be a clearly defined start state for the test platform, for example described by a set of global variables defined outside the definition of the test cases.

4. Design a test case specification language abstract enough not to discourage users without extensive programming background, but at the same time powerful and general enough to serve its purpose. One might consider having several layers of abstraction in the testing platform: an easy-to-read/-write instruction language (possibly enterprise specific) that is automatically translated into a more complex language that the test execution part of the test tool is able to interpret. Still, the user of a test tool probably needs some sort of programming experience to compose new test cases.

5. When a test case fails or some other error occurs, the test tool should be able to return to a “safe state” from where the test execution can continue. The failure of a test case should not result in abortion of the entire test execution, something that can be achieved with a solid exception / error handling strategy.

6. When constructing an automated test tool it *will* have bugs, but there is one bug in particular one wants to avoid - the so called *false positive* [16]. This means that a test case is reported successful when it really has failed. Bugs that causes the test execution to abort or that gives a false negative is much less dangerous, since these kind of bugs can be observed and isolated. A common cause of the false positive is a faulty exception handler that catches exceptions that shouldn’t have been caught and then doesn’t process these exceptions correctly.

7. Allow for pausing, single-stepping and resuming the test execution. Stepping through a test case at slow speed can be very useful when debugging.

8. Produce the test result logs in a format readable by humans, but at the same time possible to read by machine if there is, or might be, a need for that.

9. Model of an SNMP test tool

In this chapter I will propose a methodology for automated testing, using SNMP as a standard interface to the system under test. The methodology should be defined detailed enough to base an implementation of an SNMP test tool on. I will start by addressing the requirements of such a methodology and a corresponding implementation. In the rest of the chapter, a model of a test tool will be outlined, and a test case specification language will be defined.

9.1 Task and requirements

As mentioned in the beginning of this report, the goals of the project was to:

- Describe a methodology for automated testing, using SNMP as an interface. This includes defining the components needed in such a testing platform (and describe how they interact), the control flow during test execution and a test case specification language in which to describe test cases to automate.

- Implement a prototype based on this methodology. The prototype should be able to operate on an actual piece of equipment having a residing SNMP agent (e.g. an AXD301 switch).

To be more specific, the functionality requirements that a testing platform based on the methodology should meet are:

At test case level

- Retrieve MIB variable values from remote agent and compare these values to expected ones.
- Set MIB variable values at remote agent.
- Receive SNMP traps and store information on them, in order to compare them to expected traps.
- Control external devices and control the system under test using other interfaces than SNMP.

At test execution level

- Allow execution of multiple successive test cases (in arbitrary order and from multiple test suites). An error occurring while executing a single test case should not affect the execution of the rest of the test cases. This implies a robust exception handling strategy.
- Logging of test execution results.
- Execute same test cases with different configuration.

At global level

- Capability of interpreting test cases stored in a suitable test case specification language.
- Allow testing of SNMP agents at arbitrary IP address, UDP port and trap UDP port.

Also, an implementation of an SNMP test tool should provide a graphical user interface.

9.2 Conceptual model

When designing the SNMP test tool model, my objective was to present a design that was general in the sense that it should be totally enterprise independent and suitable to test any SNMP controlled equipment, and extensible in the sense that it would have a clear and natural structure of separate modules. Illustrated at a high level, a conceptual model of the design is shown in Fig. 9.1:

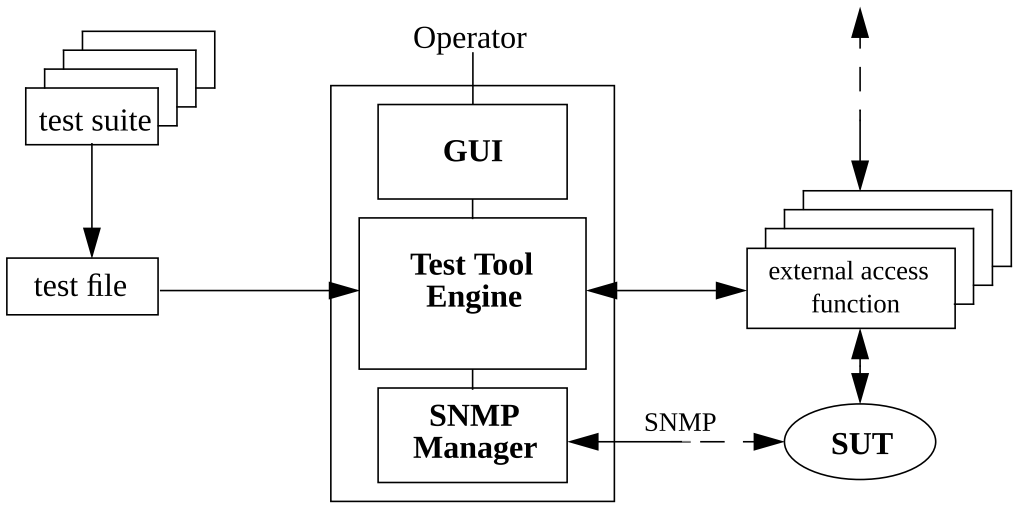


Fig. 9.1: SNMP test tool model.

This test tool model consists of three separate parts, where each underlying part is independent of the overlaying ones. These are:

- SNMP Manager - A network management entity providing an SNMP interface to overlaying components.
- Test Tool Engine (TTE)- The major part of the test tool, providing functionality for test case interpretation, communicating with the system under test (SUT) via the SNMP manager and inducing stimuli using external access functions.

- GUI - A graphical interface to the test tool. The tool should also be able to run from command line.

For each type of logical device that might be used to introduce stimuli to the SUT or for any other action using external interfaces or devices, one or more external access functions (EAF) should be provided. References to these EAFs should be included in the test cases where they are needed.

9.3 Components

SNMP Manager

The task of the SNMP manager is to offer full SNMP functionality for the commonly used versions SNMPv1 and SNMPv2c. It should provide the TTE with the ability to set up SNMP sessions with agents residing on arbitrary IP addresses / UDP ports and to handle the usual SNMP operations (GET, SET and TRAP). These operations should, in order to keep the test tool and the test case specifications simple, be synchronous. This means that when an SNMP message has been sent by the manager, it halts until a response is received or a timeout occurs. The SNMP Manager should also supply the test tool with more complex functionality based on these basic SNMP operations. For example, it is desirable to have means of confirming values of MIB variables, i.e. compare the values to expected ones.

A simple but satisfying trap handling strategy is to time-stamp all incoming traps and collect them in a buffer, which the TTE can read from and clear. Using this method, incoming traps can be examined by the TTE at any time, not necessarily at the time of arrival.

It is worth mentioning that by keeping the SNMP Manager in a separate module, one will have an SNMP managing entity that can be used with any SNMP application, not just an SNMP test tool.

Test Tool Engine (TTE)

The TTE is the part of the model that handles the actual test execution. It receives input in the form of a test file, where a test case execution order is defined. It parses and interprets the test cases used and performs the operations required. The outcome is interpreted and logged. The TTE uses the underlying SNMP manager for all communication that is not directed to external devices or to the SUT using a different interface. These means of communication are controlled through EAFs, which will be described in some detail in section 9.5.

Graphical User Interface (GUI)

It is possible, and probably useful, to extend the test tool with a GUI rather than running the tool from command line. How this GUI should be designed will not be addressed in this chapter. However, a minimum of features to expect from such a GUI is:

- Ability to display (and possibly edit) test suites.
- Quick and easy ways of changing preferences (IP address, UDP port, etc.).
- Ability to present test execution results to the user.

9.4 Example setup

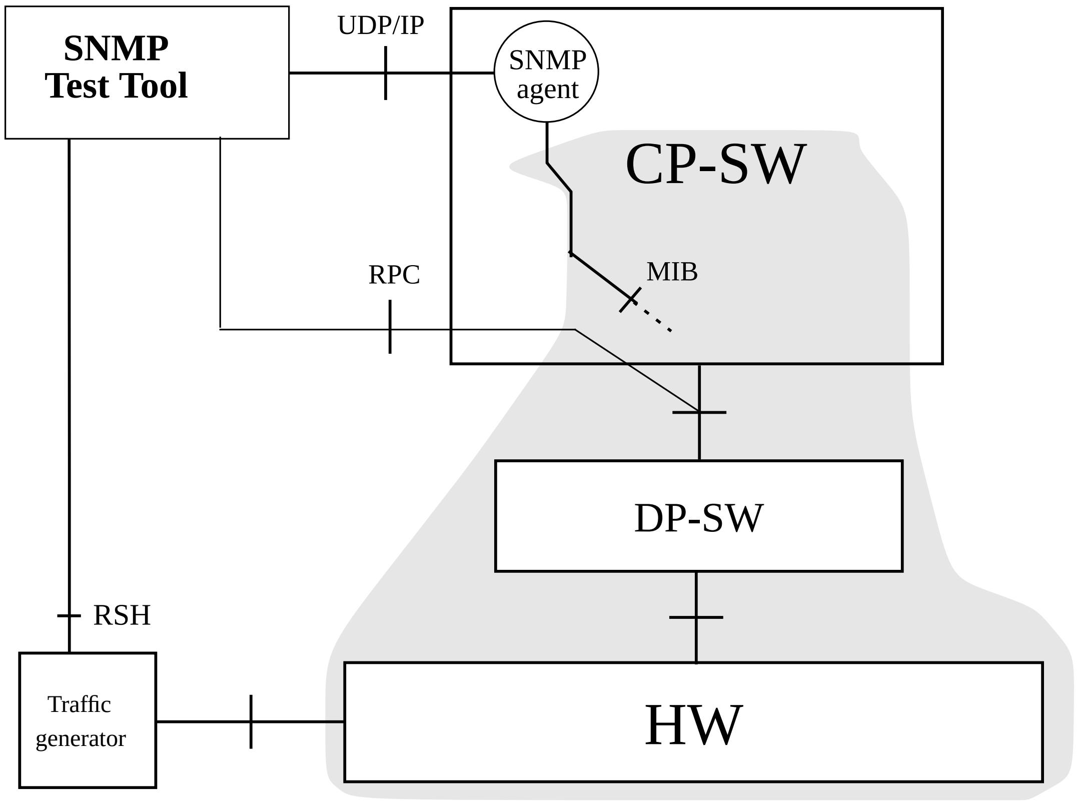


Fig. 9.2: Automated testing of the AXD301.

Fig. 9.2 illustrates how a test tool based on the model could be used to test the AXD301 switch. A major difference, when compared to the testing strategy used in Fig. 7.2, is that the main interface to the system is a well-defined management interface (SNMP over UDP/IP). Not all operations can be performed using this interface however. One might still want to use direct access to the device processors in order to, for instance, check the status of certain LEDs on the boards in the switch.

In the illustration above, two different kinds of stimuli are provided:

- Traffic generation (e.g. a HP75000) by remote shell invocation.
- DP access (through the DPCI+) by remote procedure call.

The way these entities are controlled by the test tool should be handled by (at least two) EAFs.

9.5 Test execution

In this section I will try to illustrate the process of a test execution, what the input to a test tool should be, and what output is to be expected. In short, this is what happens:

- A test file is executed.
    - One or more test suites are included, for each test suite:
        - Parse the test suite.
        - Load each test case.
    - Each test case in the test file is executed in order.
        - The result of the test case execution is logged.

Test files

The expected input to the TTE is a test file that describes which test cases in which test suites should be run and in what order they should be run. For example, study this simple test file:

```
include “myTestsuite.ts”

myTest1
myTest2
for (i := 0 , i < 10 , i := i + 1) {
        myTest3
        myTest4
}
```

The test file starts by including a test suite, containing definitions of at least the test cases named *myTest1*, *myTest2*, *myTest3* and *myTest4*. Next, test cases *myTest1* and *myTest2* are run sequentially. The *myTest3* and *myTest4* test cases are run in a loop, increasing the variable `i`. This illustrates how the test cases can use global variables (in this case `i`) that are set and altered in the test file.

The result of running a test file will be saved in a log file, having a default name if not explicitly stated otherwise. To the log file name, the date and time of test execution will be concatinated.

The syntax of test files will not be discussed in this report, and the above code segment should be considered merely an example of what the contents of a test file might look like. I leave it to the implementors to construct test files in a suitable format, depending on choice of implementation languages and tools. However, a test file must allow for invocation of test cases defined in test suites, and declaration and initialization of global variables to use in these test cases.

Test cases

The test cases definitions are stored in test suite files, either separate or grouped together in the same file. These test suites must be included in the test files that makes use of any test case defined in them. A test case definition has the format:

```
testcase <name> <body>
```

For example:

```
testcase exampleTest1
    external trafficGen generateSomeTraffic {1 0 500}
    snmp_set someAdmState.0 blocked
    confirm someOpState.0 disabled
end
```

The above test case starts with a stimuli inducing statement, in this case an external device is used to generate traffic over the SUT. After that, a certain MIB variable is set and the value of another is compared to an expected value. The execution of the test case is illustrated in Fig. 9.3:

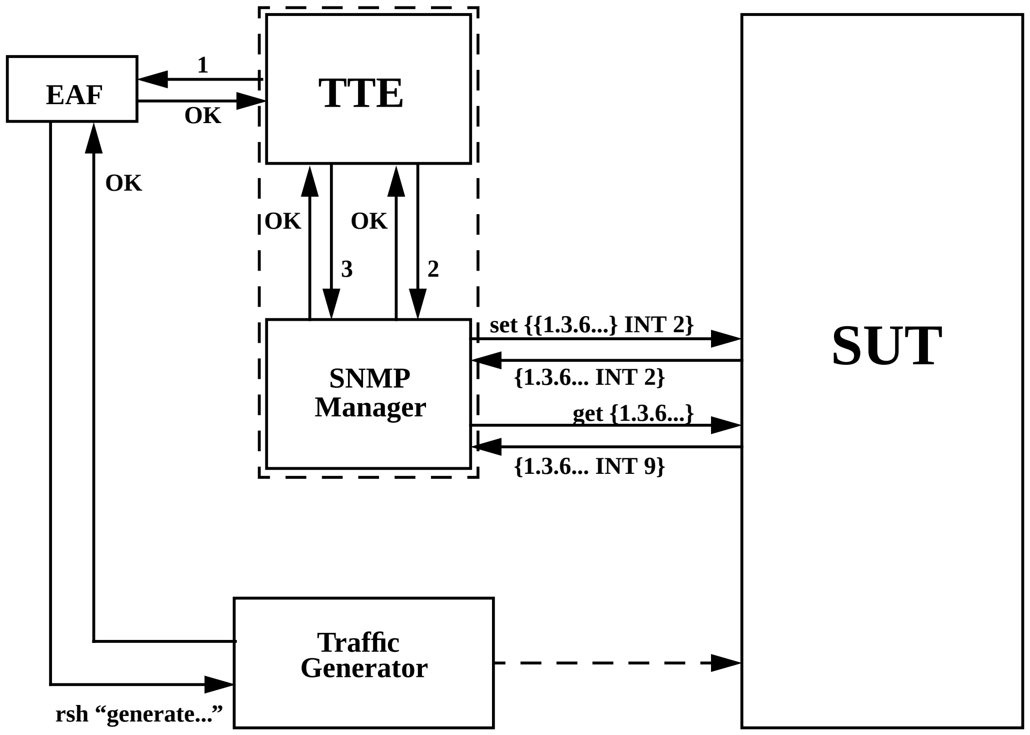


Fig. 9.3: Successful execution of the exampleTest1 test case:

1. *`external trafficGen generateSomeTraffic {1 0 500}`*
2. *`snmp_set someAdmState.0 blocked`*
3. *`confirm someOpState.0 disabled`*

The only other statements allowed in a test suite are the

`include_mib <mibfile>` and
`include_eaf <EAF-file>`

statements, which are used to declare which MIBs and EAFs that are used by the test cases defined in the file.

The test case specification language syntax will be described in the next section, and a formal BNF definition can be found in Appendix A.

External Access Functions (EAFs)

External Access Functions are mapping functions to the stimuli inducing devices used in the various test cases. These functions should be collected in one file per logical device. For example, in test case *exampleTest1* above, there must be a file named "trafficGen.eaf" containing a mapping function `generateSomeTraffic`. The file "trafficGen.eaf" could have the following appearance:

```
EAF generateSomeTraffic {arg1, arg2, arg3} {<BODY>}
EAF generateBitErrors {arg1, arg2} {<BODY>}
...
```

The purpose of EAFs is to keep enterprise-specific code / components separate from the general components that constitute the test tool. How these EAFs are implemented, if there is a need for that at all, is totally dependent on the implementation and will not be defined in this report. The important thing for a particular implementation is to have a well-defined way of invoking these functions from within test cases, and to receive possible return values.

9.6 Test case specification language

A typical test case would begin by assuming some preconditions and verifying those that can be verified. Then, if these preconditions are met, some kind of stimuli is to be introduced to the system. After the stimuli has been applied, certain conditions should be fulfilled.

A test case specification language suitable for this kind of test tool should require at least these commands:

```
snmp_get OID
snmp_getnext OID
snmp_set OID Value

confirm OID Value Timeout
confirm OID Min_value Max_value Timeout
confirm_ne OID Value Timeout

confirm_trap Traptype Timeout
clear_trap_buffer

external Device EAF Args
```

snmp_get **snmp_getnext** **snmp_set**	These commands are used to manipulate the SUT using the SNMP interface. `snmp_get` should return the value of the object instance defined by the provided OID (object identifier) if successful, `snmp_getnext` should return the next OID in lexicographical order. If the execution of any of these commands is unsuccessful an exception is raised which, if not caught, will cause the test case to abort.
confirm **confirm_ne**	These set of commands are used to compare expected MIB variable values to the actual ones. `confirm`/3 expects the value *Value* of the object instance described by *OID*, `confirm`/4 expects the value to be in the interval defined by (*Min_value*, *Max_value*) and `confirm_ne` expects anything ***but*** *Value*. The last argument to these commands, *Timeout*, defines the maximum time in msecs to wait for the value of the OID to be set to the expected one. The test tool could, for instance, poll the variable at regular time intervals, until the timeout is exceeded or the variable is set to the expected value. These commands will raise an exception if not successful.
confirm_trap **clear_trap_buffer**	All incoming traps are collected in a trap buffer, which is cleared at the start of each test case. The buffer can also be explicitly cleared in a test case using the command `clear_trap_buffer`. `confirm_trap` checks whether a trap of type *Traptype* has been received or is received within *Timeout* msecs. If not, an exception is raised.
external	The `external` command is used to access logical devices other than the SUT, or to access the SUT through other interfaces than SNMP. When invoking `external`, the file containing the corresponding EAFs should already have been included in the file containing the test case definitions. The test tool will try to map the function *EAF* of the logical device *Device* to a previously included EAF, and then invoke that EAF with arguments *Args*. If successful, `external` will return the result of executing that EAF with the provided arguments. If unsuccessful, an exception will be raised.

Table 9.1: Basic commands of the test case specification language

Example:

The AXD301 can have a number of different boards plugged in, in different slots in different subracks. By blocking these boards, they should be disabled. A very simple test case to verify this could look something like this:

```
testcase block_board_test
    snmp_set piuAdmState.$subrack.$slot blocked
    confirm piuOpState.$subrack.$slot disabled 8000
end
```

In this test case, a board identified by its position ($slot in $subrack) is blocked using an SNMP operation. Next, we want to confirm that the board is really disabled. If we cannot confirm this within 8 seconds, or if the SNMP GET operation should have already failed, the test case should fail.

In addition to these basic commands, the test case specification language should be extended with the following functionalities:

- Local variable assignment and comparison - a common procedure is to compare the value of an object instance at different times, to see whether it has changed. To do this we need to be able to store values.
- Conditional statements - in a test case, one could wish to continue the test in different manners depending on, for example, the value of a variable.
- Exception handling - the commands described above will, in case of failure, raise an exception which will cause the test case execution to halt. If these exception can be caught, the commands can be used in conditional statements by returning true (if successful) or false. Example of usage: `if (try_expr (confirm myOID “Hello” 500))` ...
- The ability to abort test cases, explicitly stating whether they were successful or not, by using special `success` and `fail` commands. The use of any of these command would abort the test case execution and continue with the test suite. If `fail` is invoked, the test case would abort with an exception, as with any ordinary failure.
- Creating and deleting entire rows in conceptual tables.
- In some test cases, it is of interest to compare the contents of an entire table before and after some sequence of actions. To simplify this procedure, the language could be extended with a `snmp_gettable (OID)`

command, together with commands for manipulating retrieved tables.

- Explicit logging, enabling log print-outs of informative strings and variable dumps from within the test cases.

To allow for these extensions in functionality, the following commands should be added to the test case specification language:

```
set VarName Value
if Boolean Statements end
try_expr Statement
succeed
fail

snmp_createrow TableOID Indices Values
snmp_deleterow TableOID Indices

snmp_gettable TableOID
snmp_table_diff Table1 Table2
snmp_table_find Table Row
snmp_table_size Table

log String
```

set	Assigns *Value* to variable name *VarName*. Data type issues will not be addressed here, one can assume that the only allowable data type is a string of arbitrary length.
if	Evaluates *Statements* only if *Boolean* evaluates to `true`.
try_expr	If *Statement* raises an exception, it is caught and `true` is returned, otherwise a value different from `true` is returned.
succeed **fail**	Both commands aborts the test case. In the case of **succeed** a normal abortion is performed. In the case of **fail** an exception is raised.

snmp_createrow	Creates a new row in table identified by *TableOID*. The new row will have indices as stated in the list *Indices*, and will have the entries stated in the list *Values*. In the list of *Values*, a '?' marks an undefined value, which implies that a default value should be supplied for that columnar object instance.
snmp_deleterow	Deletes the row in table *TableOID* with indices as stated in the list *Indices*.
snmp_gettable	This command tries to retrieve the contents of the table specified by *TableOID*. If the object identifier isn't that of a table, or if the table cannot be retrieved for some other reason, an exception is raised. The representation of the returned table contents should be a collection (e.g. list) of rows.
snmp_table_diff	Given two tables, *Table1* and *Table2*, in the format returned by `snmp_gettable`, this commands returns a new table consisting only of those rows that exists in *Table1* but not in *Table2*.
snmp_table_find	This command returns all rows in *Table* that matches *Row*. If values for all elements in *Row* are specified, a table consisting of one or zero rows should be returned. Otherwise, a table of all matching rows is returned.
snmp_table_size	Returns the number of rows in *Table*.
log	This command is the test case writer's only means of controlling the output to the test result log. It can be used for simple string output only, but should be able to handle more complex cases such as `log (snmp_get myTable.1.3.2).` This example statement should print the return value from **snmp_get** to the log.

Table 9.2: Additional commands to the test case specification language

Example:

```
testcase find_matching_rows
    set table (snmp_gettable piuTable)
    set matching_rows (snmp_table_find
                          $table
                          {? ? ? 1 ? enabled ? ? ? ?})
    set number_of_matching_rows
```

```
                (snmp_table_size $matching_rows)
        if ($number_of_matching_rows = 0) fail end
    end
```

9.7 Test result log

The log file created while running a test file is the only feedback given to the user of the test tool, at least if no GUI is being used. Into the log the order and result of the test cases are written, along with any exceptions and user-forced entries. A test case is considered successful if no uncaught exceptions are raised during execution. If an exception is raised, the test case is aborted and considered a failure, after which execution of the next test case commences.

This is an example of a log file created by running the test file defined in file “myTestfile.tf”, executing the test cases *myTest1*, *myTest2*, *myTest3* and *myTest4*, in that order:

```
##### TEST EXECUTION LOG FOR FILE “myTestfile.tf”
##### Date: 1998-12-24

### Starting test case myTest1 at 13:35:01
### Finishing test case myTest1 at 13:36:16

### Starting test case myTest2 at 13:36:16
could not confirm “1.3.6.1.4.1.193.14.1.2.4.6.1.1.6.1.4 == enabled, value
set to disabled
        in “confirm piuOpstate.$subrack.$slot enabled 5000”
### Test case myTest2 failed at 13:36:23

### Starting test case myTest3 at 13:36:23
no matching function for external
        in “external DP change_mep {1, 0, 1}”
### Test case myTest3 failed at 13:36:24

### Starting test case myTest4 at 13:36:24
This is a line produced with the ‘log’ command. /Martin
### Finishing test case myTest4 at 13:36:48

##### 4 test cases run
##### 2 successful
##### 2 failed
```

Fig. 9.3: Example of a test execution log file

In this example the *myTest1* and *myTest4* were considered successful, while *myTest2* and *myTest3* failed for some reason (two different types of exceptions was raised). In test case *myTest4* the log command has been used to print a string to the log.

9.8 Test execution flowchart

Below is a simplified flow chart, describing the evaluation process when executing a test file. Not included in the chart are among other things:

- Log handling, except for the explicit “log” command.
- Errors, exceptions and abnormal abortions.
- Nested commands, such as log (snmp_get ...).

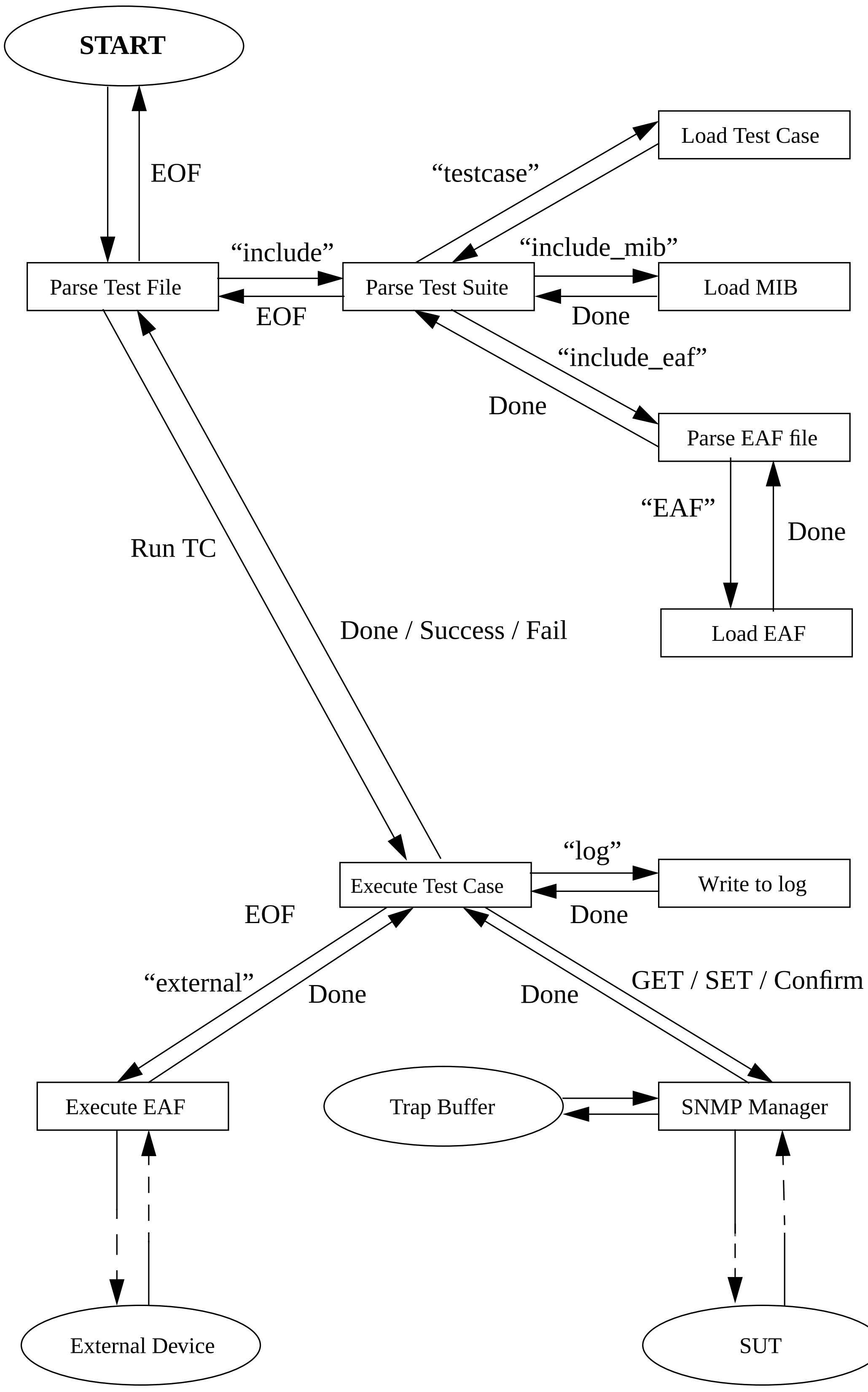


Fig. 9.4: Test execution flow chart

10. Implementation

10.1 Choice of programming language and development tools

When it was time to start working on the implementation, I had a rather clear picture of the model, its components, the way they were supposed to interact and the functionality that was expected. The choice of programming language and development tools to use was mine completely, but I had certain demands that needed to be met:

- I wanted a package / library that could handle SNMP communication at a low level, providing some sort of API. The idea of coding SNMP frame and UDP handling did not appeal to me and, as it turned out, there was no need for me to do that.
- The development tools had to provide means of building a decent graphical user interface.
- The tools would have to be available for the platform in use, namely Solaris 2.5.1.

SNMP packages are available for most common programming languages and platforms, but after performing some brief research, providing me with an overview of the publicly available tools, I ended up with three main implementation alternatives:

- C - using, for instance, the CMU SNMP library.
- Tcl/Tk - using a network management extension for Tcl called Scotty.
- Erlang - using the SNMP agent/management components included in the Erlang Open Telecom Platform (OTP).

My experience with the last two programming languages was limited, to say the least, but they still appeared to me to be the most interesting implementation alternatives. The reason for this was partly that I knew that their respective SNMP packages were functionable and used. In the end I chose to implement the tool in Tcl/Tk, due to several reasons:

- The syntax of Tcl/Tk is easy to learn, code, and read, as is the case with many script languages. It is also suitable to write test cases using a simple script language directly, rather than having to build a parser and interpreter.
- The Erlang OTP SNMP agent part is well used and can be considered robust. This, however, might not apply to the SNMP manager part, which has been intended to be used mainly to test agent implementation. Besides, the OTP SNMP manager lacks direct support for synchronous SNMP communication.

The advantage with an Erlang solution over a Tcl/Tk solution would be that Erlang is well used within the department, implying a wide-spread knowledge. Also, most of the already existing code is written in Erlang. Another thing that might prove a disadvantage with a script language solution is speed of execution. However, for this particular application, there are no performance demands of that kind. One can assume that it is the processing in the SUT and the SNMP communication that are the time consuming factors, not the test tool itself.

Using Tcl/Tk, the test tool model presented in Fig. 9.1 was implemented as illustrated in the figure below:

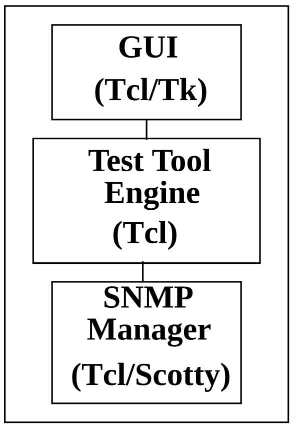


Fig. 10.1: Implementation of the model, using Tcl/Tk

10.2 Implementation description

The prototype I implemented was rather strictly based on the model I have outlined in chapter 9. The modulation is the same, as well as the functionality. The only major difference is the use of the test case specification language. There is no interpreter for that exact syntax in the implementation. Instead, the test cases are written in ordinary Tcl code. This does not imply as big syntactical changes as one might suspect, for more information on how to compose test cases for the implemented test tool, read the corresponding section in Appendix B.

The implementation is a fully-functionable test tool, to be used to test any unit equipped with an SNMP agent and being identified by an IP address. When it comes to portability of the test tool, Tcl/Tk implementations are available for most common platforms. However, the SNMP Manager is implemented using the Scotty Tcl extension, an extension guaranteed to work only on the major UNIX platforms (SunOS, Solaris, HP-UX, AIX, etc.). The Scotty package can be compiled for use on Windows NT systems, but it cannot be expected to behave without problems. Personally, I have only tried the implementation on the platform I have used during development - Solaris 2.5.

The SNMP versions supported by the test tool is SNMPv1 and SNMPv2, since these are the versions supported by Scotty (in Scotty v.2.1.10).

10.2.1 SNMP Interface module

The SNMP interface module is a collection of Tcl procedures providing the user with an extended set of SNMP operations. These procedures, in turn, uses the Scotty package for basic SNMP operations such as GET and SET. This module is not in any way aware of the application using it, and is therefore not restricted to use with the SNMP test tool.

This module handles all the SNMP commands used in the test case description language, all implemented as Tcl procedures: **snmp_get**, **snmp_getnext**, **snmp_set**, **confirm**, **confirm_ne**, **confirm_trap**, **clear_trap_buffer**, **snmp_gettable**, **snmp_table diff**, **snmp_table find**, **snmp_table size**, **snmp_createrow** and **snmp_deleterow**. The **log** and **stimuli** commands are implemented in the TTE module.

10.2.2 TTE module

This module constitutes the ‘kernel’ of the test tool and handles test suite interpretation, test execution and logging. Only the test suites are interpreted to verify syntactical correctness - both test case bodies and test files are written and treated as ordinary Tcl scripts. A procedure **RunTests** executes a test file given as argument and stores the result to a log file, having a default name if not stated otherwise. For example, executing a test file named “myTestfile.tf” would create a log file named “myTestfile.log.1998-12-24-15:01:26”, the file name suffix of course depending on the time of execution.

The only procedure supposed to be used directly by the user is the **tte** procedure. It takes as argument the path to a test file and a number of switches, configuring the SNMP session. Configurable values are:

- agent IP address
- agent UDP port
- TRAP UDP port
- SNMP version
- read and write community string
- log file name

For more information on how to use the test tool without a GUI, see Appendix B.

10.2.3 GUI module

To make the test tool more easy-to-use I extended it with a simple GUI, a task that I am sure was worth the effort. Having a tool that allows you to view / edit test files and test suites, configure settings with a few mouse clicks, to run a test file and have the resulting log displayed immediately on-screen is indeed comfortable. The only procedure a user should use from the GUI module is the **runTTEGUI** procedure. It starts up the test tool environment and reads the user's personal settings which are stored to file the first time he / she uses the tool. When executing tests, the **RunTests** procedure of the TTE module is invoked.

The test tool GUI mode is documented in some detail in Appendix B.

10.3 Differences between model and implementation

The model described in the previous chapter illustrates my suggestion of a tool used to perform automated testing on any entity equipped with an SNMP agent. The implemented prototype does not however follow this model completely, at least regarding the test case specification language. When implementing in Tcl/Tk, I chose to extend the language with new procedures, rather than constructing a parser / interpreter for the command language described in Appendix A. The syntactical deviations are not that extensive, but the consequences are considerable; since the test cases are written using extended Tcl code, the test case composer can use all the functionality that the language offers and even extend the language with new procedures to be used in the test cases. The commands defined in Appendix A all have corresponding procedures in the Tcl implementation, the only major difference being that the command **succeed** is replaced by Tcl's **return** command.

One extension to the original model should be worth mentioning: in the test suite files the statement

```
    source sourcefile
```

is allowed, beside the ones described in the model. This statements makes use of Tcl's built-in **`source`** command, which reads and executes the contents of a Tcl file. This way, the test suite composer can store often used routines in help procedures in separate files.

Included in the implementation is the concept of EAF handling, something that really wasn't necessary. Since the implementation has been extended to include Tcl's **source** command, any Tcl code file can be sourced, including files defining procedures that enables communication over other interfaces than SNMP.

11. Conclusions and future work

11.1 Design properties

In section 8.2 a number of issues to consider when designing this kind of automated test tool were discussed. In this section I will view my test tool model with that list of desirable properties in mind, by discussing each of these issues in order and point out to what extent they have been integrated in my model / implementation.

1. I don’t think I have made the implementation too large and complex, though several procedures in the SNMP interface module are used only in a small subset of the tests I have composed. Using a script language with a reasonable simple syntax should contribute to easy-to-read code and thus further reduce the risk of bugs.
2. The model has been designed to be general, and the implementation is based on that model. Though implemented with a particular target system in mind, all enterprise-specific information is stored separate from the test tool, either in test files / suites or in external access functions.
3. The test execution strategy in my model allows execution of arbitrary test cases, in arbitrary order and from different test suites. It is however up to the test case composer to include initialization and preconditional testing in order to enable the test case to be run in different states. This functionality is not included in the test tool.
4. The test case specification language defined in Appendix A is not very simple, and the Tcl extension used in the implementation is definitely not. On the other hand, the language could not have been abstracted to a level that much higher, without losing functionality. The fact that the test cases are composed directly in Tcl syntax implies that greater programming skills are required, but also that the user is provided with a more powerful test case description platform.

   One might consider building an enterprise-specific test case editor, generating Tcl code, but that would be out of the scope for this project.
5. In the implementation, any exception occurring while running a particular test case will be caught and will cause that test case to abort. Execution will then commence at test file level, i.e. the next test case will be executed.
6. One cannot guarantee that the problem with false positives has been completely avoided. The best one can do is to run exhaustive tests on the implementation and study the source code, especially the parts concerning exception handling.
7. Pausing, aborting and resuming functionality is included in the implementation, but on a higher level. Each test case is executed as an atomic operation, and a user request for pause or abortion will not be addressed until the current test case execution has finished. The reason for this is that in

the implementation, each test case is represented as a Tcl procedure. When running a test case, that procedure is invoked, and control will not be turned over to the test tool until the test case execution is completed.

8. As for now, the test tool generates a log file with a reasonably well-defined format. It is clearly human readable and should without difficulties be readable by machine as well.

11.2 Future extensions

The implementation, as for now, is a fully functionable tool for testing of SNMP controlled equipment. But if it was to be used in a full scale testing environment, one might consider adding a number of functionality extensions and putting some effort into making it into a truly robust, powerful test platform. Possible future extensions could be:

- *Test case debugging* - Upon detecting a failure while executing a test case, one might want to insert break points in the test case code, causing test execution to halt at these points. By continuing test execution step-by-step (i.e. line-by-line) the failure can hopefully be pin-pointed within short. With the current implementation, the test case code is simply extended Tcl code, and the test cases are executed as Tcl procedures. This means that whatever debugging means seem necessary, they have to be provided by the Tcl kernel.

  There are a number of Tcl debugging extensions available that can be used with a Tcl application by recompiling the Tcl/Tk shell, linking the debugger library. The most common one, written by Don Libes, is very similar to well-known debuggers such as gdb, and should be fairly easy to integrate with the test tool.

- *Test case composer tool* - With the current implementation, a test case composer would need at least some basic skills in Tcl programming, and probably a somewhat solid programming background. By allowing the users to compose test cases at a higher layer of abstraction, the tool might prove accessible to more people than just those with extensive programming experience. One could have an enterprise-specific test case composer tool, hiding the low-level Tcl syntax while at the same time offering high-level enterprise-specific commands. This would mean a less general testing platform, and also a less powerful one, but it would make it easier to construct test cases and probably reduce the risk of user-inflicted errors.

- *On-line help* - Adding an built-in help / tutorial is something I haven't found time for during this project. Of course, it would not be hard to implement this feature.

- *Building a library of enterprise-specific help procedures* - Much of the Tcl code used in test cases are identical for several test cases within a suite, or in different suites. By neatly packaging this code in procedures grouped together in files, it would be easier to add new test cases and test suites later on. The test case codes tend to be shorter and easier to read / write, and the risk of user-inflicted errors are reduced.

For instance, one could have a help procedure library with:

- One file per functional testing area (Equipment Management, Fault Management, etc.).
- One file per external device or interface to access (Traffic Generators, Device Processors, etc.).
- Files with general, commonly used procedures.

11.3 Evaluation of model and implementation

At the end of this project there are three questions regarding the work I have done that I might ask myself:

- Are the goals of the project fulfilled?
- Is the model / implementation suitable for general SNMP testing?
- Is it suitable for the primary system?

In this section I will try to answer these questions in order.

Are the goals of the project fulfilled?

The goals of the project was to develop a methodology for automated SNMP testing, including a command language, and to implement a prototype. The fact that the prototype, built on this methodology and basically implementing the command language, works, indicates that the methodology serves its purpose. At the time I'm writing this, the work of automating the major part of a test suite for regression testing with my implementation is undertaken. This implies that the implementation was successful as well.

Is the model / implementation suitable for general SNMP testing?

I have tried to construct a model and implementation that should be able to use with different types of SNMP controlled entities. I have however performed my work with a particular network device in mind and all my testing during development have been done with this system. Thus, there is a risk that I have made my model / implementation less general than I had hoped for.

Is it suitable for the primary system?

As mentioned above, my goal has been to develop a general tool, not one that is adapted to be used exclusively with Ericsson's AXD301 ATM switch. To test this particular system, one might have wished for a more custom-made tool. For example, the test case specification language could have been enterprise-specific, including commands for blocking cards, generating traffic over the system, etc.

Appendix A
Test case specification language syntax

The formal syntax of the test case specification language can be defined using EBNF (Extended Backus Naur Form):

```
testcase ::= testcase atom
             test_statements
             end

test_statements ::= test_statement
                   [test_statements]

test_statement ::= set atom value |
                  if_statement |
                  snmp_statement |
                  snmp_table_statement |
                  external_statement |
                  log log_statement{log_statement} |
                  try_expr test_statement |
                  succeed |
                  fail

if_statement ::= if boolean_expression
                test_statements
                end

snmp_statement ::= snmp_get oid |
                  snmp_getnext oid |
                  snmp_gettable oid |
                  snmp_set oid value |
                  snmp_createrow oid list row |
                  snmp_deleterow oid list |
                  confirm oid value timeout |
                  confirm oid value value timeout |
                  confirm_ne oid value timeout |
                  clear_trap_buffer |
                  confirm_trap value timeout

snmp_table_statement ::=  snmp_table_diff table table
                          snmp_table_find table row
                          snmp_table_size table

external_statement ::= external atom atom
                               “(“[atom{ atom}]”)”

log_statement ::= string |
                “(“snmp_statement”)” |
                “(“snmp_table_statement”)” |
                “(“external_statement”)” |
                var

boolean_expression ::= comparable op comparable |
  try_expr test_statement |
  not “(“boolean_expression”)” |
  “(“boolean_expression”)” and “(“boolean_expression”)” |
  “(“boolean_expression”)” or “(“boolean_expression”)” |

comparable ::= var | string | integer | oid |
              snmp_get oid |
              snmp_getnext oid |
              snmp_table_size table |
              external_statement

table ::= “(“snmp_gettable oid”)” |
          “(“snmp_table_diff table table”)” |
          “(snmp_table_find table row”)” |
          var

row ::= “(“value{ value}”)” | var

oid ::= identifier{“.”identifier}
```

```
identifier ::= digit{digit} |
               var |
               atom

op ::= “<“ | “>” | ==

var ::= “$”atom

atom ::= letter {letter|digit}

list ::= “(“listelement{ listelement}”)”

listelement ::= var|integer|string|oid|atom

value ::= {any_character}

integer ::= “1”|”2”|”3”|”4”|”5”|”6”|”7”|”8”|”9”{digit}

string ::= “”"{any_character}””"

timeout ::= integer | infinite
```

Appendix B
A quick guide to the SNMP Test Tool

Requirements

In order to run the SNMP Test Tool, you will need the following:

- A Tcl/Tk interpreter
- The Scotty Tcl extension
- An SNMP equipped network entity to test. Both this entity and the host from which the test tool is run need to be assigned IP addresses.

The executable Tcl scripts that you will need are in three separate files, though included in the same Tcl package (snmpTestTool):

- “snmpInterface.tcl” (SNMP interface procedures)
- “tte.tcl” (Test Tool Engine procedures)
- “gui.tcl” (Graphical User Interface procedures)

Also, the TCLLIBPATH environment variable should include the path where the snmpTestTool package is located.

Writing test suites and test files

Test cases

A test case has the format

```
testcase name {
        statement
        statement
        ...
}
```

where each *statement* is a Tcl statement, either a Tcl kernel command or a command / procedure call from any extension package, including the SNMP Test Tool procedures. These are:

snmp_get *oid*

Retrieves the value of a MIB variable.

oid - *Object identifier in the format {n1.n2...nN}*

snmp_getnext *oid*

Retrieves the OID of the next MIB variable in lexicographical order.

oid - *Object identifier in the format {n1.n2...nN}*

snmp_set *oid value*

Sets the value of a MIB variable.

oid - *Object identifier in the format {n1.n2...nN}*

value - *New value of object instance*

Ex:
```
% snmp_get 1.3.6.1.4.1.193.14.1.2.4.5.2.1.6.1.1
  enabled
% snmp_get emTable.1.6.1.1
  enabled
% snmp_getnext emTable.1.6.1.1
  1.3.6.1.4.1.193.14.1.2.4.5.2.1.6.1.4
% snmp_set emTable.1.5.1.1 blocked
  1
%
```

confirm *oid value* (*timeout*)

Confirms expected values of one or more MIB variables.

oid - *Object identifier in the format {n1.n2...nN} (or list of OID's).*

value - *Expected value of MIB object instance (or list of values).*

timeout - *Maximum time (in msecs) to wait for values to change to 'values'*

confirm *oid min_value max_value timeout*

Confirms that the value of a MIB variable is in a defined interval.

oid - *Object identifier in the format {n1.n2...nN}*

minValue - *minimum expected value of MIB object instance*

maxValue - *maximum expected value of MIB object instance*

timeout - *Maximum time (in msecs) to wait for value to change to 'value'*

confirm_ne oid value (timeout)

Confirms that the value of a MIB variable differs from a specified value.

oid - *Object identifier in the format {n1.n2...nN}*

value - *unwanted value of MIB object instance*

timeout - *Maximum time (in msecs) to wait for value to change to other than 'value'*

Ex:
```
% confirm emTable 1.5.1.1 blocked 5000
    1
% confirm emTable 1.5.1.1 blocked
    1
% confirm_ne emTable.1.5.1.1 blocked
    Could not confirm "emTable.1.5.1.1 != blocked"
%
```

clear_trap_queue

Clears the TRAP queue.

confirm_trap *trapType* (*timeout*)

Confirms that a number of traps of a certain type are received. Note that the trap buffer should be emptied by the user at appropriate times.

trapTypes - *Types of expected TRAPs.*

timeout - *Maximum time (in msecs) to wait for incoming TRAPs that matches the expected ones.*

Ex:
```
% clear_trap_queue
% confirm_trap eqmEtAlarm
   Could not confirm trap eqmEtAlarm
%
```

external ***device function arguments***

Invokes an external access function and returns the result.

device - *Name of logical device the EAF operates on.*

function - *a external access function (EAF)*

arguments - *List of arguments to the EAF*

Ex:
```
% external HP75000 generateSomeTraffic {1 4000 3}
%
```

snmp_createrow ***tableOID indices row***

Creates a new conceptual row in a MIB table. In the row to create a '?' indicates that no value is specified for that particular variable. Instead, a default value should be provided by the SNMP agent.

tableOid - *Object identifier of a conceptual table*

indices - *list of indices of the new row to create*

row - *list of values for the new row*

snmp_deleterow ***tableOID indices***

Deletes a conceptual row from a MIB table.

tableOid - *Object identifier of a conceptual table*

indices - *list of indices of row to delete*

Ex:
```
% snmp_createrow nsyNodeTable {1} {? new ? ? ? createAndGo}
   1
% snmp_deleterow nsyNodeTable {1}
   1
%
```

snmp_gettable *tableOID*

Retrieves a conceptual MIB table. The returned table will be a list of conceptual rows, where each row is represented as a list of values.

tableOid - *Object identifier of the table*

snmp_getcolumnnames *tableOID*

Returns a list of the names of all columnar objects in a table.

tableOid - *Object identifier of the table*

Ex:
```
% snmp_gettable emTable
    {1 1 cp {} deblocked enabled 1 active}
    {1 4 et34 newEm blocked disabled 1 active}
    {1 19 cp {} deblocked faultyDependent 1 active}
% snmp_getcolumnnames emTable
    emSubrackId emSlotNo emType emUserLabel emAdmState
    emOpState emDpLmList emRowStatus
%
```

snmp_table diff *table1 table2*

Compares two tables (typically retrieved with 'snmp_gettable') and returns the a new table consisting of the rows in table1 *that are not in* table2*.*

table1 - *first table*

table2 - *second table*

snmp_table find *table row*

Matches a row to a table (typically retrieved by snmp_gettable) and returns a new table consisting of all rows in the table that matches the input row. A '?' element in the input row matches any value.

table - *a table*

row - *a list of values*

snmp_table size *table*

Returns the number of rows in a table (typically retrieved by snmp_gettable).

table - *a table*

Ex:
```
% snmp_table diff [snmp_gettable emTable] $oldEmTable
    {1 4 et34 newEm blocked disabled 1 active}
% snmp_table find [snmp_gettable emTable] {1 ? cp ? ? ? ? ?}
    {1 1 cp {} deblocked enabled 1 active}
    {1 19 cp {} deblocked faultyDependent 1 active}
```

```
% snmp_table size [snmp_gettable emTable]
   3
%
```

log ***string***

Ex:

```
% log “This string will be printed to log”
   1
% log “This is a MIB table dump: [snmp_gettable emTable]”
   1
%
```

Note:

The four different commands used to confirm MIB variables and incoming traps all take an argument stating the maximum time to wait for the expected result before a timeout occurs and an exception is raised. Until this time limit is reached or the expected value / trap can be confirmed, the test tool polls the variable / trap queue at regular time intervals. These intervals are controlled by a global Tcl variable - ***time-outInterval****. By setting the value of this variable (default is 2000, unit is msecs), you can control the polling frequency.*

Test suites

A test suite is a stand-alone file containing a number of test case definitions along with MIB, EAF and Tcl source file inclusion statements. Test suite files, which are parsed by the test tool when included by a test file, have four allowed statements:

mib ***mibFile***
include_eaf ***eafFile***
source ***TclSourceFile***
testcase ***name body***

Example of a simple test suite, stored to file “myTestSuite.ts”:

```
####################################################
# myTestSuite.ts
#
# Example of a test suite. These two tests creates and renames,
# a synchronization node.
####################################################

mib “mibs/Axd301Comm-OMS.mib”
mib “mibs/Axd301Nsy-SWS.mib”
mib “mibs/Axd301Eqm-SWS.mib”
include_eaf “eafs/HP75000.eaf”
source “helpProcedures.tcl”

# Testcase 119
#
# Create a node
#
testcase SWS-NS-119 {

   # Precondition - No synchronization nodes or references exists
   if {[snmp_table size [snmp_gettable nsyNodeTable]] != 0} fail
   if {[snmp_table size [snmp_gettable nsyTrafficalRefTable]] != 0} fail
   if {[snmp_table size [snmp_gettable nsyDedicatedRefTable]] != 0} fail

   # Action - create a new synchronization node
   log "Creating new node..."
   snmp_createrow nsyNodeTable {1} {? newNode ? ? ? createAndGo}

   # Postconditions - confirm correct mode and operational state
   log "Confirming mode and operational state..."
   confirm nsyNodeTable.1.3.1 freerunning
   confirm nsyNodeTable.1.5.1 disabled

   # Postcondition - confirm incoming trap, should be received within 5 seconds
   log "Confirming trap..."
   confirm_trap nsySynchNodeNotWorkingAlarm 5000
}


# Testcase 120
#
# Rename a node
```

```
#
testcase SWS-NS-120 {

   # Action - rename the synchronization node
   log "Renaming node..."
   set old_name [snmp_get nsyNodeTable.1.2.1]
   snmp_set nsyNodeTable.1.2.1 $old_nameextended

   # Postcondition - confirm name change
   log "Confirming name change..."
   confirm nsyNodeTable.1.2.1 $old_nameextended
}
```

Test files

A single test file is the input given to the test tool. It describes a sequence of test cases to execute and under which conditions (by setting variables / parameters) to execute them. While test case definitions and test suites remain static over longer time periods, test files can be subject of changes from one test run to another. Test files are written in pure Tcl code, extended with two types of procedures:

include *testSuiteFile*
testCase

Example of a small test file:

```
####################################################
# This test file runs a small number of tests from two
# different test suites.
####################################################

include myTestSuite.ts
include anotherTestSuite.ts

# Create a node
SWS-NSY-119

# Rename the node
SWS-NSY-120

# Run a number of equipment tests for boards in different slots
for {set slot 2} {$slot < 18} {incr slot} {
        SWS-EQM-211
        SWS-EQM-301
        SWS-EQM-089
    }
```

Starting the test tool

The test tool can be started either with or without the GUI. In any case, you will have to start up a Tcl or Tk shell first.

Without GUI

This command line mode has no features not included in the GUI version, and will only be discussed briefly in this chapter. However, a short introduction might be suitable.

- Start the Tcl shell:
  ```
  system> tclsh8.0
  ```
- Add the SNMP Test Tool package:
  ```
  % package require snmpTestTool
  ```
- Run the procedure named tte, if you don’t supply any arguments a syntax description will be displayed:
  ```
  % tte
  Usage: tte [switches] <file>

  -l <logfile>        - write test results to
                        specified logfile
  -ip <IP address>    - SNMP agent IP address
  -udp <UDP port>     - SNMP agent UDP port
  -tudp <UDP port>    - SNMP TRAP port
  -v <SNMP version>   - SNMP version,
                        either SNMPv1 or SNMPv2c
  -c <community>      - read community string
  -wc <community>     - write community string
  -a <testcases>      - abort after a number of
                        failed test cases
  %
  ```

If you, for instance, would want to test a system identified by IP address 130.100.180.111, having an SNMP agent residing at port 4012 with the test sequence listed in file “mytests.tf”, you would type:

```
% tte -ip 130.100.180.111 -udp 4012 mytests.tf
```

All switches except for agent IP address has default values which are set to:

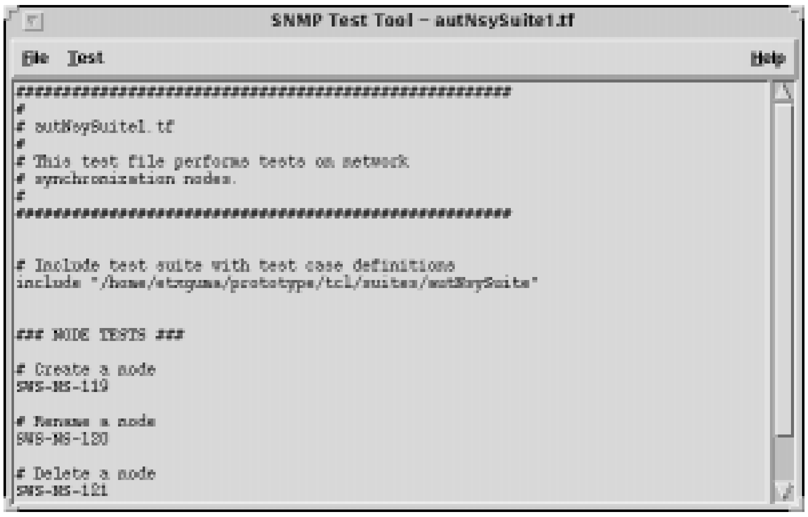
log file:	<input_filename>.log
agent UDP port:	161
TRAP UDP port:	162
SNMP version:	SNMPv2c
read community:	public
write community:	public

In GUI mode

- Start the wish shell:
  `system> wish8.0`
- Add the SNMP Test Tool package:
  `% package require snmpTestTool`
- Start the application:
  `% runTTEGUI <testfile>`

The SNMP Test Tool environment will now appear on screen, allowing you to open and edit test files and test suites, setting your personal preferences and, of course, run tests.

Choose **Open** from the **File** menu to open a test file. Once loaded, the test file will be displayed in the text editor, allowing you to edit the contents.



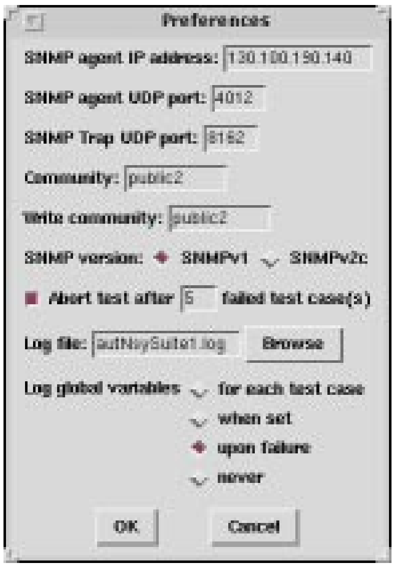
```
############################################################
#
# autNsySuite1.tf
#
# This test file performs tests on network
# synchronization nodes.
#
############################################################


# Include test suite with test case definitions
include "/home/etxguma/prototype/tcl/suites/autNsySuite"


### NODE TESTS ###

# Create a node
SWS-NS-119

# Rename a node
SWS-NS-120

# Delete a node
SWS-NS-121
```

Fig. B.1: SNMP Test Tool main window

Configuring preferences

The GUI provides you with a quick and easy way to inspect and configure testing settings. The first time you start the tool in graphical mode, a file .ttecfg is created in your home directory. This file stores information on all user specific settings and is updated every time the you change a preference.

To configure preferences, choose **Preferences** from the **Test** menu. A new window will appear:



SNMP agent IP address: 130.100.190.140
SNMP agent UDP port: 4012
SNMP Trap UDP port: 8162
Community: public2
Write community: public2
SNMP version: [✓ SNMPv1] SNMPv2c
[✓] Abort test after 5 failed test case(s)
Log file: autNoySuite1.log
Log global variables: for each test case / when set / [✓ upon failure] / never

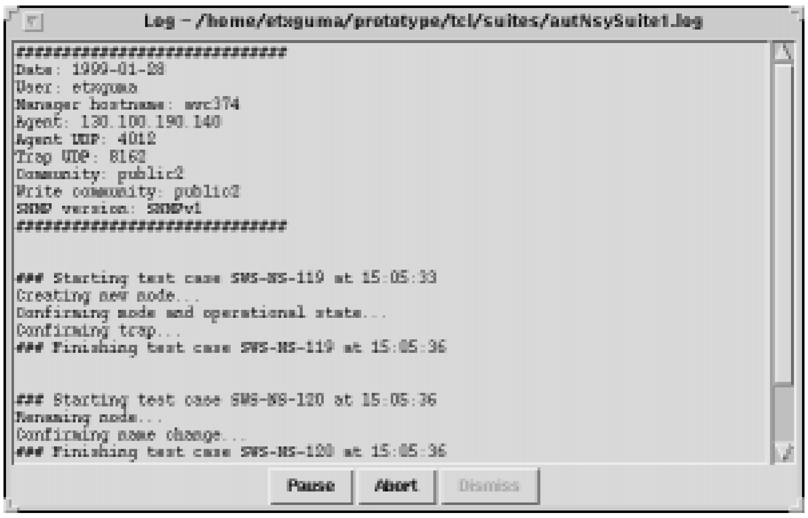
Fig. B.2: Preferences window

Change the preferences of interest and click **OK**. The updated preferences will now be saved to the personal configuration file.

Running tests

Once a test file is open and you are satisfied with your preferences, you can start the test execution by choosing **Run** from the **Test** menu. Note that it is the contents of the text editor that will be executed. Thus, if you have edited the test file you don't have to save it before running.

When the test execution starts, a log window will appear. The contents of this window will be identical to that of the log file saved to disk. Using the buttons at the bottom of the window you have the possibility to pause, resume and abort the execution. If you choose to pause or abort, the test tool will finish executing the current test case before halting.



```
##############################
Date: 1999-01-28
User: etxguma
Manager hostname: avc374
Agent: 130.100.190.140
Agent UDP: 4012
Trap UDP: 8162
Community: public2
Write community: public2
SNMP version: SNMPv1
##############################


### Starting test case SWS-NS-119 at 15:05:33
Creating new node...
Confirming mode and operational state...
Confirming trap...
### Finishing test case SWS-NS-119 at 15:05:36


### Starting test case SWS-NS-120 at 15:05:36
Renaming node...
Confirming name change...
### Finishing test case SWS-NS-120 at 15:05:36
```

Fig. B.3: Log window

Note:

A convenient way to create an executable application for the SNMP Test Tool on UNIX platforms is to use shell scripts:

```
#!/usr/local/tclsh8.0
package require snmpTestTool
tte $argv
```

and

```
#!/usr/local/wish8.0
package require snmpTestTool
runTTEGUI $argv
```

respectively.

Acronyms

AMS	AXD301 Management System
ASN.1	Abstract Syntax Notation One
ATM	Asynchronous Transfer Mode
CCITT	International Telegraph and Telephone Consultative Committee
CMIP	Common Management Information Protocol
CMOT	CMIP over TCP/IP
CNH	Connection Handling
CP	Control Processor
DP	Device Processor
DPCI+	Device Processor Control Interface +
EAF	External Access Function
EGP	Exterior Gateway Protocol
EQM	Equipment Management
FTM	Fault Management
GUI	Graphical User Interface
HEMS	High-Level Entity Management System
HMP	Host Monitoring Protocol
IAB	Internet Architecture Board
ICMP	Internet Control Message Protocol
ITU	International Telecommunication Union
IP	Internet Protocol
MAC	Medium Access Control
MIB	Management Information Base
NSY	Network Synchronization
OID	Object Identifier
OSI	Open Systems Interconnection
PING	Packet Internet Groper
PDU	Protocol Data Unit
PRM	Performance Management
RPC	Remote Procedure Call
RSH	Remote Shell
SGMP	Simple Gateway Monitoring Protocol
SMI	Structure of Management Information

SNMP	Simple Network Management Protocol
SUT	System Under Test
SWS	Switching Subsystem
TCP	Transmission Control Protocol
TTE	Test Tool Engine
UDP	User Datagram Protocol
VC	Virtual Channel

References

[1] Stallings, William, *SNMP, SNMPv2 and RMON: practical network management* 2nd ed. 1996, Addison-Wesley Pub Co

[2] Perkins, D., McGinns, E., *Understanding SNMP MIBs* 1997, Prentice Hall

[3] Black, Uyless, *Network Management Standards / the OSI, SNMP and CMOL protocols* 1992, McGraw Hill Text

[4] Harnedy, Sean, *Total SNMP: Exploring the Simple Network Management Protocol* 2nd ed. 1998, Prentice Hall

[5] Feit, Sidnie, *SNMP: A Guide to Network Management* 1995, McGraw Hill Text

[6] Rose, M., McCloghrie, K., “Structure and Identification of Management Information for TCP/IP-based Internets”, RFC 1155, may 1990

[7] Case, J., Fedor, M., Schoffstall, M., Davin, J., “The Simple Network Management Protocol”, RFC 1157, May 1990

[8] Rose, M., McCloghrie, K., “Concise MIB Definitions”, RFC 1212, March 1991

[9] Case, J., McCloghrie, K., Rose, M., Waldbusser, S., "Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)", RFC 1905, January 1996

[10] Case, J., McCloghrie, K., Rose, M., Waldbusser, S., "Coexistence between Version 1 and Version 2 of the Internet-standard Network Management Framework", RFC 1908, January 1996.

[11] Waldbusser, S., "Remote Network Monitoring Management Information Base", RFC 1757, February 1995

[12] Case, J., Mundy, R., Partain, D., Stewart, B., “Introduction to Version 3 of the Internet-standard Network Management Framework”, Internet draft, July 1998

[13] “Network Management Basics”, http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/55018.htm, Cisco Systemc Inc., 1995

[14] Zallar, Kerry, “Automated Software Testing - A Perspective”, http://www.crl.com./~zallar/autotest.html

[15] “Automatic test suite generation”, http://www.bonnell.com/mbi/isac/autogen.html, Mount Bonnell Inc., 1998

[16] Pettichord, Bret, “Success with Test Automation”, http://www.io.com/~wazmo/succpap.htm, 1996

[17] Powers, Mike, “Styles for Making Test Automation Work”, http://www.stlabs.com/testnet/docs/auto_style.htm, ST Labs Inc., 1997

[18] Kaner, Cem, “Improving the Maintainability of Automated Test Suites”, http://www.kaner.com/lawst1.htm, 1997

[19] Bach, James, “Useful Features of a Test Automation System”, http://www.stlabs.com/testnet/docs/TPFEAT.HTM, ST Labs Inc., 1996

[20] Bach, James, “Test Automation Snake Oil”, http://www.stlabs.com/testnet/docs/snakeoil.HTM, ST Labs Inc., 1996

[21] Marick, Brian, “Classic Testing Mistakes”, http://www.stlabs.com/marick/Classic/MISTAKES.html, ST Labs Inc., 1997

[22] Hancock, James, “When to Automate Testing”, http://www.stlabs.com/testnet/docs/jimauto.htm, ST Labs Inc., 1998

[23] Ousterhout, John K., *Tcl and the Tk Toolkit* 1994, Addison-Wesley Pub Co

[24] Johnson, Ray, “Tcl Style Guide”, Sun Microsystems Inc., August 1997