MIB Based Functional Model for Test Generation

Jorge Mamede[1,2], Eurico Carrapatoso[1,3], and Manuel Ricardo[1,3]

[1] INESC Porto - Instituto de Engenharia de Sistemas e Computadores do Porto
Campus da FEUP, Rua Dr. Roberto Frias, 4200-378 Porto, Portugal
{jmamede,emc,mricardo}@inescporto.pt
http://www.inescporto.pt

[2] I.S.E.P. Instituto Superior de Engenharia do Porto
Rua de S. Tomé, Porto, Portugal
http://www.isep.ipp.pt

[3] F.E.U.P. Faculdade de Engenharia da Universidade do Porto,
Campus da FEUP, Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
http://www.fe.up.pt

Abstract. This paper proposes a method for generating a functional model of complex IETF network elements in short times. For that purpose, the Management Information Bases (MIBs) are used, enriched with behaviour and described in a formal language that can be used for automatic test derivation. The method is demonstrated by applying it to an MPLS router.

1 Introduction

IETF protocols are playing a central role in the telecommunications world since, as known, networks are increasingly becoming IP oriented. In order to make the overall network manageable, IETF uses the Simple Network Management Protocol (SNMP) to provide interoperability between the network managers and network elements of different vendors. This protocol forces the network elements to store information about configuration, fault, and performance in standard databases, known as Management Information Bases (MIBs).

On the other hand, the adoption of IETF protocol based equipment by the public telecom operators is pushing the traditional practices testing to these equipments. This situation brings two problems: 1) IETF protocol specifications are less behaviour oriented than ETSI or ITU specifications and, for instance, no SDL models are usually available; 2) test suites, when available, are not so complete as in ETSI or ITU.

This paper reports a study being carried out in INESC Porto to reducing these problems, which is related with previous works [1]. We assume that, ideally, tests should be generated automatically. For that, we need a rigorous model of the system to test. Then, the problem addressed in this paper is the following - how can we develop in short time a functional model that can be used for test derivation?

The solution we propose, and partially present in this paper, answers: from the MIB specifications. These databases reflect the essential aspects related to equipment or protocol configuration, performance and fault detection. If this structured database is placed in the center of a model, and some additional code is developed that exposes the inputs and outputs essential for testing, a functional model can be derived in short times. The problem is similar to that of, for instance, having the variables of a C++ class (the MIB, in our approach) and just having to develop the methods required to manipulate them. More than giving us the framework required for thinking about behaviour, these variables let us also estimate how far we are from a final functional model. Besides working on the concept, we are also applying it to a real case - an MPLS router.

The work is reported in 7 sections. Section 2 introduces the essential network management concepts, by emphasizing the meaning of the information contained in a MIB. Section 3 describes some automatic test generation methods and tools available. We intend to show that, if a model is available, the tools developed by the testing community can be used. Section 4 exposes, in abstract, our thesis about test generation from MIBs. Section 5, introduces the MPLS technology, and describes the MPLS router main functions. The router MIBs are also presented in detail. Section 6 shows the functional model of the MPLS router defined from the MIB information. We used the Promela language for expressing it. Section 7 concludes this paper.

2 SNMP MIB - The network element management information base

In order to understand the management of an IP network element, a set of concepts must be first introduced - manager, agent, management protocol, and management information base.

The manager, or management station, is the entity responsible for configuring and monitoring the network elements. It can receive alarms from the network elements.

The agent is the functional entity located at the network element that communicates with the manager. The agent is responsible for storing management information related to the behaviour of the network element and its components. When the manager requests the agent to provide information or to carry out an action, the agent behaves accordingly. The agent may also, asynchronously, notify the manager about abnormal situations by sending *trap* messages.

The management protocol is the language the manager and the agent use to communicate. The well known Simple Network Management Protocol (*SNMP*), is the protocol commonly used in the IETF world. This protocol defines commands and traps. The manager issues a command for setting / retrieving information at / from the agent. The agent issues a trap to inform the manager about some unexpected occurrence at the network element.

The Management Information Base, known as MIB, is the structured collection of information representing the network element being managed [2] [3].

Each network element component, also called resource, is typically represented by one object. An object representing a complex resource may, in turn, aggregate other objects representing parts of the resource. As a whole, objects are specified so that the manager may have information about the configuration and the operational status of the network element.

In order to serve the needs of the manager and to enable interoperability, a common scheme for representing MIBs is used [2] . MIB objects are specified in Structure of Management Information (SMI), that is a subset of the Abstract Syntax Notation One (*ASN.1*). In SMI, each object is referenced via an object identifier - the (*OID*). Multiple instances of the same resource are represented by multiple objects of the same type, defined as table entries, and identified by the index of their position in the table. During the network element operation, the information relevant for the MIB is collected by the agent and organized according to SMI rules.

MIBs are developed by IETF working groups and then standardized [4]. However, since the object definition can depend on implementation aspects, vendors may have to develop MIB extensions.

3 Automatic test generation

Testing software, protocols, or equipment can be broadly defined as the activity of evaluating if a given implementation behaves as expected. This activity is usually time consuming and error prone. In order to overcome these problems, researchers have worked for decades on the automatic generation of tests from models describing the expected behaviour of the object under test. Automatic test generation is simplified when the object behaviour model is formal. Successful examples of automatic test generation can be found in [5], [6], [7], [8], and [9].

A model representing the object under test can be a Finite State Machine (*FSM*) obtained, from instance, from an Extended FSM (*EFSM*), or a set of communicating EFSM (*CEFSM*). Input Output Finite State Machines (IOFSM) are a specialization of the latter, where actions are further classified as input or output. From this model, a reachability graph can be obtained, where nodes represent the model states and branches represent transitions between states. Using animated graph exploration techniques, reachable states and transitions can be found from some initial state [10]. This reachability graph can be used as the departing point for automatic test generation, since it can describe all the execution sequences valid for the object. Several approaches exist.

Some methods refer explicitly a Labelled Transition System (LTS), a large automaton representing the behaviour of the object. The LTS consists of states, representing the system states, and the transitions among states are labelled with interactions between the system and its environment. In some sense, the labelled transition system represents also the system reachability graph. The states of a model can be reached on-the-fly using a depth-first search mode. During the search, a test derivation algorithm is applied recursively to a set of states [11] and, based on the reachable states, tests are generated. A methodology where test cases are assigned pass or fail verdicts, is also presented in [7]. A pass verdict means that the implementation under test exhibited a trace that can also be found in the reachability graph. Fail is used to classify an implementation showing an execution not previewed in the model, or that has stopped executing and it should not. In order to avoid the state space explosion problem, the search is limited in depth. Some test generation tools can, in some way, be described under the above method. TVEDA [12], developed by France Telecom CNET, generates TTCN tests from SDL specifications, and uses heuristics of experienced test case developers to generate high quality tests. TGV [6] [13] generates TTCN tests from LOTOS and SDL specifications, and allows test purposes to be specified by means of automata. TorX [14] generates tests from LOTOS and Promela specifications.

There are also automatic test generation methods that consist of defining test purposes and using them to drive the model, from which test cases are constructed. These test purposes are described in MSCs, or can be obtained by identifying paths in the reachability graph. Test cases are generated based on that and assign verdicts pass, inconclusive, or fail. The pass verdict means that the behaviour exhibited by the implementation under test is described in the model. The inconclusive verdict is assigned when, although correct from the point of view of the model, the execution is not conclusive with respect to the test purpose. The fail verdict is assigned when the system behavior is forbidden by the model. The SaMsTaG [15], [16] uses this approach. Further enhancements were introduced in AUTOLINK [17], [5]. There, the state space exploration based on the supertrace algorithm [10] was added. A tool using a similar approach is the TestComposer [18], [5] that, besides using the test purpose to drive the search, uses also some user defined scripts to improve performance. TestComposer also implements breadth-first search algorithm and combines it with depth-first to explore small zones in the neighborhood of a particular state [19].

Other approaches exist for automatic test case generation. Ammann *et al* [20], [8] present a methodology based on syntactic mutant analysis, where the program source code is used instead of semantic aspects associated to input-output. It defines mutant operators that are applied to the specification or temporal logic formulas to derive mutations as test requirements. Then, it drives a model checker to generate counterexamples that are used for test case definition [20], [8]. Callahan *et al* [21] identifies a computation tree consisting of a set of possible execution paths of the model. Their method consists of defining linear temporal logic properties and to check the computation tree for them. The counterexamples are produced whenever a path is detected that violates assertions or satisfies a negative temporal requirement [21]. Gargantini *et al* describe a requirement specification based method for constructing a suite of test sequences. Sets of temporal properties are defined that force a model checker to generate the desired counterexamples [9].

4 Test Generation from MIB based specification

We assume that, for generating tests, a functional model of the system under test is required. This functional model, we say, can be developed based mainly on the MIB information, the protocol specification, and some human knowledge. Fig. 1 illustrates the process.

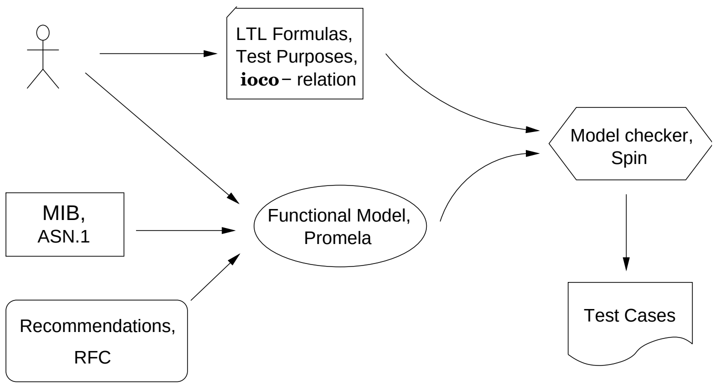

Fig. 1. Test generation methodology

The MIB contains information related to the network element resources. Each resource is described by a set of objects having information about configuration, faults, or performance. By combining this information with the protocol description, it will be possible to develop a functional model that can be used for generating tests. Human experience, here, can also be important, since the relations between MIB objects need to be identified. The resulting model should, in our approach, be as close as possible to the original MIB objects.

This model is represented in Promela (Process Meta Language), so that it can be verified using the Spin [10] techniques. Besides, it is extraordinary important that the Promela implementation is optimised to minimise the state space [10] [22]. Human experience may also play here a relevant role. The functional model must then be verified to detect unreachable code and system locks.

The automatic generation of the test cases can be achieved using methods similar to those described in section 3. Different approaches can be used, such as, definition of test purposes, use of Linear Temporal Logic (*LTL*) formulas, and ioco-relation oriented test suite.

5 MPLS router

Multi-Protocol Label Switching (MPLS) is a network technology under standardisation by the IETF [23]. MPLS emerged to converge the internet routing protocols and the connection-oriented forwarding techniques [24]. It enables the establishment of a virtual connection, the *path*, between two MPLS edge routers for exchanging IP traffic.

The purpose of MPLS is to speed up packet forwarding in routers. Therefore, MPLS packets use fixed-length *labels*, that enable fast address lookups [23]. Traffic engineering is also a powerful argument of MPLS. Its ability to forward IP traffic over arbitrary non-shortest paths and the capacity to emulate high-speed tunnels between non MPLS domains, makes it interesting to service providers, when trying to provide IP VPN services [24] [25]. MPLS also supports classes of services. This feature enables the provisioning of differentiated services, some of them capable of satisfying guaranteed QoS requirements [25] [26].

5.1 Description

A router supporting MPLS is called Label Switch Router (*LSR*). A group of communicating LSRs makes an MPLS network, also known as MPLS domain. LSRs located at the edge of the MPLS domain are referred to MPLS edge routers. With respect to the traffic they serve, they are further classified as ingress or egress routers. Traffic flows in an MPLS domain are associated to traffic classes, named Forwarding Equivalence Classes (*FEC*). To each FEC is associated a label.

When an IP packet enters an MPLS domain via an ingress router, this router analyses the packet header, determines its FEC, and appends one label to the packet. This label will be used inside the MPLS domain to forward the packet between routers. While a traditional IP router forwards a packet based on its IP destination address, an MPLS router fowards a packet based on the label above mentioned. This label is used inside the MPLS router as the index of a table, which specifies a new label and the next hop in the path. The path followed by packets belonging to the same Forwarding Equivalence Class is named Label Switched Path (*LSP*). An LSP starts at an ingress router and terminates at an egress router. Each LSP is made of one or more segments. A segment connects adjacent routers and is identified by a label and a router interface. Only the packets with the segment label are allowed to travel through the segment interface. The MPLS router receives a packet associated to an input segment and forwards it through one or more output segments. So, the packet label is changed in every MPLS router. Edge routers are different. Ingress routers only have output segments while egress routers only have input segments. Within the MPLS domain several subdomains may exist. Subdomains are associated to different levels of labels. When a packet enters a subdomain, a new label is added to it in a stack like style. The generalisation of this feature in the MPLS network enables hierarchical forwarding or *tunneling*.

At the end of and LSP, the egress router is responsible for removing the label from the packet. If the egress router is located in the edge of the MPLS domain (no more labels in the stack), besides removing the label, it also analises the IP packet header and, according to its forwarding table, sends the packet to the next IP hop.

5.2 MIBs

The MPLS management information is divided into several MIBs, shown in Fig. 2, where each MIB is associated to a particular management purpose. They are described as follows:

- **MPLS-TC-MIB**. Textual Conventions MIB, where the textual definitions and objects entities common to other MPLS MIBs are defined [27];
- **MPLS-LSR-MIB**. Tree of objects used to model and manage the behaviour of the MPLS router. This is the heart of the management architecture for MPLS [27] [28];
- **MPLS-LDP-MIB**. Used when the routers exchange label information using the MPLS Label Distribution Protocol (LDP); its objects model and manage the MPLS Label Distribution Protocol;
- **MPLS-TE-MIB**. To configure and control the traffic engineered tunnels; it contains information related with the tunnels, identifies the LSRs that support them, and keeps statistical information;
- **MPLS-FTN-MIB**. Defines objects to coordinate mappings between FECs and Next Hop Label Forwarding Entries (NHLFEs).

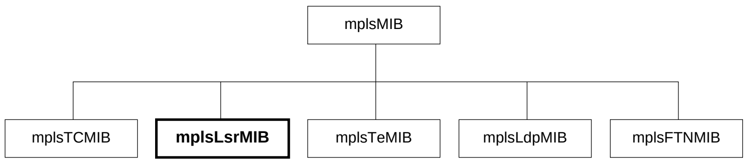

Fig. 2. MPLS MIB

In this paper we address mainly the MPLS-LSR-MIB [28]. It contains the information that better describes the behaviour of an MPLS router. The tree structure of the MPLS-LSR-MIB objects is shown in Fig. 3. The LSR MIB is organised in three structures: the *mplsLsrObjects*, the *mplsLsrNotifications*, and the *mplsLsrConformance*. The *mplsLsrObjects* define the object types and tables used to store the LSR configuration and its working status. The *mplsLsrNotifications* defines the occurrences that result in traps to the manager. The *mplsLsrConformance* defines the mandatory objects in a compliant LSR MIB. From these three main components, only the first two address device dynamics. The last determines its static capabilities.

The *mplsLsrObjects* MIB has objects (1) to enable the MPLS protocol in the MPLS-capable interfaces (*mplsInterfaceConfTable*), (2) to monitor interface performance parameters (*mplsInterfacePerfTable*), (3) to configure the incoming LSP segments (*mplsInSegmentTable*), (4) to define the input segment performance parameters (*mplsInSegmentPerfTable*), (5) for indexing the mplsOutSegmentTable (*mplsOutSegmentIndexNext*), (6) to configure the LSR output seg-

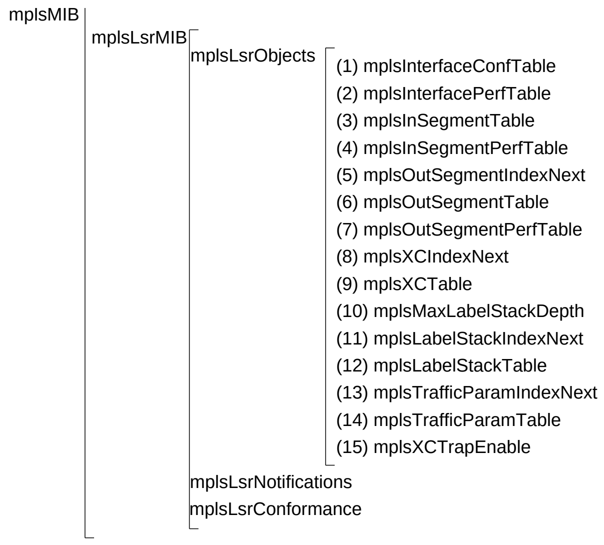
mplsMIB
mplsLsrMIB
mplsLsrObjects
(1) mplsInterfaceConfTable
(2) mplsInterfacePerfTable
(3) mplsInSegmentTable
(4) mplsInSegmentPerfTable
(5) mplsOutSegmentIndexNext
(6) mplsOutSegmentTable
(7) mplsOutSegmentPerfTable
(8) mplsXCIndexNext
(9) mplsXCTable
(10) mplsMaxLabelStackDepth
(11) mplsLabelStackIndexNext
(12) mplsLabelStackTable
(13) mplsTrafficParamIndexNext
(14) mplsTrafficParamTable
(15) mplsXCTrapEnable
mplsLsrNotifications
mplsLsrConformance

Fig. 3. LSR MIBs

ments (*mplsOutSegmentTable*), (7) to define the output segment performance parameters (*mplsOutSegmentPerfTable*), (8) for indexing the mplsXCTable (*mplsXCIndexNext*), (9) to set the cross connections (*mplsXCTable*) between input and output segments, (10) to define the maximum depth of the label stack (*mplsMaxLabelStackDepth*), (11) for indexing the mplsLabelStackTable (*mplsLabelStackIndexNext*), (12) to specify the label stack to be pushed onto a packet (*mplsLabelStackTable*), (13) for indexing the mplsTrafficParamTable (*mplsTrafficParamIndexNext*), (14) to define the LSP traffic parameters (*mplsTrafficParamTable*), and (15) to enable the traps (*mplsXCTrapEnable*). The *mplsLsrNotifications* MIB has one object, the *mplsLsrNotifyPrefix*, not represented in the figure, where two traps are defined (*mplsXCDown* and *mplsXCUp*) to report a change in LSP status.

Most of these objects consist of tables. Each table entry is itself a structure of objects, typically the leaves the MIB, that characterise router facilities.

6 MPLS LSR functional model

This section describes a functional model of the MPLS LSR router suitable for test generation. The model is defined based on communicating extended finite state machines representing the router, its network environment, and the network manager.

6.1 Relevant MIB objects

The model was described in Promela [10], a language used mainly to model distributed and concurrent reactive systems.

Promela consists mainly of three types of constructs: processes, variables, and message channels. A process is a state machine described in C like syntax and, among other features, it enables the description of non-deterministic choices. A channel is the mechanism available for processes exchanging messages. Using them, communications between processes can be made synchronous or asynchronous. The variables are of C types and, usually, are manipulated by processes. They can can also be organised as arrays, structures, or channels.

Modeling the MIB data structure in Promela assumes a particular relevance. Our thesis, as mentioned before, forces the functional model to **strongly** depend on it. Therefore, ASN.1 types shall be consistently represented by Promela types, according to their role. Simple and structured types are represented directly using the Promela predefined types and the *typedef* construct. Tables are represented by arrays of structures. However, in order to improve the model behaviour, two fields, *next* and *previous*, were added to each structure to allow linked list operations on tables. In this way, operations such as insertion, remotion, and search can be easily represented.

The *mplsInterfaceConfTable* object (1) shown in Fig. 3 consists of an array of MplsInterfaceConfEntry objects, whose composition is shown bellow. Each table entry is associated to a physical interface.

```
MplsInterfaceConfEntry ::= SEQUENCE {
  mplsInterfaceConfIndex             InterfaceIndexOrZero,
  mplsInterfaceLabelMinIn            MplsLabel,
  mplsInterfaceLabelMaxIn            MplsLabel,
  mplsInterfaceLabelMinOut           MplsLabel,
  mplsInterfaceLabelMaxOut           MplsLabel,
  mplsInterfaceTotalBandwidth        MplsBitRate,
  mplsInterfaceAvailableBandwidth    MplsBitRate,
  mplsInterfaceLabelParticipationType BITS,
  mplsInterfaceConfStorageType       StorageType
}
```

Each object represented characterises one behaviour aspect of the MPLS interface. We have concentrated on 3 of them, whose description is quoted from [28]:

- **mplsInterfaceLabelMaxIn** This is the maximum value of an MPLS label that this LSR is willing to receive on this interface.
- **mplsInterfaceLabelMinOut** This is the minimum value of an MPLS label that this LSR is willing to send on this interface.
- **mplsInterfaceTotalBandwidth** This value indicates the total amount of usable bandwidth on this interface and is specified in kilobits per second (Kbps). This variable is not applicable when applied to the interface with index 0.

These object descriptions can be used to define the interface behaviour. Some of them are used to filter arriving packets; other objects can be considered or not according to their relevance in the behaviour we intend to address. The *mplsInterfaceTotalBandwidth* object, for instance, reports the value of a physical characteristic that is not relevant in a model oriented to functional test generation. However, it can be important for the generation of test cases addressing performance or quality of service.

Other structures exist on the MPLS MIB containing objects that are relevant for functional modeling. For simplicity, we will represent only the structures having relevant objects.

The *mplsInSegmentTable* object (3) shown in Fig. 3 consists of an array of MplsInSegmentEntry objects, whose composition is shown bellow. Each input segment of the MPLS router is associated to an *MplsInSegmentEntry* object.

```
MplsInSegmentEntry ::= SEQUENCE {
   mplsInSegmentIfIndex            InterfaceIndexOrZero,   -- Interface number
   mplsInSegmentLabel              MplsLabel,              -- Segment Label
         ...
   mplsInSegmentXCIndex            Integer32,              -- index of mplsXCTable entry
                                                           -- that accepts that segment
         ...
}
```

The *mplsOutSegmentTable* object (6) shown in Fig. 3 consists of an array of MplsOutSegmentEntry objects, whose composition is shown bellow. Each output segment of the MPLS router is associated to an *MplsOutSegmentEntry* object. When a new element is added to this table, the *mplsOutSegmentIndexNext* (5) is used to define its position on the table.

```
MplsOutSegmentEntry ::= SEQUENCE {
  mplsOutSegmentIndex              Integer32,       -- Segment index
  mplsOutSegmentIfIndex            InterfaceIndex,  -- Interface number
        ...
  mplsOutSegmentXCIndex            Integer32,       -- index of mplsXCTable entry
                                                    -- that accepts that segment
        ...
 }
```

For multicast purposes, the same packet can be forwarded via multiple output segments. In this case, all these output segments have the same *mplsOutSegmentXCIndex* value.

The *mplsXCTable* object (9) shown in Fig. 3 consists of an array of MplsXCEntry objects, whose composition is shown bellow. The *mplsXCTable* entries connect an input segment to an output segment. This table contains information for identifying the LSPs connected through this LSR. The addition of a new LSP consists of inserting a new entry in mplsXCTable in the position indexed by the *mplsXCIndexNext*. An MplsXCEntry is defined as follows:

```
MplsXCEntry ::= SEQUENCE {
      mplsXCIndex                Integer32,    -- XConnect index
      mplsXCLspId                MplsLSPID,    -- LSP identifier
         ...
   }
```

6.2 Functional Model in Promela

The functional model we have built consists of three main blocks, including the block required to close the system and enable its simulation. Blocks are connected through synchronous message channels. Fig. 4 shows the model architecture.

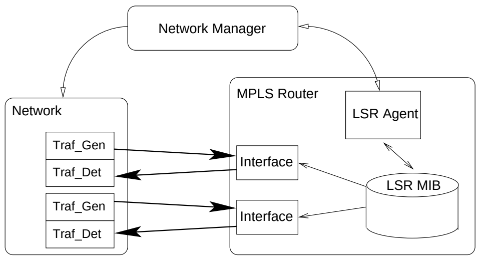

Fig. 4. Model architecture

Network Manager The manager is responsible for configuring the router and its environment. Configuring the router consists in instantiating network interfaces and setting the MIB values for the establishment of LSPs.

Network The network block aims at closing the router conveniently, so that the system can be simulated and verified. It is composed mainly of packet generators and detectors. A generator/detector pair is associated to each router interface.

The MPLS packet consists of header and data. Since the header contains one or more label stack entries, it was implemented by a channel of *LabelStackEntry*. The Promela definitions of each *LabelStackEntry* and the MPLS packet follow:

```
typedef LabelStackEntry {
        int            Label;    /* MPLS label */
        byte           Exp;      /* Exprimental bits, not used in the model */
        bit            S;        /* Stack ctrl bits: S=1 ->last stack entry */
        byte           TTL;      /* Time To Live */
}
typedef MPLSPacket{
        chan   LabelStack = [MAX_LABEL_STACK_SIZE] of { LabelStackEntry }
        chan   Data       = [MAX_DATA_SIZE]        of { byte }
}
```

LSR The router block consists of the MIB data structure, the MPLS agent, and the router interfaces. The MIB contains the information stored in the router, and it is used by the functional model. The MPLS agent serves the manager requests. Only functions related to interface instantiation were considered. The key functional entity of the router is the interface. When the interface receives a packet, this packet is parsed and evaluated. If valid, the interface switches the packet label, and forwards the packet to the correspondant output network interface. Most of this behavior can be inferred from the MIB. The state machine representing the interface is shown in Fig. 5. It refers to the MIB objects presented in the previous section that are relevant for packet forwarding. In order to simplify the reading of the diagram, the following macros were defined:

```
#define __ValidInLabel__ \
         ( (toplsentry.Label > mplsInterfaceConfEntry[ifnum].mplsInterfaceLabelMinIn) && \
           (toplsentry.Label < mplsInterfaceConfEntry[ifnum].mplsInterfaceLabelMaxIn) )
```

```
              /* tests the Label of first label stack entry at "ifnum" interface */
#define __InvalidInLabel__ ( ! __ValidInLabel__ )

#define __ValidOutLabel__  \
        ( (mplsInterfaceConfEntry[mplsOutSegmentEntry[osidx].mplsOutSegmentIfIndex]. \
                                    mplsInterfaceLabelMinOut <= newlse.Label) && \
           ( mplsInterfaceConfEntry[mplsOutSegmentEntry[osidx].mplsOutSegmentIfIndex]. \
                                    mplsInterfaceLabelMaxOut >= newlse.Label) )
            /* tests the stacks first Label of the packet to be transmitted through */
            /* the interface associated with output segment with the "osidx" index  */
#define __InvalidOutLabel__ ( ! __ValidOutLabel__ )

#define __ValidInSegXConn__     ( mplsInSegmentEntry[isidx].mplsInSegmentXCIndex != 0 )
            /*tests if the input segment with the "isidx" has an entry in mplsXCtable*/
#define __InvalidInSegXConn__   ( ! __ValidInSegXConn__ )

#define __ValidInSegment__      ( isidx != IS_NULL )
             /* Check if the input segment index is valid */
#define __InvalidInSegment__   ( ! __ValidInSegment__ )

#define __ValidOutSegment__     ( osidxstk.stksize != 0 )
#define __InvalidOutSegment__   ( ! __ValidOutSegment__ )
```

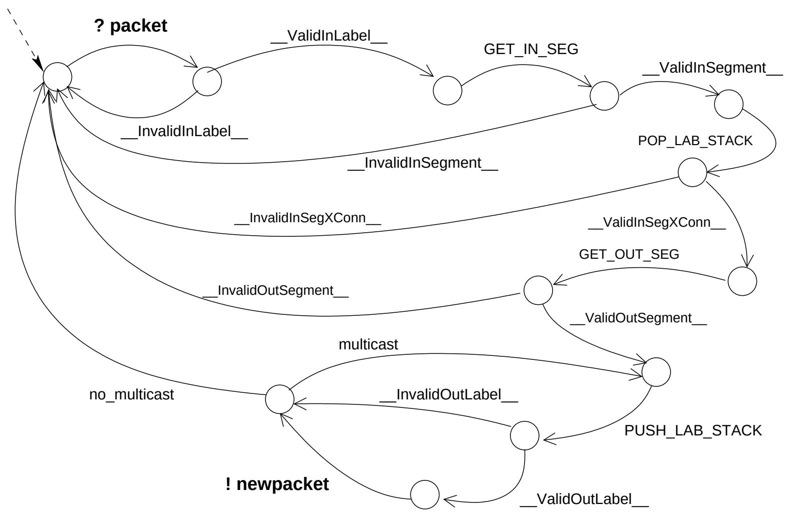

Fig. 5. Interface state machine

One packet arriving at the interface identified by *mplsInterfaceConfIndex* is received. It is removed from the channel where the associated network generator places it. The packet validation procedure is then initiated. In our model this validation is initiated by verifying whether the packet label fits between the maximum and minimum values imposed by the input interface. If this is true, *__ValidInLabel__* holds; otherwise *__InvalidInLabel__* holds and the packet is dropped. Then, a search is initiated in the *mplsInSegmentTable* to check if there exists an entry for the input segment. The search is implemented in Promela by the inline macro *GET_IN_SEG* that is executed as an atomic transition [29] [22]. As result of this search, the index of the entry matching the label is obtained. In this case, then the *__ValidInSegment__* holds. The macro *GET_IN_SEG* is presented below:

```
inline GET_IN_SEG(ret,ifindex,pktlabel)
{                                                          /* ret - return inseg index  */
        Integer32 idx   = mplsInSegmentTable_headidx;      /* ifindex - interface index */
                                                           /* pktlabel - packet label   */
        ret = IS_NULL;
        do
        :: ( ( idx != IS_NULL) &&
             ( mplsInSegmentEntry[idx].mplsInSegmentIfIndex == ifindex ) &&
             ( mplsInSegmentEntry[idx].mplsInSegmentLabel == pktlabel  ) ) ->
                {
                   ret = idx;
                   break;
                }

        :: ( ( idx != IS_NULL) &&
             ( ( mplsInSegmentEntry[idx].mplsInSegmentIfIndex != ifindex ) ||
               ( mplsInSegmentEntry[idx].mplsInSegmentLabel != pktlabel  ) ) ) ->
                {
                   idx = mplsInSegmentEntry[idx].next;
                }
        :: (idx == IS_NULL) ->
                        break;
        od;
}
```

If no entry exists, *__InvalidInSegment__* holds and the packet is dropped. Assuming that there is a valid input segment, the next step consists of processing the packet label stack. This task is assured by the inline *POP_LAB_STACK* macro that removes one or more *LabelStackEntry*s from the *LabelStack* channel. We made also this macro atomic, and represent it as a transition in the state diagram.

```
inline POP_LAB_STACK (toplsentry,isidx)
{                                            /* toplsentry - top label stack entry */
     if                                      /* isidx      - input segment index   */
     :: ( toplsentry.S == 0 ) ->
        {
           npop = mplsInSegmentEntry[isidx].mplsInSegmentNPop - 1;

           do
           :: (npop != 0) ->
               {
                    packet.LabelStack ? lsentry;
                    npop = npop - 1;
                    if
                    :: (lsentry.S == 1) ->
                            break;
                    fi;
               }
           :: (npop == 0) ->
                   break;
           od;
          }
     :: ( toplsentry.S == 1) ->
               skip;
     fi;
}
```

The interface proceeds by checking if the MplsXCEntry object indexed by the input segment exists. *__ValidInSegXConn__* holds if this index is valid. The *MplsXCEntry* identifies the LSP, and relates the input segment to the output segment or segments, in case of multicast. The selection of the output segments is done by another inline macro, the *GET_OUT_SEG*. This macro searches the *mplsOutSegmentTable* for entries indexing the referred *mplsXCEntry*. *__ValidOutSegment__* holds when entries are detected. The indexes of the detected entries are then pushed into a return stack (*osidxstack* below).

```
inline GET_OUT_SEG(osidxstack, xcindex)
{                                         /* osidxstack - stack of detected indexes */
                                          /* xcindex - the index of the mplsXCEntry */
    Integer32 idx = mplsOutSegmentTable_headidx;

      do
      :: ( ( idx != OS_NULL ) &&
           ( mplsOutSegmentEntry[idx].mplsOutSegmentXCIndex == xcindex )) ->
              {
                  _push_int_(osidxstack,mplsOutSegmentEntry[idx].mplsOutSegmentIndex);
                  idx = mplsOutSegmentEntry[idx].next;
              }
      :: ( ( idx != OS_NULL ) &&
           ( mplsOutSegmentEntry[idx].mplsOutSegmentXCIndex != xcindex )) ->
              {
                  idx = mplsOutSegmentEntry[idx].next;
              }
      :: ( idx == OS_NULL ) ->
             break;
      od;
}
```

The packet shall then be forwarded to the output segments. The new packet header is defined by pushing one or more label stack entries. This is performed by the inline *PUSH_LAB_STACK* macro whose code was omitted but was considered in the state machine diagram. Finally, the new packet is sent to the network receiver of the output interface. The output interface also constrains the label of the leaving packet. If the packet is valid then *__ValidOutLabel__* holds, and the transmission is performed. When multicast is used, the task is repeated for all output segments connected to the same input segment. The interface state machine shown in Fig. 5 is implemented in Promela as a process.

As it can be observed, it is possible to implement a Promela functional model of an MPLS router using its MIBs. We used atomic (inline macros) transitions in order to minimize the state space. The model defined in Promela was optimized and verified using the SPIN tool [10], [22]. By applying the current automatic test generation techniques we expect to be able to generate tests from this model.

7 Conclusion

This paper addresses the testing of IETF network elements with respect mainly to the services they provide. These services are usually managed according to the IETF standards, which include the use of standard Management Information Bases (MIBs).

Automatic test generation requires, at least, one model describing the behaviour of the implementation or system under test. Using this model, tests can be derived so that some implementation relation can be proved, or that the implementation satisfies a set of requirements (test purposes). Behaviour models, however, are not easily available for most of the IETF protocols. More than that, if the network element is addressed as a whole, it consists of a set of combined communication protocols and services.

In this paper we claim that MIBs can be used as the nucleous of a functional model, that can be used to generate tests. In order to demonstrate the claim, but also to identify the limitations of the approach, we applied the concept to an MPLS router. As a result, we obtained a simple functional model, described in Promela, that will be used for test generation.

The main contributions of this paper are the following:

- **MIB based functional model.** The concept that a model relevant for test generation can be obtained by adding behaviour to Management Information Bases is, *to the best of our knowledge*, original. Moreover, this functional model avoids an exhaustive representation of the states associated to message fields, that characterizes the IETF protocols.
- **MPLS router described in Promela.** A simple functional description of an MPLS router was developed in Promela. It can be simulated and was verified using the SPIN related techniques.

This work will continue towards the validation of the concept. If some free tool can be obtained for generating test cases from Promela, we will generate test cases. Based on our experience of testing real dimension test equipments, we will also evaluate what type of faults these tests can address. If satisfied with the results, we will apply the method to another relevant IP network element.

References

1. Manuel Ricardo. *A Methodology for Testing Complex Telecommunications Network Elements.* PhD thesis, Faculdade de Engenharia da Universidade do Porto, Portugal, 2000.
2. William Stallings. *SNMP, SNMPv2, and CMIP: the practical guide to network management.* Addison-Wesley Longman Publishing Co., Inc., 1993.
3. K. Schmidt D. Mauro. *Essential SNMP.* O'Reilly, 2001.
4. IETF. RFC 2438 - Advancement of MIB specifications on the IETF Standards Track, October 1998.
5. M. Schmitt, M. Ebner, and J. Grabowski. Test generation with autolink and testcomposer. In *Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC - SAM.*
6. G. Viho et al. Using on-the-fly verification techniques for the generation of test suites. In *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, pages 348–359. Springer Verlag, 1996.
7. J. Tretmans and A. Belinfante. Automatic Testing with Formal Methods. In *EuroSTAR'99:* 7th *European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12 1999. EuroStar Conferences, Galway, Ireland.

8. Paul Ammann, Paul E. Black, and Wei Ding. Model Checkers in Software Testing. NIST-IR 6777, February 2002. NIST.
9. A. Gargantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Lecture Notes in Computer Science*, volume 1687, Toulose, France, September 1999. ESEC/FSE '99, Springer-Verlag.
10. G.J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
11. J. Tretmans. Conformance Testing with Labelled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, 29:49–79, 1996.
12. A. Kerbrat R. Groz, T. Jron. Automated Test Generation from SDL. In Y. Lahav (eds.) R. Dssouli, G. vonBochmann, editor, *SDL'99 The Next Millenium*, pages pages 135–152, Montral, Qubec, June 1999. 9th SDL Forum, Elsevier.
13. Jean-Claude Fernandez et al. An experiment in automatic generation of test suites for protocols with verification technology. *Science of Computer Programming*, 29(1-2):123–146, 1997.
14. R. de Vries et al. Côte de Resyste in PROGRESS. In STW Technology Foundation, editor, PROGRESS *2000 – Workshop on Embedded Systems*, pages 141–148, Utrecht, The Netherlands, October 13 2000.
15. J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs, 1993.
16. Jens Garbowski. SDL and MSC Based Test Case Generation - An Overall View of the SAMSTAG Method. Technical Report IAM-94-005, University of Berne, Institute for Informatics, Berne, Switzerland,, May 1994.
17. Michael Schmitt, Beat Koch, Jens Grabowski, and Dieter Hogrefe. Autolink - Putting formal test methods into practice, 1997.
18. A. Kerbrat and I. Ober. Automated test generation from SDL/UML specifications. In *The 12th International Software Quality Week*, San Jose, California, May 1999.
19. Ana R. Cavalli, David Lee, Christian Rinderknecht, and Fatiha Zaidi. Hit-or-jump: An algorithm for embedded testing with applications to IN services. In *FORTE*, pages 41–56, 1999.
20. Paul Ammann, Paul E. Black, and William Majurski. Using Model Checking to Generate Tests from Specifications. In *ICFEM*, pages 46–, 1998.
21. J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using modelchecking. In *Proceedings 1996 SPIN Workshop*, aug 1996.
22. Theo C. Ruys. *Towards Effective Model Checking*. PhD thesis, University of Twente, The Netherlands, 2001.
23. IETF. RFC 3031 - Multiprotocol Label Switching Architecture, January 2001.
24. Armitage G. MPLS: The Magic Behind the Myths. In *IEEE Communications Magazine*. IEEE, January 2000. pg.124-131.
25. Tony Li. Mpls and the Evolving Internet Architecture. In *IEEE Communications Magazine*. IEEE, December 1999. pg.38-41.
26. X. Xiao and L. Ni. Internet QoS: A Big Picture. In *IEEE Network*. IEEE, March/April 1999. pg.9-18.
27. IETF. Multiprotocol Label Switching (MPLS) Management Overview, December 2002. draft-ietf-mpls-mgmt-overview-02.
28. IETF. Multiprotocol Label Switching (MPLS) Label Switch Router (LSR) Management Information Base, January 2002. draft-ietf-mpls-lsr-mib-08.
29. G.J. Holzmann. Tutorial: Design and validation of protocols. *Computer Networks and ISDN Systems*, 25(9):981–1017, 1993.