Specification and Testing of the Behavior of Network Management Agents Using SDL-92

Olaf Henniger, Michel Barbeau, *Member, IEEE*, and Behçet Sarikaya, *Senior Member, IEEE*

Abstract— **We develop a method for specifying the behavior of network management agents for the Internet framework using SDL-92 which is object-oriented. The starting point is the definition of a management information base (MIB) in ASN.1 with macros as well as the description of the simple network management protocol version 2 (SNMPv2 protocol). The behavior of an agent is defined in an SDL-92 process which dynamically creates processes for the rows of the tables of the MIB and in several procedures that process variable bindings in protocol data units (PDU's) and carry out specific operations on conceptual rows such as creation, retrieval, and bulk retrieval. Some reusable, MIB-independent process types and procedures are identified. For large MIB's, we first obtain a class diagram representation of the MIB. From the class diagram, the classes with behavior are mapped to process types representing conceptual rows of specific tabular objects. SNMPv2 protocol operations on these tabular objects can be easily specified with the help of MIB-independent procedures. Next, considered is the generation of test cases. MIB test cases can be generated from the SDL-92 specification using specification analysis and behavior inversion techniques. Many test cases for an example single-table MIB are designed and specified in TTCN based on the specification in SDL-92.**

I. Introduction

THIS PAPER deals with the application of a systematic test design methodology to the new field of conformance testing of network management systems. The context of this work is version 2 of the simple network management protocol (SNMPv2) framework of the Internet engineering task force (IETF) [12]. The paper is concerned with the specification of managed node components, namely, network management agents and management information bases (MIB's) as well as with the problem of testing the conformance of managed node implementations to network management standards, i.e., to requests for comments (RFC's) [4]–[6] emitted by the IETF. The conformance testing problem is twofold. The first issue is the development of test suites to be applied to the implementations while the second issue is the development of a test system that effectively exercises the implementations with the test suites. This paper addresses solely the first issue and discusses a test design approach specifically for that purpose.

A systematic approach to the test design problem for managed nodes, such as the one we propose, is relevant because this is a means of obtaining high quality software. Quality here means conformance to specifications, which is required for interoperability of communicating system, and robustness. Testing of managed nodes means examining their behavior, i.e., their reaction to different inputs. The behavior of managed nodes is far from being explicit in IETF RFC's. Systematic test generation should begin with a very clear picture of the behavior to be tested. Therefore, our methodology consists of two major activities, namely, formal modeling of behavior of managed nodes and then test suite generation.

We advocate object-oriented modeling of the structure and behavior of managed nodes. The input of this activity is the content of the relevant IETF RFC's, i.e., ASN.1 and natural language descriptions of MIB's and of agents. From these descriptions, a more abstract representation consisting of a class diagram which uncovers the structural aspects and an SDL-92 [10] model which encompasses the structure and behavior are produced.

The test design activity follows ISO's conformance testing methodology and yields test suites precisely expressed in the tree and tabular combined notation (TTCN) [9].

In Section II, we discuss formal modeling of the structure and behavior of network management agents with a single-table MIB using SDL-92. In Section III, the approach is illustrated with a recently developed MIB for ATM virtual links. In Section IV, the test suite development activity is presented. Finally, we give our conclusions in Section V.

II. Formal Modeling of Agents and MIB's

We develop a method for obtaining a specification in SDL-92 of the behavior of an agent entity. There are reusable parts that are independent of a specific MIB. The data type definitions (PDU's, rows in MIB tables, etc.) in the specification are obtained from the ASN.1 definitions given in the relevant RFC's. The behavior definition is obtained from the informal prose descriptions given in [4]–[6]

There are several approaches for bringing together ASN.1 data type definitions and SDL: modeling of ASN.1 definitions in the data type notation of SDL-92 [2], inclusion of ASN.1 definitions as external data description [7], combined use of SDL-92 and ASN.1 as in the SDL dialect defined in [11]. We have opted for modeling the ASN.1 definitions in the data type notation of SDL-92, i.e., ASN.1 syntax is not

Manuscript received July 18, 1995; revised April 3, 1996; approved by IEEE/ACM TRANSACTIONS ON NETWORKING Editor V. Sahin. O. Henniger was supported in part by a scholarship granted by the German Academic Exchange Service (DAAD).

O. Henniger is with GMD, the German National Research Center for Information Technology, Darmstadt, Germany (email: henniger@darmstadt.gmd.de).

M. Barbeau is with the University of Sherbrooke, Québec, Canada (email: michel.barbeau@dmi.usherb.ca).

B. Sarikaya is with the University of Aizu, Aizu-Wakamatsu, Fukushima, Japan (email: sarikaya@u-aizu.ac.jp).

Publisher Item Identifier S 1063-6692(96)07567-X.

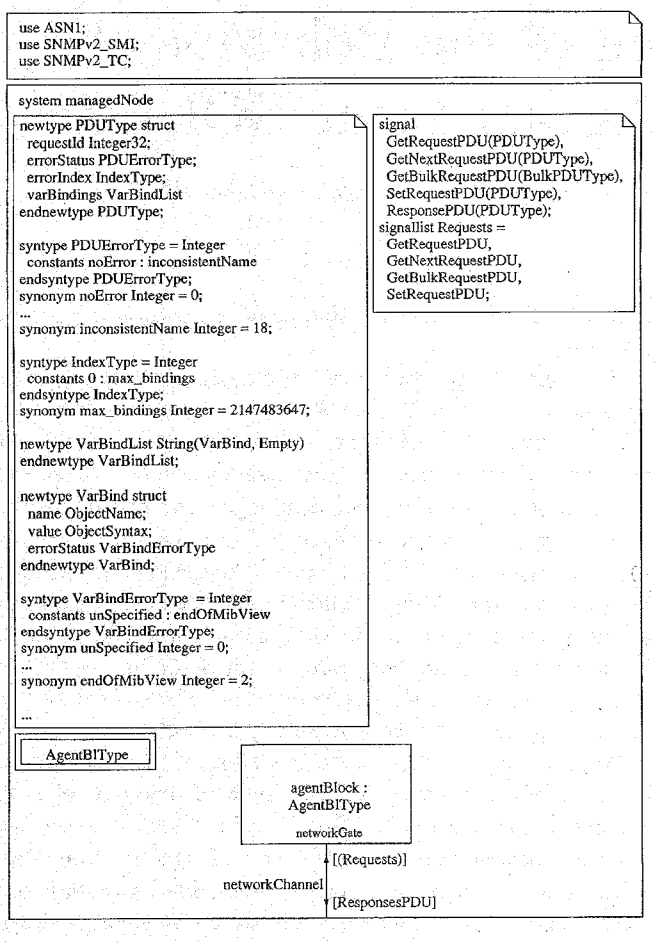


Fig. 1. System diagram.

used in the final specification of the network management agent. This is justified because only a subset of ASN.1 containing only predefined SDL-92 data and extended data definition constructs like structure sorts is used in the SNMPv2 framework. Complicated ASN.1 abstract data types are not required.

Predefined ASN.1 types that are used in SMI (like *OBJECT IDENTIFIER*, *OCTET STRING*, and *BIT STRING*) and that do not have a direct counterpart in SDL-92 (for example, *INTEGER* has) are mapped to SDL-92 data types. For instance, *OBJECT IDENTIFIER* is modeled as a string of *ObjIdComponents* which are positive integers. The SDL-92 package called *ASN1* contains all such types. We mapped the types defined in the ASN.1 module *SNMPv2-SMI* [5] (e.g., *ObjectName*, *Integer32*, *IpAddress*, and *Counter32*) to SDL-92 data types and included them in another SDL-92 package called *SNMPv2_SMI*. We mapped the types defined in the ASN.1 module *SNMPv2-TC* [6] (e.g., *RowStatus*) to SDL-92 data types and included them into the SDL-92 package called *SNMPv2_TC* (see Appendix).

A. Single Table MIB Example

A managed node stores a collection of managed objects (i.e., an MIB) which are abstractions of logical or physical resources of the managed node. IETF has developed the structure of management information (SMI) for modeling such data [5].

SMI is based on the abstract syntax notation one (ASN.1) [8], [15]. ASN.1 is used to define a template for managed object types. The template demands the data type of the value of a managed object, the allowed kind of access (reading, writing, or creating), and a textual description. The data type is either simple or at most a two-dimensional (2-D) array of simple type elements. Each instance of the template, i.e., each object type, has a unique identity, its object identifier, which is a sequence of nonnegative integers. We use the following example [14]:

```
grokIndex OBJECT-TYPE
  SYNTAX        INTEGER
  MAX-ACCESS    not-accessible
  STATUS        current
  DESCRIPTION
    "The auxiliary variable used to identify
    instances of the columnar objects in the
    grok table."
  ::= { grokEntry 1 }
```

Objects of the *grokIndex* type are of the simple data type *INTEGER*. Simple objects may be grouped into a table as follows:

```
grokEntry OBJECT-TYPE
  SYNTAX        GrokEntry
  MAX-ACCESS    not-accessible
  STATUS        current
  DESCRIPTION
     "An entry (conceptual row) in the grok
     table."
  INDEX         { grokIndex }
  ::= { grokTable 1 }

GrokEntry ::= SEQUENCE {
  grokIndexINTEGER,
  grokIPAddrIpAddress,
  grokCountCounter32,,
  grokStatuRowStatus }
grokTable OBJECT-TYPE
  SYNTAX   SEQUENCE OF GrokEntry
  MAX-ACCESSot-accessible
  STATUS   current
  DESCRIPTIOThe (conceptual) grok table."'
  ::= { adhocGroup 2 }
```

The definition of *GrokEntry* describes the contents of a table row. Each row consists of an index value *grokIndex*, an IP address *grokIPAddress*, a counter *grokCount* of the number of packets sent to that IP address *grokIPAddress*, and a row status *grokStatus* (used in the process of row creation and deletion).

B. Structural Aspects

In SDL-92, the top level entity is a system. Fig. 1 shows a diagram of a system called *managedNode*. The upper rectangle contains imports of packages. The largest rectangle represents the structure of *managedNode* itself. It is made of data type declarations (upper left rectangle), signals and signal list declarations (upper right rectangle), a block type

named *AgentBlType*, and an instance of it named *agentBlock*. The instance *agentBlock* is connected to the environment through a channel, named *networkChannel* and labeled with a signal (*ResponsePDU*) and a list of signals (*Requests*) flowing through it.

The block type diagram of *AgentBlType* is shown in Fig. 2. The block type contains the abstract process types *AgentPrType*, modeling the SNMPv2 protocol operations, and *RowType*, modeling the behavior of a generalized row. *AgentPrType* is specialized for the specific MIB of our example in the process type *GrokMIBAgentPrType*. *RowType* is specialized into *GrokEntryType* for rows of the table *grokTable* of the specific MIB.

When the system is initialized, one instance, named *agent*, of process type *GrokMIBAgentPrType* is created. Initially, there are no instances of the process type *GrokEntryType*. They are created dynamically by the *agent* (dashed arrow relation in Fig. 2). The *agent* receives SNMP PDU's via its gate *networkGate*, carries out error checking, and distributes signals to the rows via its gate *queryGate*. The *agent* can activate, deactivate, or destroy conceptual rows as well as read or write the values of the elements contained in the rows.

C. Modeling of PDU's

In [4], the SNMP PDU's are defined in an ASN.1 module. All PDU types, except for *GetBulkRequest-PDU*, have the same simple structure:

```
PDU ::= SEQUENCE {
    request-id Integer32,
    error-status INTEGER {
        noError(0),
        ...
        inconsistentName(18)},
    error-index INTEGER (0..max-bindings),
    variable-bindings VarBindList}
max-bindings INTEGER ::= 2147483647
VarBindList::=SEQUENCE
    (SIZE (0..max-bindings)) OF VarBind
VarBind ::= SEQUENCE {
    name ObjectName,
    CHOICE {
        value ObjectSyntax,
        unSpecified NULL,
        noSuchObject[0] IMPLICIT NULL,
        noSuchInstance[1] IMPLICIT NULL,
        endOfMibView[2] IMPLICIT NULL}}
```

Each PDU carries a list of variable bindings *VarBindList*. Each variable binding *VarBind* is made up of an object name *name* which may be paired with a value *value* (may be *unspecified*) or an error indication, either *noSuchObject*, *noSuchInstance*, or *endOfMibView*.

PDU's in SDL-92 are modeled as the type *PDUType*, see upper left box in Fig. 1. The mapping is straightforward. A variable binding list in SDL-92 is a one-dimensional (1-D) array of structures of sort *VarBind* consisting of an object name *name*, an object value *value*, and an error status *errorStatus*.

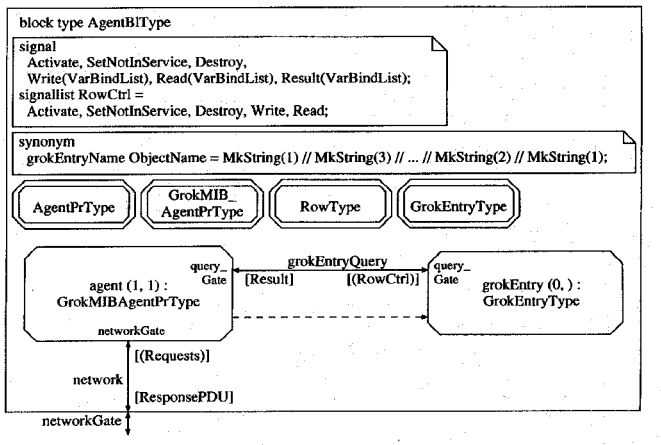


Fig. 2. Block type diagram for the example.

D. Modeling of Managed Objects

There are different types of managed objects: simple managed objects (e.g., *grokStatus*), rows (e.g., *grokEntry*), and tables (e.g., *grokTable*). A behavior is specified in terms of states and transitions only for rows. Simple objects and tables do not have a behavior.

Rows are mapped to SDL-92 process types, see Fig. 3. Columnar objects are mapped to SDL-92 variables of the types provided in the ASN.1 definitions. Access to the managed objects and their default values are modeled in the SDL-92 procedures for accessing and initializing conceptual row process instances.

Tables are not explicitly mapped to SDL-92 constructs. They correspond to sets of row processes. For example, *grokTable* is represented by a set of *grokEntry* processes in the block of type *GrokEntryType* (Fig. 2).

E. Modeling of Conceptual Rows

The process type *RowType* models the behavior of a generalized row. Its diagram is pictured in Fig. 3.

Rows represent dynamic entities. In SMI, the states which a row goes through are coded in a field of type *RowStatus* (see Section II-A). The values of *RowStatus* are interpreted as in Table I. Some of these values represent states of instances, some represent actions on instances, and others represent both states and actions.

Conceptual rows can be created and destroyed upon the request of a manager. Creation of a conceptual row is initiated by a SetRequest-PDU carrying the value *createAndGo* or *createAndWait* for a conceptual row's variable of type *RowStatus*. Value *createAndGo* is used for single step creation, i.e., all the values of the fields of the row are provided in a single set-request operation. Value *createAndWait* is for negotiated creation during which values of components are written one after the other, thus allowing detailed error checking. Destroying a conceptual row is initiated by a *SetRequest-PDU* carrying the value *destroy* for the variable of type *RowStatus*.

In the process type diagram *RowType* (Fig. 3), there are three states: *notReady*, *notInService*, and *activated*. Initially, an

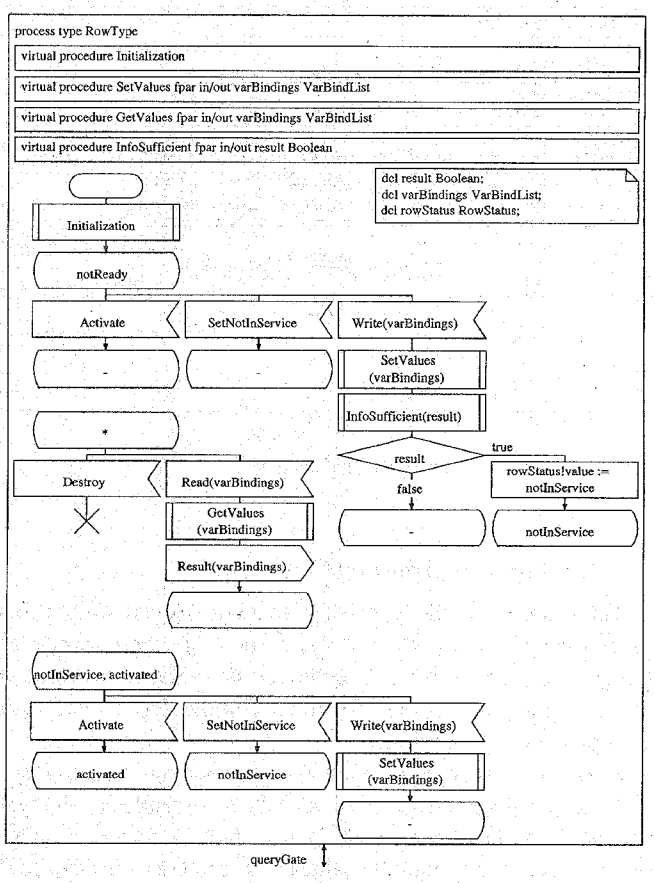


Fig. 3. Process type diagram *RowType*.

TABLE I
INTERPRETATION OF ROW STATUS VALUES

Value	Name	Interpretation
1	*active*	state and action
2	*notInService*	state and action
3	*notReady*	state
4	*createAndGo*	action
5	*createAndWait*	action
6	*destroy*	action

object is created and put into the *notReady* state. All objects for which no default value is defined have to be set in a set-request operation before the state *notInService* can be reached. If all required values are present, then the procedure *InfoSufficient* returns *true*. In the state *notInService/activated*, the conceptual row can be activated/deactivated. In any state, the conceptual row can be destroyed and values can be read or written.

As shown in Figs. 9 and 10, the process type *RowType* is inherited by process types defining specific rows. In the subtypes, the virtual procedures *Initialization*, *SetValues*, *GetValues*, and *InfoSufficient* are redefined to adapt the process type definition to the specific rows. Furthermore, a formal parameter *index* of type *Integer32* is added to pass the index value to the row. The index cannot be set directly by *a SetRequest-PDU* because the allowed kind of access is nonaccessible. The index value is part of the object name for the column instances of row. When a row is created, the index

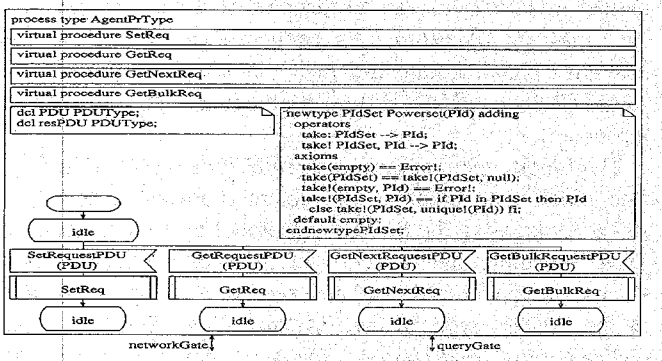


Fig. 4. Process type diagram *AgentPrType*.

value is retrieved from the object name and used as actual parameter for the created process instance.

F. Modeling of the SNMPv2 Protocol Operations

SNMPv2 protocol operations are modeled in the process type diagram *AgentPrType* (Fig. 4). According to the received input signal, a procedure for the corresponding protocol operation is invoked.

As shown in Figs. 11 and 12, the process type *AgentPrType* is inherited and specialized by MIB-specific process types. Process type *GrokMIBAgentPrType*, a part of which is shown in Fig. 11, inherits and specializes *AgentPrType* to adapt it to the *grokTable* example. For instance, the behavior of the procedure *SetReq* is redefined in the process type *GrokMIBAgentPrType*. The procedure *SetReq* invokes the procedure *GrokEntrySetReq*, which is a specialization of the procedure *RowSetReq* defined in Fig. 5.

The first step in processing an SNMP PDU is to find sublists in the variable-binding list that are destined for the same table and the same row. PDU's received from a manager have to be relayed to the managed objects addressed by the PDU. A simple method of dynamic process creation is adopted. A new process is created for each row. The process id values (*PId*'s) of the row process instances belonging to the same table are stored in a set of type *PIdSet*, e.g., in the set *grokEntries* defined in Fig. 11 for our *grokTable* example. Before creating a new row process instance, the procedure *FindPId* is called which returns in its in/out-parameter either a *null* PId (if the row has not yet been created) or the PId of the row process. After creation, the process identifier of the newly created process instance is included into the set *grokEntries*.

The procedure *FindPId* provides the connecting link between the PIdsets and the index values (Fig. 13). In this procedure, the index values given in the PDU are compared with the index values of the existing conceptual row process instances. The index values are visible via *view* expressions.

Processing of PDU variable bindings is done as follows: the procedure *FindRowStatus* returns a pointer to the next row status and row name (Fig. 13). It is assumed that the row status occurs first in the variable bindings list in order to simplify the search. For single-table MIB's, there is only one row name, e.g., *grokTable*. *FindRowStatus* can be called repeatedly to completely process the variable bindings and

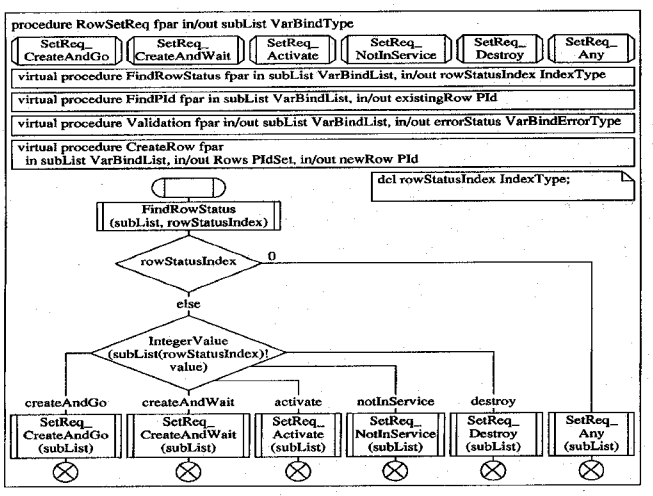


Fig. 5. Processing of a *SetRequestPDU*.

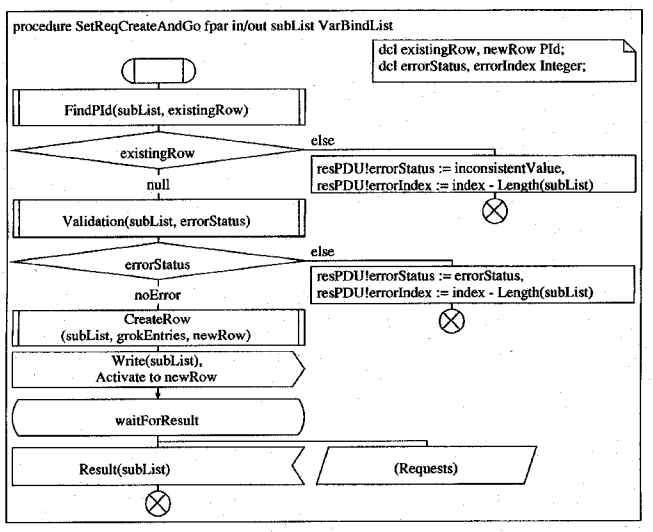


Fig. 6. Processing of a *SetRequestPDU* with *rowStatus* set to *createAndGo*.

prepare a response PDU. When there are no more variable bindings the response PDU is sent (Fig. 11).

Processing of a *SetRequestPDU* (Fig. 5) depends on the row status value. Fig. 6 shows the procedure diagram for processing a *SetRequestPDU* with *rowStatus* set to *createAndGo*. In Fig. 6, *FindPId* is called first. An error occurs if it does not return a *null* PId. Otherwise the procedure *Validation* checks whether the values given in the PDU are consistent and returns *noError* if so. If the values are consistent, a new conceptual row is created by the *CreateRow* procedure which is defined in Fig. 13; its process identifier is saved, and an *Activate* and a *Write* signal are sent to the newly created process.

The *Write* signal works as follows on the row process (Fig. 3): the procedures *SetValues* and *InfoSufficient* defined in Fig. 9 are invoked. The procedure *SetValues* sets the values in a given row according to the access restrictions. For *grokTable*, *grokIpAddress* and *RowStatus* can be set, but *grokCount* can only be read using the *GetValues* procedure. The procedure *InfoSufficient* checks if the required values are present. For *grokTable*, the *grokIPAddress* is the only required value.

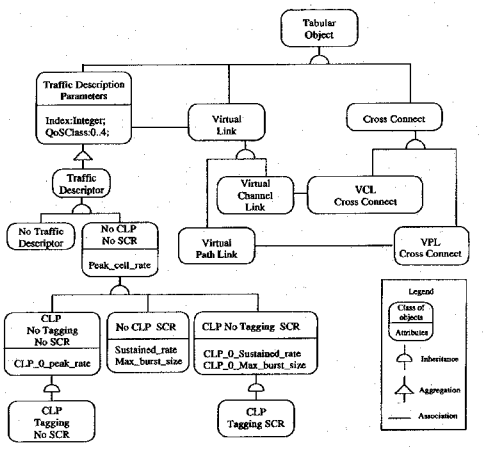


Fig. 7. Class diagram of ATM MIB.

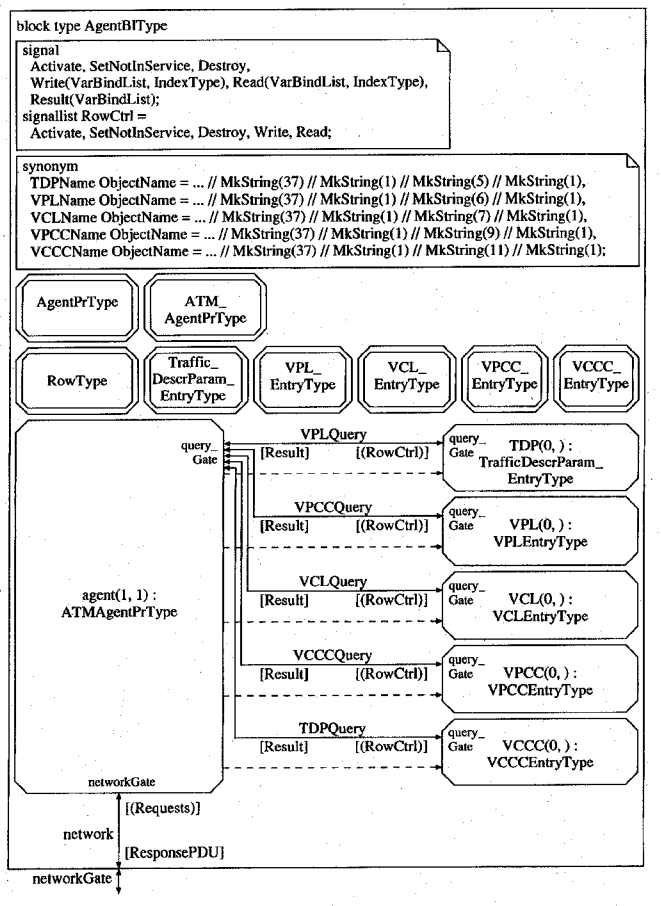


Fig. 8. Block diagram of a subset of ATM MIB.

Depending on the value of *RowStatus* in the variable binding list, *RowSetReq* calls one of the functions *SetReqCreateAndGo*, *SetReqCreateAndWait*, etc. As an example the definition of *SetReqCreateAndGo* is shown in Fig. 6.

III. Advanced Modeling of MIB's

MIB's used in practice contain several managed objects which are organized into groups corresponding to different functional aspects of the managed system. In case of a

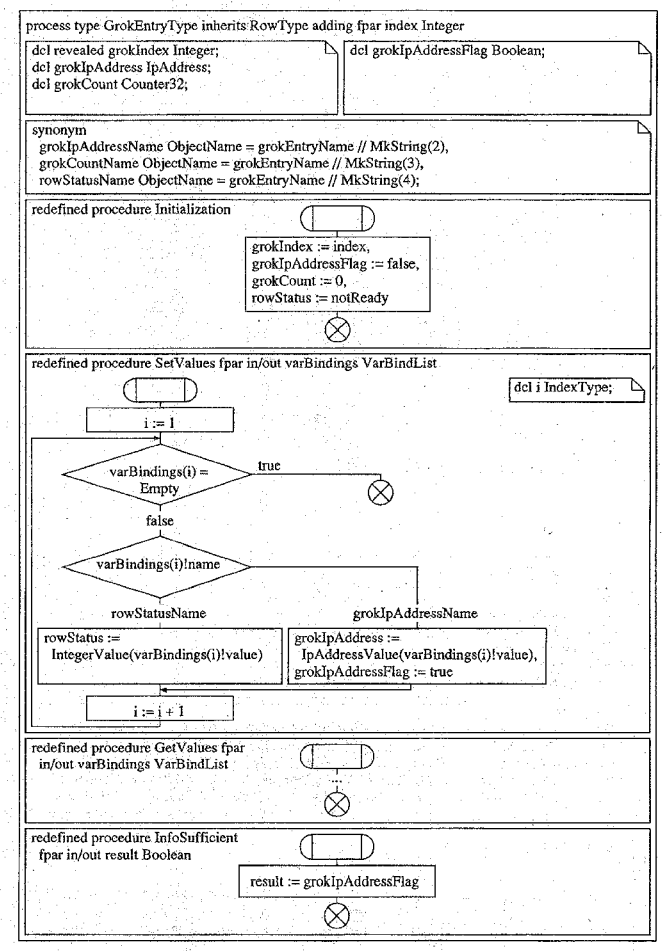


Fig. 9. Process type diagram *GrokEntryType*.

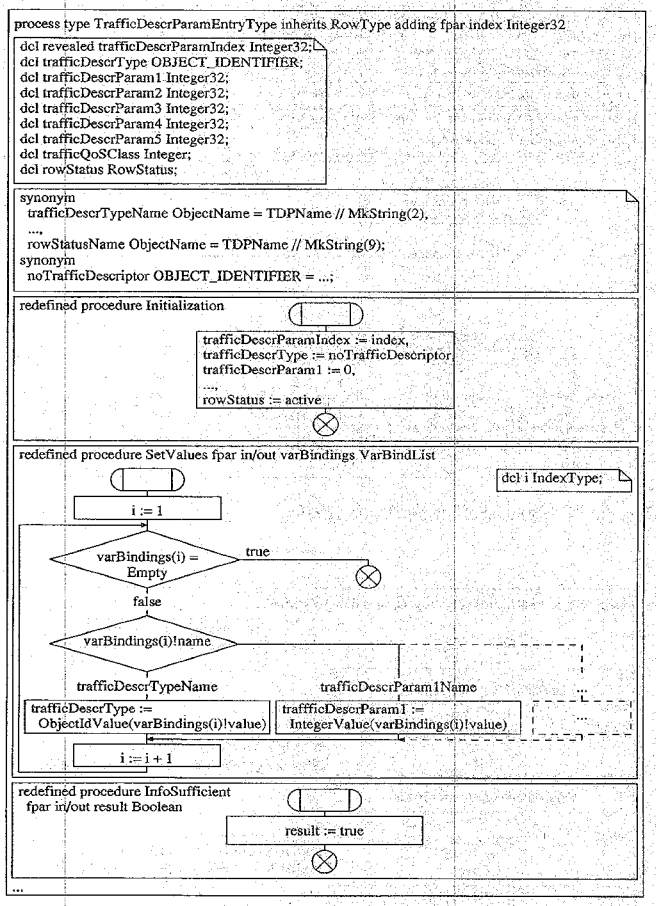


Fig. 10. Process type diagram *TrafficDescrParamIndex*.

large, complex MIB, we use class diagrams [3] to model the conceptual structure of the MIB. Using class diagrams, we can show common aspects of objects and specializations of these common aspects. Common aspects are represented in superclasses and the specializations are represented as subclasses.

We first shortly introduce the asynchronous transfer mode MIB (ATM MIB) which will be used as an example in this section. Next, various steps of the modeling are explained.

A. ATM MIB

ATM is a networking technology to be used over fiber optic transmission lines to transmit short packets called cells in a connection-oriented manner. In an ATM network, there are two kinds of virtual connections, namely, switched virtual connections and permanent virtual connections. The purpose of the ATM MIB is primarily to manage permanent bidirectional virtual connections. Segments of virtual connections inside ATM networks or switches are managed. There are two categories of permanent virtual connections, i.e., permanent virtual channel connections (VCC's) and permanent virtual path connections (VPC's). A VCC is carried by a VPC. Conversely, a VPC can carry several VCC's. A VCC/VPC is made of virtual channel/path links (VCL/VPL's) cross connected together.

B. Derivation of a Class Diagram from an MIB

A class diagram is a graphic notation for object-oriented analysis and design. A class diagram shows the classes of objects, subtyping relationships among classes (i.e., inheritance), containment relationships (i.e., aggregation), and other associations among objects.

Row object types defined in a MIB are mapped to classes. The components in a row are modeled either as attributes (for components of simple types such as *INTEGER*) or as relations among objects (for components representing indexes in other tables).

The derivation of a class diagram takes two steps. In the first step, the MIB definition and additional information about the system to be managed are inspected. Classes, attributes, and associations are extracted and represented in graphical form. The second step is a repeated refinement using inheritance relations among classes to eliminate common definitions of attributes and relations.

The class diagram for the ATM MIB (including only groups related to the ATM layer is given in Fig. 7). Bandwidth requirements of VCL's and VPL's are given by the users and described in terms of traffic description parameters. In Fig. 7, the class *Virtual Link* has an association to the class *Traffic Description Parameters*. The cardinality of this association is one-to-two because two parameters are required to characterize each traffic flow direction on a virtual link. An instance of class

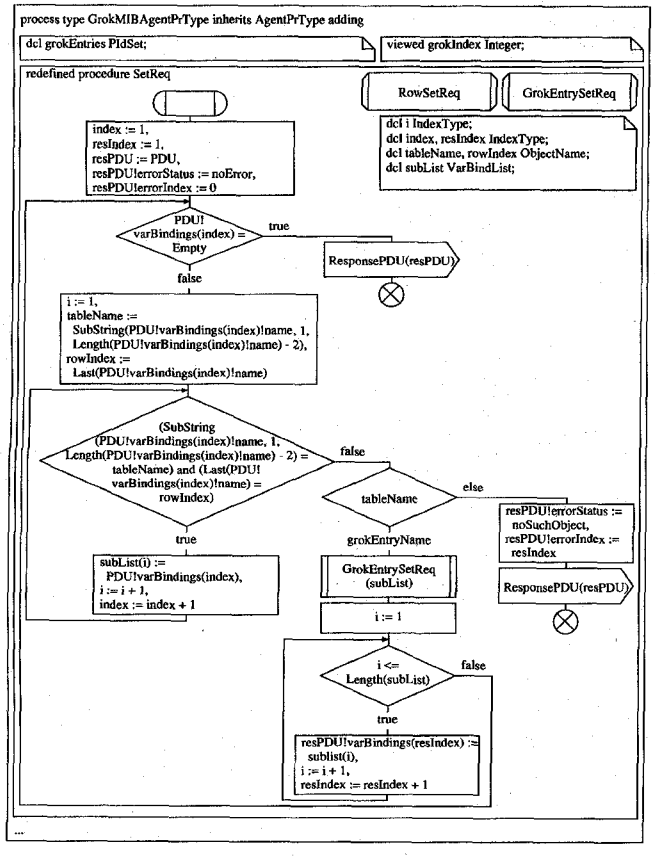


Fig. 11. Processing type diagram *GrokMIBAgentPrType*.

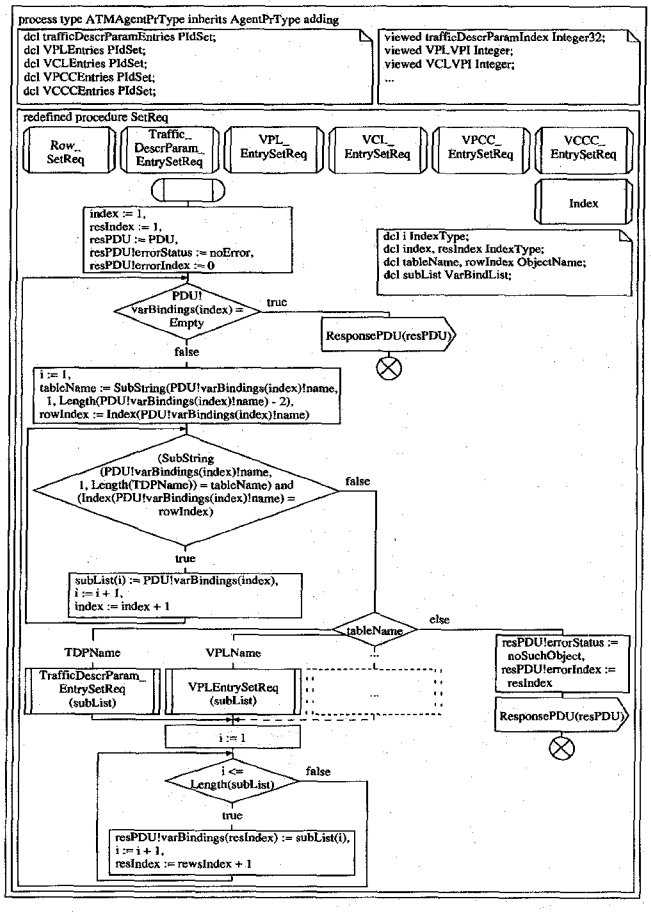


Fig. 12. Process type diagram of *ATMAgentPrType*.

Traffic Description Parameters has two attributes, namely, *Index* and *QoSClass*. The attribute *Index* serves to identify the instance whereas the attribute *QoSClass* indicates the quality of service required by the connections.

Class *Traffic Description Parameters* is defined as a tabular object in the ASN.1 definition of the ATM MIB [1]. One of the fields in the rows of that table is of data type *OBJECT IDENTIFIER*. Its values identify seven possible ATM traffic descriptor types. This is modeled as a class with subclasses and an aggregation relation. That is, an instance of the class *Traffic Description Parameters* contains also an instance of the class *Traffic Descriptor* which has seven subclasses. In Fig. 7, *CLP* denotes cell loss priority and *SCR* denotes sustained cell rate.

C. Derivation of SDL-92 Process Type Definitions from the Class Diagram

The classes that correspond to rows of tabular objects are mapped to SDL-92 process types. These process types inherit the behavior of *RowType* and are specialized to model their corresponding objects. Component classes of a composite class corresponding to a row are mapped to data in the process type of the composite class.

The ATM MIB system diagram is the same as in Fig. 1, as introduced in Section II. The ATM MIB is encapsulated into the SDL-92 block pictured in Fig. 8. Most classes from the class diagram of Fig. 7 are mapped to SDL-92 process types. Class *Traffic Descriptor* is mapped to data in the process type *TrafficDescrParamEntryType* for the composite class *Traffic Description Parameters*. Note that Fig. 8 resembles Fig. 2.

The process type of the agent process is given in Fig. 12. This process inherits from the process type *AgentPrType* given in Fig. 4 and its behavior is a generalization of *GrokMIBAgentPrType* given in Fig. 11.

SNMPv2 protocol operations on the ATM MIB objects can easily be specified based on Section II above. An example on a *SetRequestPDU* with *createAndGo* as the row status is given in Fig. 14 which directly follows from Fig. 13.

IV. MIB Test Case Generation

This section presents the second activity of our methodology, i.e., test suite design from SDL-92 models of managed nodes. The process of obtaining a test suite from a formal specification can be viewed as obtaining the behavior of an entity, called its conformance tester, exhibiting an inverse behavior. That is, inputs (outputs) to (from) the IUT are inverted to become the outputs (inputs) from (to) the tester. We call this process *behavior inversion* [13]. We first develop our method of MIB test case generation and then show an application of it through the design of several test cases.

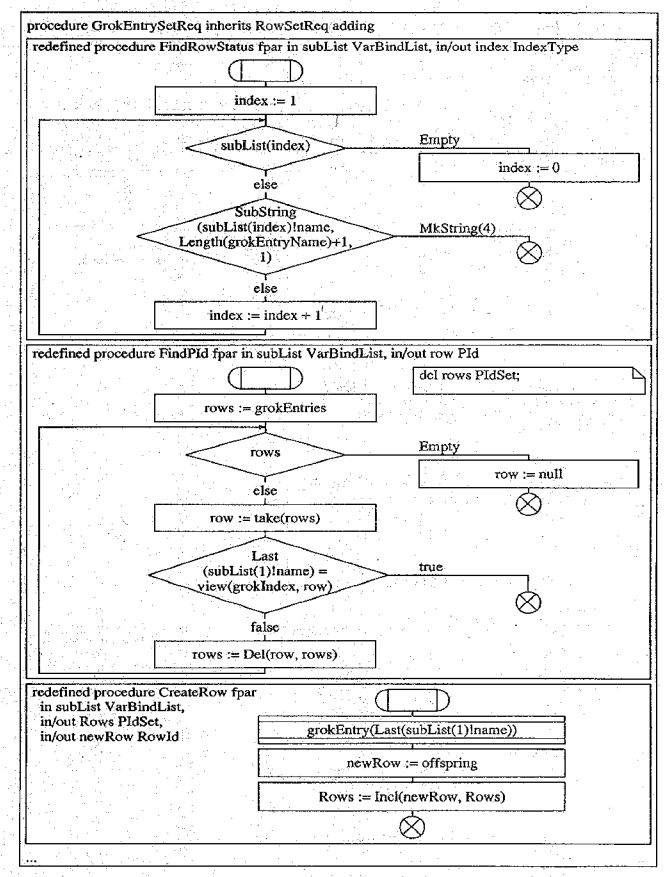


Fig. 13. Specialized procedures for the *grokTable* example.

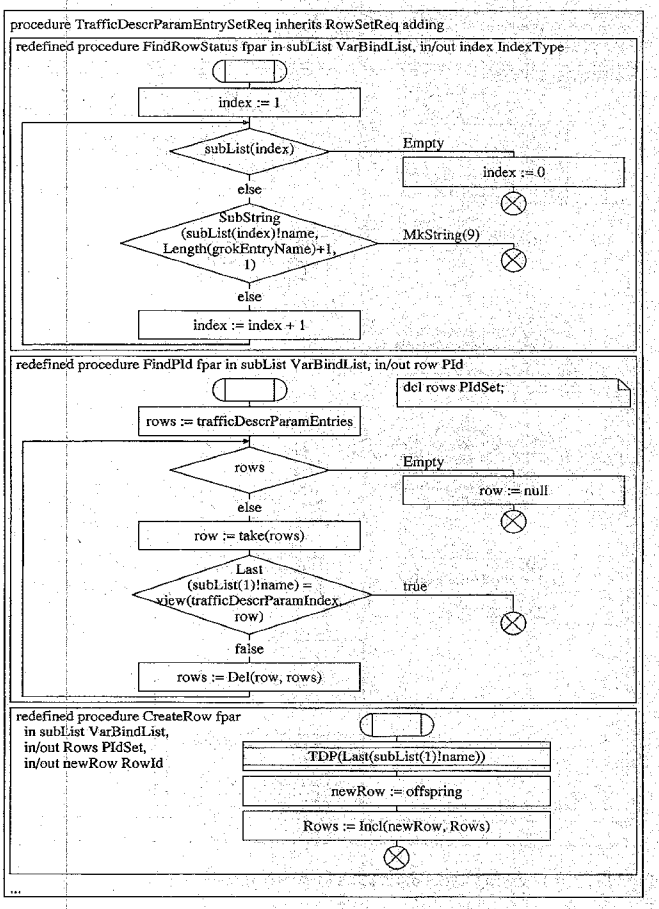


Fig. 14. Specialized procedures for the *Traffic Description Parameter* table of the ATM MIB.

A. Testing Static and Dynamic Control and Data Flow in a Specification

Test cases are selected such that all static and dynamic control and data flow paths in an SDL-92 specification are covered. The model of an SDL-92 specification is an extended finite-state machine. The testing strategy for the control flow aspect is state and transition coverage. Every distinct path, consisting of one or several transitions in the extended finite-state machine, is exercised by test cases. Each test case is assigned a distinct purpose, i.e., test of a given behavior following a certain path. Parameter values of input and output signals of the test case are selected according to the test purpose and such that predicates of the transitions in the test case are satisfied (in order to make the test case executable). The testing strategy for the data flow aspect is variation of parameter values and combination of parameter values.

Process creation, used in the specification to model rows that can be created and destroyed dynamically by the manager, brings another dimension to the control and data flow coverage. Test cases must be designed to exercise creation and destruction of these processes as well as the control and data flow aspects of them.

We use TTCN, ISO's language for specifying protocol conformance test cases [9]. A test case in TTCN is made of events, namely, inputs, outputs, and time-outs. "?" and "!," are the prefix input and output events, respectively, from the point of view of the tester. Apart from a header section, providing identification and general information, a TTCN test case behavior table has five columns: line number (Nr), label, behavior description, constraint reference (Cref), and verdict (V). The label column provides branching labels for *gotos*. The general form of a goto statement is "$->$ ⟨label⟩". TTCN incorporates ASN.1 as a data definition language. The constraints reference column serves to describe particular ASN.1 values to use in input and output events. Verdicts are associated to events terminating branches of the behavior tree. The verdict can be either *Pass* (the purpose of the test is successfully achieved), *Fail* (a nonconforming behavior is detected) or *Inconclusive* (neither *Pass* nor *Fail*).

B. Test Case Generation Examples

We illustrate our test design method on the SDL-92 specification of the *grokTable* agent described in Section II. First *AgentPrType* defined in Fig. 4 is considered. In this process type, the behavior corresponding to different SNMP PDU's coming from the management station is defined. A set of test cases is defined by the analysis of what happens after a *SetRequestPDU* is received. The input of *SetRequestPDU* defines the first external event in the test case. Next, the *SetReq*

TABLE II
TEST CASE 1

Nr	Label	Behavior description	Cref	V
1		!SetRequestPDU	SetRequestPDU1	
2		START transmissionTimer		
3	LB	?ResponsePDU	ResponsePDU1	(PASS)
4		CANCEL transmissionTimer		
5		+POSTAMBLE1		
6		?TrapPDU		
7		−> LB		
8		?TIMEOUT transmissionTimer		INCONC
9		?OTHERWISE		FAIL

TABLE III
SETREQUEST-PDU SEND CONSTRAINT FOR TEST CASE 1

ASN.1 PDU Constraint Declaration
Constraint Name: SetRequestPDU1
PDU Type: SNMPPDU
Derivation Path:
Comments: Send Constraint
Constraint Value
{
request_id 0, error_status noError, error_index 0,
variable_bindings {
{name {grokIPAddress 1}, value address_A},
{name {grokStatus 1}, value createAndGo}}
}

TABLE IV
RESPONSE-PDU RECEIVE CONSTRAINT FOR TEST CASE 1

ASN.1 PDU Constraint Declaration
Constraint Name: ResponsePDU1
PDU Type: SNMPPDU
Derivation Path:
Comments: Receive Constraint
Constraint Value
{
request_id 0, error_status noError, error_index 0,
variable_bindings {
{name {grokIPAddress 1}, value address_A},
{name {grokStatus 1}, value ceateAndGo}}
}

procedure which is defined in Fig. 11 is considered. In this procedure, the *varBindings* parameter of the *SetRequestPDU* is processed. The sublist referring to *grokEntryName* is processed and a corresponding sublist is created and sent in a *ResponsePDU*, thereby defining the second external event in the test case.

We will next elaborate the specification in TTCN of the behavior of the above test case called Test Case 1. In line 1 of Table II, a *SetRequestPDU* is sent by the tester to the IUT. The constraint *SetRequestPDU1* (Table III) defines the values of the PDU fields. Line 3 represents the expected response from the IUT, i.e., a *ResponsePDU*. The constraint of this response is *ResponsePDU1*, instantiated with parameter values (Table IV). A subtree called *POSTAMBLE1* is attached in line 5. Lines 6–9 of Table III define the other events that can occur instead of a *ResponsePDU*.

TABLE V
POSTAMBLE FOR TEST CASE 1

Nr	Label	Behavior description	Cref	V
1		!GetRequestPDU	GetRequestPDU1	
2		START transmissionTimer		
3	LB	?ResponsePDU	ResponsePDU2	(PASS)
4		CANCEL transmissionTimer		
5		?TrapPDU		
6		−> LB		
7		?TIMEOUT transmissionTimer		INCONC
8		?OTHERWISE		FAIL

TABLE VI
GETREQUEST-PDU SEND CONSTRAINT FOR POSTAMBLE 1

ASN.1 PDU Constraint Declaration
Constraint Name: GetRequestPDU1
PDU Type: SNMPPDU
Derivation Path:
Comments: Send Constraint
Constraint Value
{
request_id 1, error_status noError, error_index 0,
variable_bindings {
{name {grokIPAddress 1}, value unSpecified},
{name {grokCount 1}, value unSpecified}}
{name {grokStatus 1}, value unSpecified}}
}

TABLE VII
RESPONSE-PDU RECEIVE CONSTRAINT FOR POSTAMBLE 1

ASN.1 PDU Constraint Declaration
Constraint Name: ResponsePDU2
PDU Type: SNMPPDU
Derivation Path:
Comments: Receive Constraint
Constraint Value
{
request_id 1, error_status noError, error_index 0,
variable_bindings {
{name {grokIPAddress 1}, value address_A},
{name {grokCount 1}, value ?},
{name {grokStatus 1}, value activated}}
}

The purpose of the postamble given in Table V is to check if the set operation has really been performed in the MIB. In line 1 a *GetRequestPDU* is sent to the IUT. Its constraint is *GetRequestPDU1* defined in Table VI. Line 3 is for handling a response to the *GetRequestPDU* from the IUT. If the *ResponsePDU* matches the constraint defined in Table VII, i.e., if the MIB really holds the value previously set, the verdict *PASS* is assigned. Lines 5–8 define the other events that can occur instead of a *ResponsePDU*.

TABLE VIII
SETREQUEST-PDU CONSTRAINT FOR TEST CASE 2

ASN.1 PDU Constraint Declaration
Constraint Name: SetRequestPDU2
PDU Type: SNMPPDU
Derivation Path:
Comments: Send Constraint
Constraint Value
{
request_id 0, error_status noError, error_index 0,
variable_bindings {
{name {grokIPAddress 1}, value address_A},
{name {grokStatus 1}, value createAndGo}}
{name {grokIPAddress 2}, value address_B},
{name {grokStatus 2}, value createAndGo}}
}

TABLE IX
POSTAMBLE FOR TEST CASE 2

Nr	Label	Behavior description	Cref	V
1		!GetRequestPDU	GetRequestPDU1	
2		START transmissionTimer		
3	LB	?ResponsePDU	ResponsePDU2	
4		CANCEL transmissionTimer		
5		!GetRequestPDU	GetRequestPDU2	
6		START transmissionTimer		
7		?ResponsePDU	ResponsePDU3	(PASS)
8		CANCEL transmissionTimer		
9		?TrapPDU		
10		−> LC		
11		?TIMEOUT transmissionTimer		INCONC
12		?OTHERWISE		FAIL
13		?TrapPDU		
14		−> LB		
15		?TIMEOUT transmissionTimer		INCONC
16		?OTHERWISE		Fail

TABLE X
SETREQUEST-PDU CONSTRAINT FOR TEST CASE 3

ASN.1 PDU Constraint Declaration
Constraint Name: SetRequestPDU3
PDU Type: SNMPPDU
Derivation Path:
Comments: Send Constraint
Constraint Value
{
request_id 0, error_status noError, error_index 0,
variable_bindings {
{name {grokIPAddress 1}, value address_A},
{name {grokStatus 1}, value destroy}}
{name {grokIPAddress 2}, value address_B},
{name {grokStatus 2}, value destroy}}
}

Testing of dynamic creation of processes is discussed hereafter. In Fig. 11, *grokEntries* is defined as a set of process identifiers keeping track of dynamically created conceptual row processes. This set is initially empty, i.e., no conceptual rows exist. When a row is created by a *SetRequestPDU*, the procedure *CreateRow* is invoked (see Fig. 6). This procedure (Fig. 13) adds the *PId* of the new process to the *PIdSet grokEntries*. Before a new process is created it is checked by the procedure *FindPId* (Fig. 13) whether the conceptual row has already been created. A test case (Test Case 2) whose behavior description is the same as in Table II is defined to create two *grokTable* rows (for *address_A* and *address_B*). The constraint for *SetRequestPDU* is defined as shown in Table VIII. The postamble part of Test Case 2 requires reading the first and second rows with two consecutive *GetRequestPDU*s and verifying the responses (see Table IX).

Another test case (Test Case 3) is needed to cover the destroy transition in Fig. 3 in the state *activated*. In this test case, the two processes created in Test Case 2 are destroyed. The behavior of Test Case 3 is the same as Test Case 1; however, the data is different again. The constraint of *SetRequestPDU* for Test Case 3 is defined in Table X. The postamble to Test Case 3 requires the verification of the fact that *grokTable* now contains no rows; details of the postamble are omitted.

V. CONCLUSION

This paper addressed a new application field for the formal description technique SDL-92: network management systems, in particular, SNMPv2 of the Internet framework. We have developed reusable parts that specifications of agents with different MIB's have in common and showed a way how to obtain such specifications. For large MIB's, a class diagram representing classes of objects and their relation is developed to exploit common properties. Classes of the class diagram are mapped to SDL-92 process types.

Specifications of the behavior of network management agents in SDL-92 can be used, for instance, as a starting point for the generation of test cases for network management agents. Other possible uses are in agent implementation and in management protocol analysis and simulation.

MIB implementations of managed nodes can be tested for conformance to the MIB and SNMPv2 standards. The test cases can be designed based on an SDL-92 specification of the MIB using specification analysis and behavior inversion. These techniques have been demonstrated on the *grokTable* MIB example.

Syntax and static semantics of our specifications in SDL/PR and TTCN.MP have been checked by means of SDL-TTCN CASE tools [16].

More research needs to be done in improving the test case design method for testing the behavior of an agent managing certain network resources. Theoretical investigations are needed to algorithmically define the various steps of our method.

ISO's network management framework GDMO and its related protocol CMIP present a more powerful network management framework than the Internet framework. The method described in this paper needs to be extended to model GDMO and CMIP.

APPENDIX

MODELING OF PREDEFINED ASN.1 DATA TYPES AND OF DATA TYPES FROM THE SNMPv2-SMI AND SNMPv2-TC MODULES

```
package ASN1;
  syntype ObjIdComponent = Natural
  endsyntype ObjIdComponent;
  newtype OBJECT_IDENTIFIER
    String(ObjIdComponent, Empty)
  endnewtype OBJECT_IDENTIFIER;
  ...

endpackage ASN1;

package SNMPv2_SMI;
  syntype Integer32 = Integer
    constants -2 147 483 648 : 2 147 483 647
  endsyntype Integer32;
  syntype Counter32 = Integer
    constants 0 : 4 294 967 295
  endsyntype Counter32;
  newtype IpAddress Array(IpIndex, Octet)
  endnewtype IpAddress;
  syntype IpIndex = Integer
    constants 1 : 4
  endsyntype IpIndex;
  syntype Octet = Integer
    constants 0 : 255
  endsyntype Octet;
  syntype ObjectName = OBJECT_IDENTIFIER
  endsyntype ObjectName;
  newtype ObjectSyntax
    operators
      IntegerValue :
        ObjectSyntax -> Integer;
      ObjectIdValue :
        ObjectSyntax -> OBJECT_IDENTIFIER;
      IpAddressValue :
        ObjectSyntax -> IpAddress;
      CounterValue :
        ObjectSyntax -> Counter32;
      ...

      ObjectSyntaxInteger :
        Integer -> ObjectSyntax;
      ObjectSyntaxObjectIdValue :
        OBJECT_IDENTIFIER -> ObjectSyntax;
      ObjectSyntaxIpAddress :
        IpAddress -> ObjectSyntax;
      ObjectSyntaxCounter :
        Counter32 -> ObjectSyntax
      ...

    alternative ASN1;
      ObjectSyntax ::=
        CHOICE {
          ...}
    endalternative;
  endnewtype ObjectSyntax;
  ...

endpackage SNMPv2_SMI;

package SNMPv2_TC;
  synonym activate RowStatus = 1;
  synonym notInService RowStatus = 2;
  synonym notReady RowStatus = 3;
  synonym createAndGo RowStatus = 4;
  synonym createAndWait RowStatus = 5;
  synonym destroy RowStatus = 6;
  syntype RowStatus = Integer
    constants activate : destroy
  endsyntype RowStatus;
  ...

endpackage SNMPv2_TC;
```

REFERENCES

[1] M. Ahmed and K. Tesink, *Definitions of Managed Objects for ATM Management Version 8.0 using SMIv2*. IETF-RFC 1695, Aug. 1994, pp. 1–73.

[2] G. V. Bochmann and M. Deslauriers, "Combining ASN.1 support with the LOTOS language," in *Protocol Specification, Testing, and Verification, IX*, E. Brinksma, G. Scollo, and C.A. Vissers, Eds. Amsterdam, The Netherlands: Elsevier, 1989, pp. 175–186.

[3] G. Booch, *Object Oriented Design with Applications*. New York: Benjamin/Cummings, 1994.

[4] J. Case, K. McCloghrie, M. Rose, and S. Waldbusser, "Protocol operations for version 2 of the simple network management protocol (SNMPv2). Network Working Group Request for Comments: 1448," Apr. 1993.

[5] ——, "Structure of management information for version 2 of the simple network management protocol (SNMPv2). Network Working Group Request for Comments: 1442," Apr. 1993.

[6] ——, "Textual conventions for version 2 of the simple network management protocol (SNMPv2). Network Working Group Request for Comments: 1443," Apr. 1993.

[7] J. Fischer and R. Schröder, "Combined specification using SDL and ASN.1," in *6th SDL Forum*, O. Færgemand and A. Sarma, Eds. Amsterdam, The Netherlands: Elsevier, 1993, pp. 293–304.

[8] *Information technology—Open Systems Interconnection—Specification of Abstract Syntax Notation One (ASN.1)*, International Standard ISO/IEC 8824, 1990.

[9] *Information technology—Open Systems Interconnection—Conformance testing methodology and framework*, International Standard ISO/IEC 9646, 1991.

[10] *Specification and description language SDL '92*, ITU-T Recommendation Z.100, 1992.

[11] *SDL combined with ASN.1*, ITU-T Recommendation Z.105, 1994.

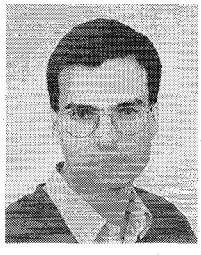
[12] M. T. Rose, *The Simple Book: An Introduction to Internet Management*, 2nd ed. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[13] B. Sarikaya, *Principles of Protocol Engineering and Conformance Testing*. New York: Simon & Schuster, Sept. 1993.

[14] W. Stallings, *Data and Computer Communications*, 4th ed. New York: Macmillan, 1994.

[15] D. Steedman. *Abstract Syntax Notation One (ASN.1): The Tutorial and Reference*. London, England: Technology Appraisals Ltd., 1993.

[16] Telelogic, *Getting Started with SDT 3.0*. Malmö, Sweden: Telelogic AB, Feb. 1995.



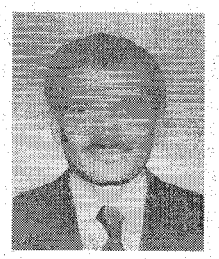
Olaf Henniger received the Diplom-Ingenieur degree in automation engineering from the Otto-von-Guericke University Magdeburg, Germany, in 1991.

Until 1994, he was a Research Fellow at the Otto-von-Guericke University Magdeburg, Germany. Currently, he is a Research Staff Member at GMD, the German National Research Center for Information Technology. He continues to work on his doctoral thesis dealing with testing of communicating systems.

Michel Barbeau (S'89–M'91) received the B.Sc. degree from University of Sherbrooke, Canada, in 1985, and the M.Sc. and Ph.D. degrees in computer science from the University of Montreal, Canada, in 1987 and 1991, respectively.

Since 1991, he has been Professor of Computer Science at the Department of Mathematics and Computer Science at the University of Sherbrooke, Canada. He was also a Visiting Member of the Computer Communications Lab of The University of Aizu, Japan. His current research interests include telecommunications and real-time systems software engineering.

Dr. Barbeau was awarded the Gold Medal of the Governor General of Canada in 1991; this is awarded to students who graduate with highest standing, for study at the Master and Doctorate level, in their institution.



Behçet Sarikaya (SM'91) received the B.S.E.E. and the M.Sc. degrees from the Middle East Technical University (METU), Ankara, Turkey, in 1973 and 1976, respectively, and the Ph.D. degree from McGill University, Montréal, Québec, Canada in 1984, all in computer science.

Presently, he works in the University of Aizu in Aizu-Wakamatsu, Fukushima, Japan, as a Professor. Previously, he has held full-time faculty positions in the Universities of Sherbrooke, Concordia, and Montréal in Canada and Bilkent in Turkey. His current research interests lie in multimedia networking, personal communication systems, and integrated network management. He has been Co-Chairman of IFIP PSTV VI held in Montréal in 1986 and Program Chair of MmNet'95 held in Aizu-Wakamatsu in 1995. Since 1987, he has continuously served in the program committees of several international conferences. He is the author of the book *Principles of Protocol Engineering and Conformance Testing* (Englewood Cliffs, NJ: Prentice-Hall, 1993).