Reverse Engineering Internet MIBs

J. Schönwälder, A. Müller
Computer Science Department
Technical University Braunschweig
Bültenweg 74/75
38106 Braunschweig
Germany
{*schoenw,amueller*}*@ibr.cs.tu-bs.de*

Abstract

The Internet-standard management protocol SNMP manipulates data structures that are defined in MIB modules. A large number of MIB modules has been defined over the last decade. Some of them are rather complex and full of technical details, which makes it hard to understand them. Furthermore, the limitations of the current data definition language make it impossible to formally express the conceptual model underlying a well-engineered MIB module.

This paper presents a reverse engineering algorithm which extracts conceptual models from the MIB data definitions. The algorithm uses several heuristics that are derived from common MIB naming and registration conventions. The output produced by the algorithm is a graphical representation for conceptual MIB models, which is a slightly customized version of a UML class diagram. A prototype implementation of the algorithm is briefly described which has been integrated into the `libsmi` software package.

Keywords

Internet Management, Information Modeling, MIB Engineering, Reverse Engineering, Unified Modeling Language

1 Introduction

The Internet-standard management framework is based on the Simple Network Management Protocol (SNMP) which is used to access and manipulate well-defined data structures on managed devices. These data structures represent the management information relevant for the purpose of managing a network infrastructure and they are usually specific to particular network protocols, device types or management tasks.

The data definitions are organized in so-called Management Information Base (MIB) modules. Many MIB modules have been defined over the last decade. The IETF has standardized about 100 MIB modules with nearly 10,000 defined objects. In addition, there is an even larger and growing number of enterprise-specific MIB modules defined unilaterally by various vendors, research groups, consortia, and the like resulting in an unknown and virtually uncountable number of defined objects.

Our experience with many of these MIB modules reveals that the conceptual models of a MIB module is often hidden behind lots of technical and syntactical details. Nevertheless, it is important to understand the underlying conceptual model in many phases of the MIB module life cycle.

This paper presents work on a reverse engineering algorithm which takes a MIB module as input and extracts the underlying conceptual model. The algorithm uses several heuristics such as common naming conventions to derive properties that are not formally defined in MIB modules. The conceptual model is represented as a slightly customized UML class diagram. The algorithm has been implemented and integrated into the open source `libsmi`[1] MIB compiler package.

The rest of this paper is organized as follows. Section 2 first motivates why a conceptual model is important in a MIB module's life cycle. Section 3 then describes how conceptual MIB models can be represented as a variation of UML class diagrams. Some background information on relationships in MIB data definitions is provided in Section 4 before the reverse engineering algorithm is described in Section 5. Section 6 briefly outlines the implementation and presents an example UML diagram generated from an IETF standards-track MIB module. Some related work is discussed in Section 7 before the paper concludes in Section 8.

2 Conceptual versus Computational Models

The development and implementation of MIB modules is first of all a software engineering process. However, a MIB engineering process has several special characteristics since MIB modules define interfaces in a highly distributed system. Aspects such as long-term stability, proper version handling, extensibility and communication efficiency play an important role. MIB modules are representations of so called computational models. Computational models are specific to a certain implementation technology.

The objective of a conceptual model in a MIB engineering process is to represent the domain of discourse, that is the data involved in the problem to be solved along with the operations on the data, independent of any implementation characteristics [1]. The term "‘conceptual model"’ originates from the field of databases where it refers to the representation of data and their interrelations which are to be managed by an information system, independent of any implementation characteristics. In other words, a conceptual model is usually the output from the analysis stage in a software engineering process while the computational model is the output from the design stage.

This paper argues that the conceptual model plays an important role in the MIB module's life cycle. Some of the reasons are:

[1]`<URL:http://www.ibr.cs.tu-bs.de/projects/libsmi/>`

- Conceptual MIB models are fundamental in the MIB design phase. Organizations like the IETF require that MIB data definitions go through a peer review process to ensure that MIB modules are free of syntactic errors and conform to certain conventions. Every MIB module is assigned to a MIB reviewer before it is published as part of an RFC document. MIB reviewers are usually experienced MIB authors and experts of the SMI data definition language. However, they are usually not experts of the particular problem domain. Hence, it is important that MIB reviewers can easily access the conceptual model that has been used during the design stage in order to reduce the time spent on MIB reviews.

- Management software designers and implementors generally benefit if they understand the conceptual model behind a MIB module. Awareness of the conceptual model allows them to design efficient and extensible implementations that will require less maintenance costs when the MIB module evolves over time.

- MIB modules can have a very long lifetime and it is not unlikely that the authors responsible for a certain MIB module change over time. To ensure stability and consistency, it is important that the conceptual ideas behind a MIB module specification do not get lost when the responsible MIB authors change.

- It is often necessary to educate a larger community (e.g. network operators, software engineers, students) about MIB modules. It has proven to be very effective to start from the underlying conceptual models and to work step by step towards the concrete MIB module specifications.

- Parts of standardized MIB modules are sometimes integrated into more comprehensive information models. This task is greatly simplified by concentrating on the integration of conceptual models rather than the integration of isolated concrete MIB data definitions.

3 Representing Conceptual MIB Models in UML

The Unified Modeling Language (UML) [2, 3] has become the lingua franca for visualizing, specifying and documenting artifacts of software intensive systems. UML is very rich and features several diagram types to represent structural, behavioral and architectural aspects. Current MIB modules only specify structural aspects in a machine readable format. The focus in this paper is therefore on structural aspects, which are usually visualized in UML class diagrams.

Figure 1 shows a hand crafted UML class diagram visualizing the conceptual model underlying the `IF-MIB` module [4] and the `IF-INVERTED-STACK-MIB`

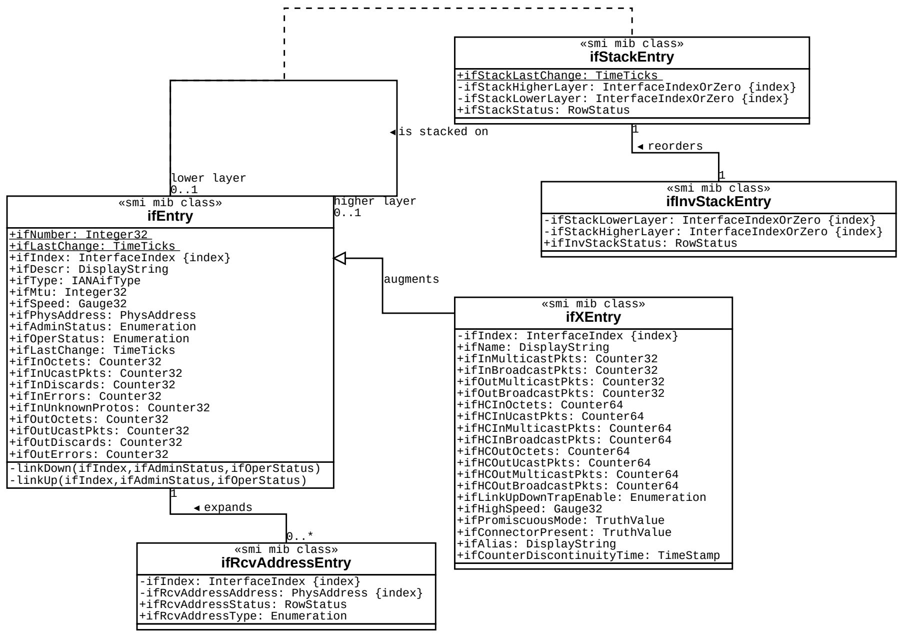
«smi mib class»
ifStackEntry
+ifStackLastChange: TimeTicks
-ifStackHigherLayer: InterfaceIndexOrZero {index}
-ifStackLowerLayer: InterfaceIndexOrZero {index}
+ifStackStatus: RowStatus
◂is stacked on
1
◂ reorders
1
lower layer
0..1
higher layer
0..1
«smi mib class»
ifInvStackEntry
-ifStackLowerLayer: InterfaceIndexOrZero {index}
-ifStackHigherLayer: InterfaceIndexOrZero {index}
+ifInvStackStatus: RowStatus
«smi mib class»
ifEntry
+ifNumber: Integer32
+ifLastChange: TimeTicks
+ifIndex: InterfaceIndex {index}
+ifDescr: DisplayString
+ifType: IANAifType
+ifMtu: Integer32
+ifSpeed: Gauge32
+ifPhysAddress: PhysAddress
+ifAdminStatus: Enumeration
+ifOperStatus: Enumeration
+ifLastChange: TimeTicks
+ifInOctets: Counter32
+ifInUcastPkts: Counter32
+ifInDiscards: Counter32
+ifInErrors: Counter32
+ifInUnknownProtos: Counter32
+ifOutOctets: Counter32
+ifOutUcastPkts: Counter32
+ifOutDiscards: Counter32
+ifOutErrors: Counter32
-linkDown(ifIndex,ifAdminStatus,ifOperStatus)
-linkUp(ifIndex,ifAdminStatus,ifOperStatus)
augments
«smi mib class»
ifXEntry
-ifIndex: InterfaceIndex {index}
+ifName: DisplayString
+ifInMulticastPkts: Counter32
+ifInBroadcastPkts: Counter32
+ifOutMulticastPkts: Counter32
+ifOutBroadcastPkts: Counter32
+ifHCInOctets: Counter64
+ifHCInUcastPkts: Counter64
+ifHCInMulticastPkts: Counter64
+ifHCInBroadcastPkts: Counter64
+ifHCOutOctets: Counter64
+ifHCOutUcastPkts: Counter64
+ifHCOutMulticastPkts: Counter64
+ifHCOutBroadcastPkts: Counter64
+ifLinkUpDownTrapEnable: Enumeration
+ifHighSpeed: Gauge32
+ifPromiscuousMode: TruthValue
+ifConnectorPresent: TruthValue
+ifAlias: DisplayString
+ifCounterDiscontinuityTime: TimeStamp
1
◂ expands
0..*
«smi mib class»
ifRcvAddressEntry
-ifIndex: InterfaceIndex {index}
-ifRcvAddressAddress: PhysAddress {index}
+ifRcvAddressStatus: RowStatus
+ifRcvAddressType: Enumeration

Figure 1: Conceptual model of the `IF-MIB` and the `IF-INVERTED-STACK-MIB`

module [5][2]. There are some important details that distinguish the UML class diagram notation for SMI MIB modules shown in Figure 1 from ordinary UML class diagrams:

1. The classes representing MIB definitions use the ≪`smi mib class`≫ stereotype in order to distinguish them from other UML classes.

2. There is a UML class for each conceptual row definition in a MIB module. The name of the UML class is taken from the row definition.

3. Scalars that are logically bound to a particular UML class are shown as underlined class variables in the UML diagram.

4. Scalars that are not logically bound to a conceptual row are logically grouped into additional UML classes which only contain class attributes.

5. Notifications whose required variables are attributes of a single UML class are shown as private operations for that class. The operation is marked private since it only conceptually exists as an internal interface to emit a notification when the corresponding event occurs.

[2]The diagram only visualizes those definitions that are current and ignores all deprecated or obsolete definitions.

6. Class attributes that are used to uniquely identify a class instance are marked with the {`index`} UML property. The order of the instance identifier components are indicated by the order of the attributes in the UML class representation. Note that all instance identifying attributes are listed, even if they are actually defined in other tables.

7. The standard UML visibility attributes have a slightly different meaning. Public visibility (indicated by a $+$ symbol) implies that the corresponding attribute is publically accessible for reading and/or writing. Private visibility (indicated by a $-$ symbol) implies that the attribute is not directly accessible, e.g. because it is a not-accessible index component of a MIB table. Note that an attribute with private visibility can still be read indirectly by extracting the actual value from the instance identifier of another readable attribute of the same UML class.

Figure 1 shows several different relationships between the UML classes. The generalization between the `ifEntry` and the `ifXEntry` class represents a MIB table augmentation where every `ifXEntry` instance is bound to an `ifEntry` instance and vice versa. Table augmentations are frequently used in MIBs to extend an existing table definition. The association between `ifRcvAddressEntry` and `ifEntry` shows a MIB table extension where multiple `ifRcvAddressEntry` instances can relate to a single `ifEntry` instance. This is indicated by the cardinalities of the UML association.

The `ifStackEntry` is an association class. It represents properties of the “is stacked on” relationship between network interfaces. The association class itself has a class attribute (the MIB scalar `ifStackLastChange`) which indicates the timestamp of the last change in an `ifStackEntry` instance. Finally, the association between an `ifStackEntry` and an `ifInvStackEntry` represents a MIB table reordering relationship where the not-accessible index components have been reordered. (Such reorder relationships enable management applications to read tables more efficiently and are thus an SNMP specific optimization.)

Figure 2 shows a hand crafted UML class diagram for the `HOST-RESOURCES-MIB` [6]. This diagram clearly shows the generalization/specialization relationship between the `hrDeviceEntry` and the derived classes `hrProcessorEntry`, `hrNetworkEntry`, `hrPrinterEntry`, and `hrDiskStorageEntry`. The “implements” association links the `hrNetworkEntry` class to the `ifEntry` class of the `IF-MIB` shown in Figure 1.

4 Relationships in MIBs

The SMIv2 data definition language [7, 8, 9] only provides limited mechanisms to formally express relationships between MIB objects. In particular, the SMIv2 provides no mechanism to formally express that a certain column points to another table

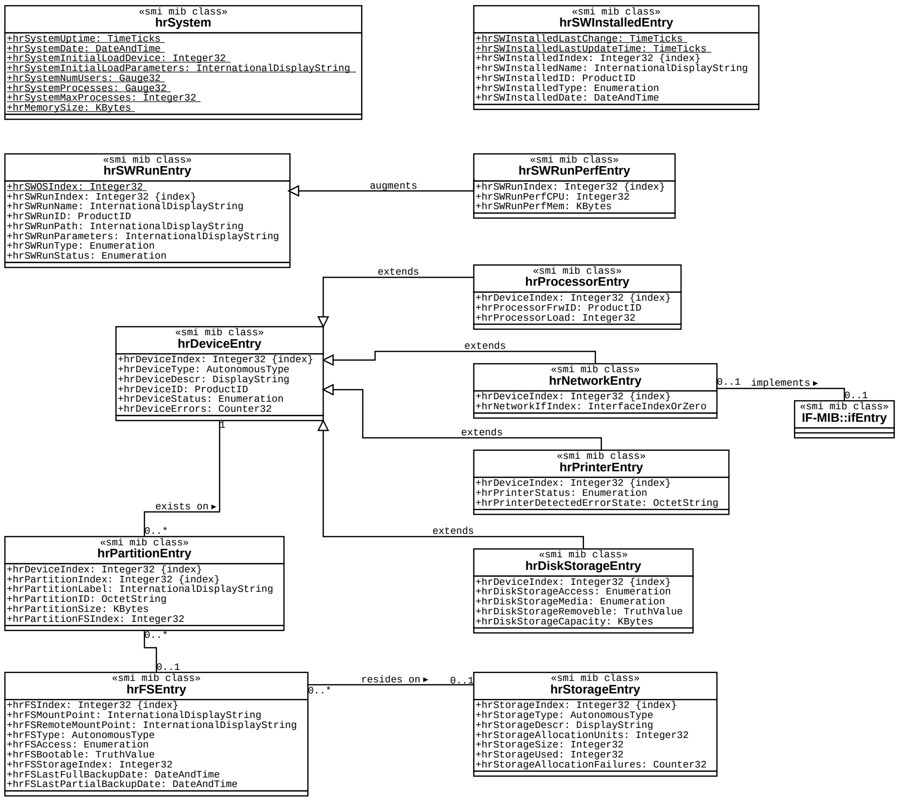
«smi mib class»
hrSystem
+hrSystemUptime: TimeTicks
+hrSystemDate: DateAndTime
+hrSystemInitialLoadDevice: Integer32
+hrSystemInitialLoadParameters: InternationalDisplayString
+hrSystemNumUsers: Gauge32
+hrSystemProcesses: Gauge32
+hrSystemMaxProcesses: Integer32
+hrMemorySize: KBytes
«smi mib class»
hrSWInstalledEntry
+hrSWInstalledLastChange: TimeTicks
+hrSWInstalledLastUpdateTime: TimeTicks
+hrSWInstalledIndex: Integer32 {index}
+hrSWInstalledName: InternationalDisplayString
+hrSWInstalledID: ProductID
+hrSWInstalledType: Enumeration
+hrSWInstalledDate: DateAndTime
«smi mib class»
hrSWRunEntry
+hrSWOSIndex: Integer32
+hrSWRunIndex: Integer32 {index}
+hrSWRunName: InternationalDisplayString
+hrSWRunID: ProductID
+hrSWRunPath: InternationalDisplayString
+hrSWRunParameters: InternationalDisplayString
+hrSWRunType: Enumeration
+hrSWRunStatus: Enumeration
augments
«smi mib class»
hrSWRunPerfEntry
+hrSWRunIndex: Integer32 {index}
+hrSWRunPerfCPU: Integer32
+hrSWRunPerfMem: KBytes
extends
«smi mib class»
hrProcessorEntry
+hrDeviceIndex: Integer32 {index}
+hrProcessorFrwID: ProductID
+hrProcessorLoad: Integer32
«smi mib class»
hrDeviceEntry
+hrDeviceIndex: Integer32 {index}
+hrDeviceType: AutonomousType
+hrDeviceDescr: DisplayString
+hrDeviceID: ProductID
+hrDeviceStatus: Enumeration
+hrDeviceErrors: Counter32
extends
«smi mib class»
hrNetworkEntry
+hrDeviceIndex: Integer32 {index}
+hrNetworkIfIndex: InterfaceIndexOrZero
0..1
implements ▸
0..1
«smi mib class»
IF-MIB::ifEntry
extends
«smi mib class»
hrPrinterEntry
+hrDeviceIndex: Integer32 {index}
+hrPrinterStatus: Enumeration
+hrPrinterDetectedErrorState: OctetString
1
exists on ▸
0..*
extends
«smi mib class»
hrPartitionEntry
+hrDeviceIndex: Integer32 {index}
+hrPartitionIndex: Integer32 {index}
+hrPartitionLabel: InternationalDisplayString
+hrPartitionID: OctetString
+hrPartitionSize: KBytes
+hrPartitionFSIndex: Integer32
«smi mib class»
hrDiskStorageEntry
+hrDeviceIndex: Integer32 {index}
+hrDiskStorageAccess: Enumeration
+hrDiskStorageMedia: Enumeration
+hrDiskStorageRemoveble: TruthValue
+hrDiskStorageCapacity: KBytes
0..*
0..1
«smi mib class»
hrFSEntry
+hrFSIndex: Integer32 {index}
+hrFSMountPoint: InternationalDisplayString
+hrFSRemoteMountPoint: InternationalDisplayString
+hrFSType: AutonomousType
+hrFSAccess: Enumeration
+hrFSBootable: TruthValue
+hrFSStorageIndex: Integer32
+hrFSLastFullBackupDate: DateAndTime
+hrFSLastPartialBackupDate: DateAndTime
0..*
resides on ▸
0..1
«smi mib class»
hrStorageEntry
+hrStorageIndex: Integer32 {index}
+hrStorageType: AutonomousType
+hrStorageDescr: DisplayString
+hrStorageAllocationUnits: Integer32
+hrStorageSize: Integer32
+hrStorageUsed: Integer32
+hrStorageAllocationFailures: Counter32

Figure 2: Conceptual model of the `HOST-RESOURCES-MIB`

or a particular column in another table. Since the knowledge of relationships is essential for understanding a MIB module, MIB authors usually follow certain naming conventions to indicate where a referencing columns points to. Another frequently used technique is to introduce special types which implicitly identify the referenced table.

4.1 Table Existence Relationships

Table existence relationships express the conditions under which rows in different tables share fate. In other words, these fate-sharing relationships define whether the existence of a particular row is bound to the existence of other rows in other tables [10]. Only a few of these existence relationships are described with machine readable constructs in the SMIv2.

One-to-one table augmentations represent extensions of a base table where each instance in the augmenting table exists according to the same semantics as instances in the augmented table. In other words, the cardinality of the association between both tables is 1:1. The `ifXTable` in the `IF-MIB` is an example for a one-to-one augmentation of the `ifTable`. One-to-one table augmentations are frequently used to extend existing tables without having to modify the existing table. In particular, an augmenting table can live in a separate vendor specific MIB module and augment standardized tables. The SMIv2 language provides the `AUGMENTS` clause to define one-to-one table augmentations.

Sparse table augmentations are similar to one-to-one table augmentations, except that not all rows in the augmented table must be present in the augmenting table. The cardinality of the association between both tables is therefore 1:0..1. This mechanism is frequently used to extend a base table with specific information that is only applicable to a subset of the rows in the base table. The `hrNetworkTable` in the `HOST-RESOURCES-MIB` is an example of a sparse table augmentation of the `hrDeviceTable`. The SMIv2 language does not provide a specific clause to define sparse table augmentations. However, a common way to define sparse table augmentations is to use identical `INDEX` clauses for the base table and the extending tables.

Table expansions are used to model multi-valued attributes that are related to a row in a base table. The cardinality of the association between both tables is 1:0..n. An example for a table expansion is the `ifRcvAddressTable` in the `IF-MIB` which expands the `ifTable` by listing the set of physical addresses which will be used to receive frames on a particular interface. The SMIv2 language does not provide a specific clause to define table expansions. Table expansions are usually defined by copying the `INDEX` clause of the base table and adding additional elements at the end.

The *table reordering relationship* exists between two tables if they have the same number of rows and only the ordering of the elements in the `INDEX` clause differs. The cardinality of the association between both tables is 1:1. Table reorderings are useful to speedup certain lookups in potentially large tables. The `ifInvStackTable` in the `IF-INVERTED-STACK-MIB` for example reorders the `ifStackTable` in the `IF-MIB`. The SMIv2 language again does not provide a specific clause to define table reorderings.

It should be noted that the proposed SMIng [11] provides language support to formally define all four table existence relationships.

4.2 Reference Relationships

There are cases where relationships between MIB rows are defined by using explicit pointers. The SMIv2 provides special data types for this purpose [8]: `RowPointer` and `VariablePointer`. It is also common that MIB authors introduce special types that directly relate to the instance identification used by the referenced table. A good example is the `hrNetworkIfIndex` column of the `hrNetworkTable` in the `HOST-RESOURCES-MIB` which points to a row of the `ifTable` in the `IF-MIB`. A common way to mark these reference relationships is to encode the name of the referenced column in the name of the referring column. In more recent MIB modules, MIB authors introduce special types that can be used by the referring column (`InterfaceIndexOrZero` in our example).

4.3 Tagging Relationships

The `SNMP-TARGET-MIB` [12] introduces a tagging mechanism for efficiently representing associations of the cardinality n:m. A tagged table has a column of type `SnmpTagList`, which holds a list of words delimited by white space characters. The referring table has a column of type `SnmpTagValue`, which only contains a single word without any white space characters. The `SnmpTagValue` column usually refers to all rows in the tagged table which contain a specific `SnmpTagValue` value in the tag list in the `SnmpTagList` column.

5 Reverse Engineering Algorithm

The reverse engineering algorithm described below constructs a graph which represents the conceptual model of a given MIB module. The nodes of the graph represent the classes of the UML class diagram while the edges represent relationships between these classes. The algorithm can be divided into ten steps. Each step analyzes some specific MIB information in order to refine the graph. Below is the description of the steps based on the SMIv2 MIB module language [7, 8, 9]. Minor modifications of the algorithm are possible to process MIB modules that conform to the experimental SMIng [11] and to take advantage of the improved SMIng capabilities to express table existence relationships.

5.1 Description of the Algorithm

Below is a step-by-step description of the algorithm.

Step 1: Creating Nodes

The first step populates the graph by creating nodes for all tables and scalars in the MIB module. It is possible to restrict the number of nodes generated by excluding objects whose status is obsolete or deprecated. Note that scalars are not grouped in the first step since some of them may be merged with nodes that represent tables in subsequent steps.

Step 2: Creating Edges from Existence Relationships

The second step of the algorithm creates edges between graph nodes by interpreting `AUGMENTS` and `INDEX` clauses. The algorithm detects the table existence relationships introduced in Section 4.1 as follows:

- One-to-one table augmentations are detected easily by checking for the presence of `AUGMENTS` clauses. The algorithm adds a new edge to the graph for each `AUGMENTS` clause.
- Sparse table augmentations are detected by analyzing all `INDEX` clauses of the MIB tables. Edges are created between nodes that represent tables where the `INDEX` clauses satisfy the following two conditions: First, the number of elements in the `INDEX` clauses is equal. Second, the elements in both `INDEX` clauses are identical and occur in the same order.

  Once a sparse table augmentation has been detected, it is necessary to decide which of the tables is the augmented table and which is the augmenting table. The heuristic rule used to make a decision is to assume that the table which is registered first in the OID tree is the augmented table.
- New edges representing table expansions are created between nodes that represent tables which satisfy the following conditions: First, the number of elements in the `INDEX` clause of one table must be greater than the number of elements in the `INDEX` clause of a second table. Second, all elements that are listed in the `INDEX` clause with fewer elements must be present in the `INDEX` clause of the other table and they must be listed at the same position.

  Simply checking these rules may result in bogus edges since it is possible to have multiple levels of table expansions. It is thus useful to search for the longest overlap in `INDEX` clauses first before using the position in the OID registration tree to break any ties.
- Reorder relationships are easily detected by checking for the following two rules: First, the number of elements in the `INDEX` clauses of two tables are identical. Second, the elements are exactly the same appearing in a different order. The reordered table is assumed to be the table which appears first in the OID registration tree.

Step 3: Reordering Edges

The third step reorders edges. This step is necessary since the previous step may combine sparse augmentations and table expansions in non-obvious ways. Therefore, every expand relationship is checked whether there is a sparse augmentation of the base table so that the expand relationship can be re-routed to the augmenting table. The decision whether the expand relationship should be reordered is based on the commonality of the table descriptors.

The commonality of two descriptors is determined by first computing normalized names. A normalized name is constructed by removing prefixes that are common to all descriptors defined in a MIB module. Next, the number of identical characters in two normalized names is counted, starting at the beginning of the normalized names.

If the algorithm detects a table expansion where the base table has a sparse augmentation where the commonality between the expanding table descriptor and the augmenting table descriptor is higher, then the edge representing the expansion relationship is rerouted from the base table to the augmenting table.

Step 4: Creating Edges based on Index Reference Relationships

Section 4.2 described that it is common practice in newer MIBs to introduce special types for instance identification and instance referencing purposes. Step four tries to take advantage of this. It loops over all nodes representing tables and checks whether there are any other tables indexed by columns of similar types. Some well known and frequently used type names are ignored in this step to prevent the algorithm from generating too many false edges. If the algorithm detects more than one match, then the first table in OID order or the table with the most common name prefix is selected. It is possible that some types only differ in their range by exactly one number. A good example are the types `InterfaceIndex` and `InterfaceIndexOrZero` defined in the `IF-MIB`. The algorithm will treat those types as identical if their descriptors only differ in a common suffix such as `OrZero`.

Step 5: Creating Edges based on Name Prefixes

The fifth step tries to link the remaining tables that are not yet linked to any other tables. For each remaining table, a test is being made whether the name has commonality with other table names. The test is again based on normalized names in order to deal with the convention of using a common prefix for all descriptors defined in a MIB module. It is obvious that only the nodes with the most common normalized name are selected to create new edges.

Step 6: Assigning Scalars to Tables

The algorithm assigns scalars to tables in the sixth step. This is again done by comparing normalized scalar descriptors with normalized table descriptors. If there are common prefixes in the normalized names, then the scalar is assigned to the table with the longest matching prefix. The OID order is used as a second criterion if multiple potential candidates exist.

Step 7: Grouping Scalars

The seventh step groups all scalars that are not assigned to a table into classes. A new class is created for each set of scalars which share a common parent node in the OID tree.

Step 8: Dependency Relationships

Dependency relationships are detected by first identifying so called supporting tables. A supporting table is a table which only holds index objects and support objects of well know types such as `RowStatus` and `StorageType`. If a supporting table is found, then the `INDEX` objects are compared to any other table. If a match is found and there exists already an edge between these tables, then the edge is changed to a dependency association. Otherwise a new edge is introduced to represent the dependency association.

Step 9: Reference Relationships

The nineth step tries to detect reference relationships. This is done by checking the descriptors of columnar objects. First, the descriptor is checked whether it ends with a suffix which typically indicates a pointing object (e.g. `Index` or `Pointer`). Having found such a column descriptor, the descriptor is shortened by removing the common table prefix. Next, all other tables are checked whether the shortened name appears in any of the `INDEX` objects. A new edge is created if such a matching table is found.

Step 10: Notifications

The final step assigns notifications to nodes representing MIB tables. This is done by checking whether the mandatory objects listed in the `NOTIFICATION-TYPE` clause all appear in a single table. If the test is successful, then the notification is assigned as an operation to the node representing the table.

5.2 Limitations of the Algorithm

The algorithm has several limitations since it depends on many "unwritten" rules for writing SMI MIB modules. MIB authors can choose their own names for the objects they define. There is no requirement to use the same prefix (such as `if` in the `IF-MIB`) or to use similar names for related objects. The algorithm, however, depends on these naming conventions and it will produce less meaningful results if for example all descriptors are written in lower case. The same is true for the use of specialized data types. Good MIB authors introduce new data types which can be exploited to recognize references between MIB tables. MIB authors who decide to use only base types or inlined type definitions will make it hard for the algorithm to produce meaningful results.

The definition of the algorithm so far assumes that only one MIB module is reverse engineered at a time. It is desirable to generalize the algorithm so that multiple MIB modules can be reverse engineered together. This basically requires to improve the computation of normalized names.

The algorithm sometimes uses the order of definitions in the OID tree to select one of several possible alternatives. The motivation behind this rule is that more fundamental definitions are usually introduced and thus registered first. It might be necessary to find better heuristics when the algorithm is generalized so that it can operate on multiple MIB modules which may be rooted at arbitrary locations in the OID tree.

6 Implementation and Examples

The algorithm described in the previous section has been implemented and integrated into the `smidump` program of the `libsmi` SMI compiler package [11]. The implementation can be used with MIB modules that conform to SMIv1, SMIv2 or SMIng since the API of the `libsmi` library hides most of the details between the various SMI versions. The internal graph constructed by the algorithm can be written in several output formats. The `cm-xplain` output format gives a textual explanation. Selecting the `cm-dia` output format will generate an XML representation of the UML diagram which can be loaded into the `dia`[3] diagram editor.

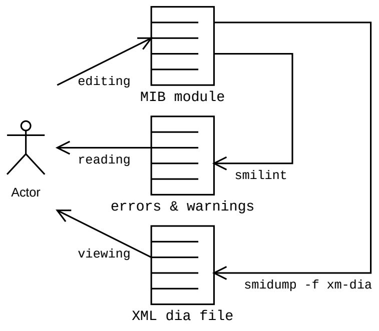

Figure 3: Using the `smilint` and `smidump` utilities during MIB design

A typical use case is shown in Figure 3. A MIB author edits a MIB module and runs it through the `smilint` syntax checker which generates error and warning messages. The MIB author also uses the `smidump` program to generate the UML class diagram which is stored in an XML file format. The XML file is then displayed by the `dia` editor. The MIB author now uses the error and warning messages and the UML diagram to further edit the MIB module specification.

Figure 4 shows the UML class diagram generated by feeding the `IF-MIB` [4] into `smidump`. The output does not 100 % match the hand crafted UML diagram shown in Figure 1. For example, the `ifNumber` scalar was not assigned to the `ifEntry` class since there is no overlap in the normalized names once the prefix `if` has been removed.

Although the process described in Figure 3 works reasonably well, one could imagine to integrate the algorithm described in this paper and parts of the `smidump` utility as a plug-in into the `dia` editor. This would address the limitation that all information about node placements is currently lost when the UML diagram is re-generated. Integrating some of the `smidump` functionality into `dia` would result in an integrated SMI MIB engineering environment with a graphical UML front-end.

7 Related Work

The Rose MIB Link product from ObjectStream[4] supports round-trip mappings from SMI MIBs to UML and back. The mapping is rather simple and follows directly the

[3]`<URL:http://www.lysator.liu.se/~alla/dia/>`
[4]`<URL:http://www.objectstream.com/>`

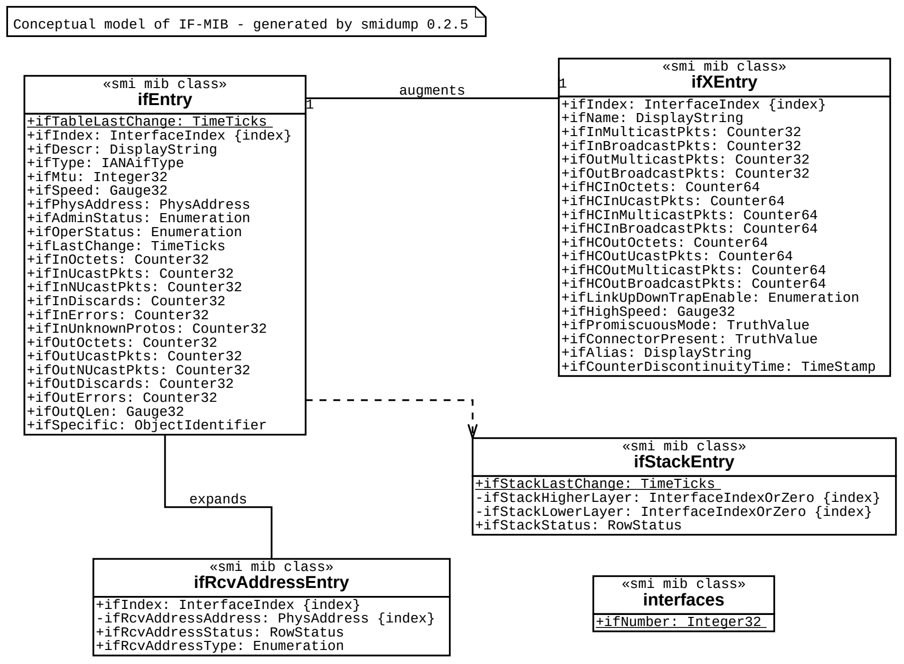
Conceptual model of IF-MIB - generated by smidump 0.2.5
«smi mib class»
ifEntry
+ifTableLastChange: TimeTicks
+ifIndex: InterfaceIndex {index}
+ifDescr: DisplayString
+ifType: IANAifType
+ifMtu: Integer32
+ifSpeed: Gauge32
+ifPhysAddress: PhysAddress
+ifAdminStatus: Enumeration
+ifOperStatus: Enumeration
+ifLastChange: TimeTicks
+ifInOctets: Counter32
+ifInUcastPkts: Counter32
+ifInNUcastPkts: Counter32
+ifInDiscards: Counter32
+ifInErrors: Counter32
+ifInUnknownProtos: Counter32
+ifOutOctets: Counter32
+ifOutUcastPkts: Counter32
+ifOutNUcastPkts: Counter32
+ifOutDiscards: Counter32
+ifOutErrors: Counter32
+ifOutQLen: Gauge32
+ifSpecific: ObjectIdentifier
augments
1
1
«smi mib class»
ifXEntry
+ifIndex: InterfaceIndex {index}
+ifName: DisplayString
+ifInMulticastPkts: Counter32
+ifInBroadcastPkts: Counter32
+ifOutMulticastPkts: Counter32
+ifOutBroadcastPkts: Counter32
+ifHCInOctets: Counter64
+ifHCInUcastPkts: Counter64
+ifHCInMulticastPkts: Counter64
+ifHCInBroadcastPkts: Counter64
+ifHCOutOctets: Counter64
+ifHCOutUcastPkts: Counter64
+ifHCOutMulticastPkts: Counter64
+ifHCOutBroadcastPkts: Counter64
+ifLinkUpDownTrapEnable: Enumeration
+ifHighSpeed: Gauge32
+ifPromiscuousMode: TruthValue
+ifConnectorPresent: TruthValue
+ifAlias: DisplayString
+ifCounterDiscontinuityTime: TimeStamp
«smi mib class»
ifStackEntry
+ifStackLastChange: TimeTicks
-ifStackHigherLayer: InterfaceIndexOrZero {index}
-ifStackLowerLayer: InterfaceIndexOrZero {index}
+ifStackStatus: RowStatus
expands
«smi mib class»
ifRcvAddressEntry
+ifIndex: InterfaceIndex {index}
-ifRcvAddressAddress: PhysAddress {index}
+ifRcvAddressStatus: RowStatus
+ifRcvAddressType: Enumeration
«smi mib class»
interfaces
+ifNumber: Integer32

Figure 4: Conceptual model of the `IF-MIB` generated by `smidump`

OID registration tree. MIB tables and sets of scalars with a common root node are mapped to a UML class. These classes use the ≪`Managed Object Class`≫ stereotype. Intermediate MIB nodes which are usually used to structure the OID space are mapped to UML classes using the ≪`OID`≫ stereotype. ≪`Managed Object Class`≫ classes are linked to ≪`OID`≫ classes using UML aggregations.

The JIDM specification translation rules [13] define how SMIv2 data definitions are mapped to CORBA interface definitions. The mapping of table rows to classes is similar to what has been proposed in this paper. However, the JIDM translation rules do not make any attempts to identify and translate relationships between MIB tables. The JIDM handling of scalars is also much simpler since no attempt is being made to assign scalars to tables.

8 Conclusions

Given the large number of SMI MIB modules that have been defined in the past, it is necessary to improve the documentation of the conceptual models underlying these MIB module. This paper presented an algorithmic approach to reverse engineer SMI MIB modules into conceptual MIB models.

This paper first described how UML class diagrams can be utilized to document the conceptual model behind a MIB module. Afterwards, a heuristic reverse engineering algorithm was presented which takes a MIB module as input and outputs a UML class diagram. The algorithm has been implemented and is freely available as part of the `libsmi` SMI compiler toolkit.

In the future, we plan to improve and fine-tune the heuristics used by the algorithm. It may also be interesting to investigate whether it is feasible to provide tighter integration into a UML-based modeling environment, which would lead to an integrated SMI MIB reverse and forward engineering toolkit.

Finally, based on the experiences obtained by developing and using this reverse engineering algorithm, we plan to revise the SMIng proposal [11] to provide stronger support for formally defining relationships between MIB tables. This will reduce the number of heuristics needed to turn a MIB module definition into a useful UML class diagram.

References

[1] N. Juristo and A. M. Moreno. Introductory paper: Reflections on Conceptual Modeling. *Data and Knowledge Engineering*, 33(2):103–117, May 2000.

[2] Object Management Group. Unified Modeling Language Specification Version 1.3. Formal Specification, Object Management Group, March 2000.

[3] G. Booch, J. Rumbaugh, and I. Jacobsen. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.

[4] K. McCloghrie and F. Kastenholz. The Interfaces Group MIB. RFC 2863, June 2000.

[5] K. McCloghrie and G. Hanson. The Inverted Stack Table Extension to the Interfaces Group MIB. RFC 2864, June 2000.

[6] S. Waldbusser and P. Grillo. Host Resources MIB. RFC 2790, March 2000.

[7] K. McCloghrie, D. Perkins, J. Schönwälder, J. Case, M. Rose, and S. Waldbusser. Structure of Management Information Version 2 (SMIv2). RFC 2578, April 1999.

[8] K. McCloghrie, D. Perkins, J. Schönwälder, J. Case, M. Rose, and S. Waldbusser. Textual Conventions for SMIv2. RFC 2579, April 1999.

[9] K. McCloghrie, D. Perkins, J. Schönwälder, J. Case, M. Rose, and S. Waldbusser. Conformance Statements for SMIv2. RFC 2580, April 1999.

[10] D. T. Perkins. Intertable Indexing in SNMP MIBs. White Paper, SNMPinfo, July 1998.

[11] J. Schönwälder and F. Strauß. Next Generation Structure of Management Information for the Internet. In *Proc. 10th IFIP/IEEE Workshop on Distributed Systems: Operations and Management*, pages 93–106. Springer Verlag, October 1999.

[12] D. Levi, P. Meyer, and B. Stewart. SNMP Applications. RFC 2573, April 1999.

[13] The Open Group. Inter-Domain Management: Specification Translation & Interaction Translation. Technical Standard C802, January 2000.