Network Working Group
Request for Comments: 3780
Category: Experimental

F. Strauss
TU Braunschweig
J. Schoenwaelder
International University Bremen

May 2004

SMIng - Next Generation Structure of Management Information

Status of this Memo

This memo defines an Experimental Protocol for the Internet community. It does not specify an Internet standard of any kind. Discussion and suggestions for improvement are requested. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This memo defines the base SMIng (Structure of Management Information, Next Generation) language. SMIng is a data definition language that provides a protocol-independent representation for management information. Separate RFCs define mappings of SMIng to specific management protocols, including SNMP.

Table of Contents

1. Introduction

In traditional management systems, management information is viewed as a collection of managed objects, residing in a virtual information store, termed the Management Information Base (MIB). Collections of related objects are defined in MIB modules. These modules are written in conformance with a specification language, the Structure of Management Information (SMI). There are different versions of the SMI. The SMI version 1 (SMIv1) is defined in [RFC1155], [RFC1212], [RFC1215], and the SMI version 2 (SMIv2) in [RFC2578], [RFC2579], and [RFC2580]. Both are based on adapted subsets of OSI's Abstract Syntax Notation One, ASN.1 [ASN1].

In a similar fashion, policy provisioning information is viewed as a collection of Provisioning Classes (PRCs) and Provisioning Instances (PRIs) residing in a virtual information store, termed the Policy Information Base (PIB). Collections of related Provisioning Classes are defined in PIB modules. PIB modules are written using the Structure of Policy Provisioning Information (SPPI) [RFC3159] which is an adapted subset of SMIv2.

The SMIv1 and the SMIv2 are bound to the Simple Network Management Protocol (SNMP) [RFC3411], while the SPPI is bound to the Common Open Policy Service Provisioning (COPS-PR) Protocol [RFC3084]. Even though the languages have common rules, it is hard to use common data definitions with both protocols. It is the purpose of this document to define a common data definition language, named SMIng, that can

formally specify data models independent of specific protocols and applications. The appendix of this document defines a core module that supplies common SMIng definitions.

A companion document contains an SMIng language extension to define SNMP specific mappings of SMIng definitions in compatibility with SMIv2 MIB modules [RFC3781]. Additional language extensions may be added in the future, e.g., to define COPS-PR specific mappings of SMIng definitions in a way that is compatible with SPPI PIBs.

Section 2 gives an overview of the basic concepts of data modeling using SMIng, while the subsequent sections present the concepts of the SMIng language in detail: the base types, the SMIng file structure, and all SMIng core statements.

The remainder of the document describes extensibility features of the language and rules to follow when changes are applied to a module. Appendix B contains the grammar of SMIng in ABNF [RFC2234] notation.

1.1. The History of SMIng

SMIng started in 1999 as a research project to address some drawbacks of SMIv2, the current data modeling language for management information bases. Primarily, its partial dependence on ASN.1 and a number of exception rules turned out to be problematic. In 2000, the work was handed over to the IRTF Network Management Research Group where it was significantly detailed. Since the work of the RAP Working Group on COPS-PR and SPPI emerged in 1999/2000, SMIng was split into two parts: a core data definition language (defined in this document) and protocol mappings to allow the application of core definitions through (potentially) multiple management protocols. The replacement of SMIv2 and SPPI by a single merged data definition language was also a primary goal of the IETF SMING Working Group that was chartered at the end of 2000.

The requirements for a new data definition language were discussed several times within the IETF SMING Working Group and changed significantly over time [RFC3216], so that another proposal (in addition to SMIng), named SMI Data Structures (SMI-DS), was presented to the Working Group. In the end, neither of the two proposals found enough consensus and support, and the attempt to merge the existing concepts did not succeed, resulting in the Working Group being closed down in April 2003.

In order to record the work of the NMRG (Network Management Research Group) on SMIng, this memo and the accompanying memo on the SNMP protocol mapping [RFC3781] have been published for informational purposes.

Note that throughout these documents, the term "SMIng" refers to the specific data modeling language that is specified in this document, whereas the term "SMING" refers to the general effort within the IETF Working Group to define a new management data definition language as an SMIv2 successor and probably an SPPI merger, for which "SMIng" and "SMI-DS" were two specific proposals.

1.2. Terms of Requirement Levels

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

2. SMIng Data Modeling

SMIng is a language designed to specify management information in a structured way readable to computer programs, e.g., MIB compilers, as well as to human readers.

Management information is modeled in classes. Classes can be defined from scratch or by derivation from a parent class. Derivation from multiple parent classes is not possible. The concept of classes is described in Section 9.

Each class has a number of attributes. Each attribute represents an atomic piece of information of a base type, a sub-type of a base type, or another class. The concept of attributes is described in Section 9.2.

The base types of SMIng include signed and unsigned integers, octet strings, enumeration types, bitset types, and pointers. Pointers are references to class instances, attributes of class instances, or arbitrary identities. The SMIng type system is described in Section 3.

Related class and type definitions are defined in modules. A module may refer to definitions from other modules by importing identifiers from those modules. Each module may serve one or multiple purposes:

- the definition of management classes,
- the definition of events,
- the definition of derived types,
- the definition of arbitrary untyped identities serving as values of pointers,

- the definition of SMIng extensions allowing the local module or other modules to specify information beyond the scope of the base SMIng in a machine readable notation. Some extensions for the application of SMIng in the SNMP framework are defined in [RFC3781],

- the definition of information beyond the scope of the base SMIng statements, based on locally defined or imported SMIng extensions.

Each module is identified by an upper-case identifier. The names of all standard modules must be unique (but different versions of the same module should have the same name). Developers of enterprise modules are encouraged to choose names for their modules that will have a low probability of colliding with standard or other enterprise modules, e.g., by using the enterprise or organization name as a prefix.

2.1. Identifiers

Identifiers are used to identify different kinds of SMIng items by name. Each identifier is valid in a namespace which depends on the type of the SMIng item being defined:

- The global namespace contains all module identifiers.

- Each module defines a new namespace. A module's namespace may contain definitions of extension identifiers, derived type identifiers, identity identifiers, and class identifiers. Furthermore, a module may import identifiers of these kinds from other modules. All these identifiers are also visible within all inner namespaces of the module.

- Each class within a module defines a new namespace. A class' namespace may contain definitions of attribute identifiers and event identifiers.

- Each enumeration type and bitset type defines a new namespace of its named numbers. These named numbers are visible in each expression of a corresponding value, e.g., default values and sub-typing restrictions.

- Extensions may define additional namespaces and have additional rules of other namespaces' visibility.

Within every namespace each identifier MUST be unique.

Each identifier starts with an upper-case or lower-case character, dependent on the kind of SMIng item, followed by zero or more letters, digits, and hyphens.

All identifiers defined in a namespace MUST be unique and SHOULD NOT only differ in case. Identifiers MUST NOT exceed 64 characters in length. Furthermore, the set of all identifiers defined in all modules of a single standardization body or organization SHOULD be unique and mnemonic. This promotes a common language for humans to use when discussing a module.

To reference an item that is defined in the local module, its definition MUST sequentially precede the reference. Thus, there MUST NOT be any forward references.

To reference an item that is defined in an external module it MUST be imported (Section 5.1). Identifiers that are neither defined nor imported MUST NOT be visible in the local module.

When identifiers from external modules are referenced, there is the possibility of name collisions. As such, if different items with the same identifier are imported or if imported identifiers collide with identifiers of locally defined items, then this ambiguity is resolved by prefixing those identifiers with the names of their modules and the namespace operator `::', i.e., `Module::item'. Of course, this notation can be used to refer to identifiers even when there is no name collision.

Note that SMIng core language keywords MUST NOT be imported. See the `...Keyword' rules of the SMIng ABNF grammar in Appendix B for a list of those keywords.

3. Base Types and Derived Types

SMIng has a set of base types, similar to those of many programming languages, but with some differences due to special requirements from the management information model.

Additional types may be defined, derived from those base types or from other derived types. Derived types may use subtyping to formally restrict the set of possible values. An initial set of commonly used derived types is defined in the SMIng standard module NMRG-SMING [RFC3781].

The different base types and their derived types allow different kinds of subtyping, namely size restrictions of octet strings (Section 3.1), range restrictions of numeric types (Section 3.4

through Section 3.10), restricted pointer types (Section 3.2), and restrictions on the sets of named numbers for enumeration types (Section 3.11) and bit sets (Section 3.12).

3.1. OctetString

The OctetString base type represents arbitrary binary or textual data. Although SMIng has a theoretical size limitation of 2^16-1 (65535) octets for this base type, module designers should realize that there may be implementation and interoperability limitations for sizes in excess of 255 octets.

Values of octet strings may be denoted as textual data enclosed in double quotes or as arbitrary binary data denoted as a `0x'-prefixed hexadecimal value of an even number of at least two hexadecimal digits, where each pair of hexadecimal digits represents a single octet. Letters in hexadecimal values MAY be upper-case, but lower-case characters are RECOMMENDED. Textual data may contain any number (possibly zero) of any 7-bit displayable ASCII characters, including tab characters, spaces, and line terminator characters (nl or cr & nl). Some characters require a special encoding (see Section 4.2). Textual data may span multiple lines, where each subsequent line prefix containing only white space up to the column where the first line's data starts SHOULD be skipped by parsers for a better text formatting.

When defining a type derived (directly or indirectly) from the OctetString base type, the size in octets may be restricted by appending a list of size ranges or explicit size values, separated by pipe `|' characters, with the whole list enclosed in parenthesis. A size range consists of a lower bound, two consecutive dots `..', and an upper bound. Each value can be given in decimal or `0x'-prefixed hexadecimal notation. Hexadecimal numbers must have an even number of at least two digits. Size restricting values MUST NOT be negative. If multiple values or ranges are given, they all MUST be disjoint and MUST be in ascending order. If a size restriction is applied to an already size restricted octet string, the new restriction MUST be equal or more limiting, that is, raising the lower bounds, reducing the upper bounds, removing explicit size values or ranges, or splitting ranges into multiple ranges with intermediate gaps.

Value Examples:

```
"This is a multiline
 textual data example."          // legal
"This is "illegally" quoted."    // illegal quotes
"This is \"legally\" quoted."    // legally encoded quotes
"But this is 'ok', as well."     // legal apostrophe quoting
""                               // legal zero length
0x123                            // illegal odd hex length
0x534d496e670a                   // legal octet string
```

Restriction Examples:

```
OctetString (0 | 4..255)         // legal size spec
OctetString (4)                  // legal exact size
OctetString (-1 | 1)             // illegal negative size
OctetString (5 | 0)              // illegal ordering
OctetString (1 | 1..10)          // illegal overlapping
```

3.2. Pointer

The Pointer base type represents values that reference class instances, attributes of class instances, or arbitrary identities. The only values of the Pointer type that can be present in a module can refer to identities. They are denoted as identifiers of the concerned identities.

When defining a type derived (directly or indirectly) from the Pointer base type, the values may be restricted to a specific class, attribute or identity, and all (directly or indirectly) derived items thereof by appending the identifier of the appropriate construct enclosed in parenthesis.

Value Examples:

```
null                             // legal identity name
snmpUDPDomain                    // legal identity name
```

Restriction Examples:

```
Pointer (snmpTransportDomain) // legal restriction
```

3.3. ObjectIdentifier

The ObjectIdentifier base type represents administratively assigned names for use with SNMP and COPS-PR. This type SHOULD NOT be used in protocol independent SMIng modules. It is meant to be used in SNMP and COPS-PR mappings of attributes of type Pointer (Section 3.2).

Values of this type may be denoted as a sequence of numerical non-negative sub-identifier values in which each MUST NOT exceed 2^32-1 (4294967295). Sub-identifiers may be denoted in decimal or `0x'-prefixed hexadecimal. They are separated by single dots and without any intermediate white space. Alternatively (and preferred in most cases), the first element may be a previously defined or imported lower-case identifier, representing a static object identifier prefix.

Although the number of sub-identifiers in SMIng object identifiers is not limited, module designers should realize that there may be implementations that stick with the SMIv1/v2 limit of 128 sub-identifiers.

Object identifier derived types cannot be restricted in any way.

Value Examples:

```
1.3.6.1                     // legal numerical oid
mib-2.1                     // legal oid with identifier prefix
internet.4.1.0x0627.0x01    // legal oid with hex subids
iso.-1                      // illegal negative subid
iso.org.6                   // illegal non-heading identifier
IF-MIB::ifNumber.0          // legal fully qualified instance oid
```

3.4. Integer32

The Integer32 base type represents integer values between -2^31 (-2147483648) and 2^31-1 (2147483647).

Values of type Integer32 may be denoted as decimal or hexadecimal numbers, where only decimal numbers can be negative. Decimal numbers other than zero MUST NOT have leading zero digits. Hexadecimal numbers are prefixed by `0x' and MUST have an even number of at least two hexadecimal digits, where letters MAY be upper-case, but lower-case characters are RECOMMENDED.

When defining a type derived (directly or indirectly) from the Integer32 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters, and the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..', and an upper bound. Each value can be given in decimal or `0x'-prefixed hexadecimal notation. Hexadecimal numbers must have an even number of at least two digits. If multiple values or ranges are given they all MUST be disjoint and MUST be in ascending order. If a value restriction is applied to an already restricted type, the new restriction MUST be equal or more limiting, that is raising the lower

bounds, reducing the upper bounds, removing explicit values or ranges, or splitting ranges into multiple ranges with intermediate gaps.

Value Examples:

```
   015                          // illegal leading zero
   -123                         // legal negative value
   - 1                          // illegal intermediate space
   0xabc                        // illegal hexadecimal value length
   -0xff                        // illegal sign on hex value
   0x80000000                   // illegal value, too large
   0xf00f                       // legal hexadecimal value
```

Restriction Examples:

```
   Integer32 (0 | 5..10)        // legal range spec
   Integer32 (5..10 | 2..3)     // illegal ordering
   Integer32 (4..8 | 5..10)     // illegal overlapping
```

3.5. Integer64

The Integer64 base type represents integer values between -2^63 (-9223372036854775808) and 2^63-1 (9223372036854775807).

Values of type Integer64 may be denoted as decimal or hexadecimal numbers, where only decimal numbers can be negative. Decimal numbers other than zero MUST NOT have leading zero digits. Hexadecimal numbers are prefixed by `0x' and MUST have an even number of hexadecimal digits, where letters MAY be upper-case, but lower-case characters are RECOMMENDED.

When defining a type derived (directly or indirectly) from the Integer64 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters, with the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..', and an upper bound. Each value can be given in decimal or `0x'-prefixed hexadecimal notation. Hexadecimal numbers must have an even number of at least two digits. If multiple values or ranges are given, they all MUST be disjoint and MUST be in ascending order. If a value restriction is applied to an already restricted type, the new restriction MUST be equal or more limiting, that is raising the lower bounds, reducing the upper bounds, removing explicit values or ranges, or splitting ranges into multiple ranges with intermediate gaps.

Value Examples:

```
   015                          // illegal leading zero
   -123                         // legal negative value
   - 1                          // illegal intermediate space
   0xabc                        // illegal hexadecimal value length
   -0xff                        // illegal sign on hex value
   0x80000000                   // legal value
```

Restriction Examples:

```
   Integer64 (0 | 5..10)        // legal range spec
   Integer64 (5..10 | 2..3)     // illegal ordering
   Integer64 (4..8 | 5..10)     // illegal overlapping
```

3.6. Unsigned32

The Unsigned32 base type represents positive integer values between 0 and 2^32-1 (4294967295).

Values of type Unsigned32 may be denoted as decimal or hexadecimal numbers. Decimal numbers other than zero MUST NOT have leading zero digits. Hexadecimal numbers are prefixed by `0x' and MUST have an even number of hexadecimal digits, where letters MAY be upper-case, but lower-case characters are RECOMMENDED.

When defining a type derived (directly or indirectly) from the Unsigned32 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters, with the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..', and an upper bound. Each value can be given in decimal or `0x'-prefixed hexadecimal notation. Hexadecimal numbers must have an even number of at least two digits. If multiple values or ranges are given, they all MUST be disjoint and MUST be in ascending order. If a value restriction is applied to an already restricted type, the new restriction MUST be equal or more limiting, that is raising the lower bounds, reducing the upper bounds, removing explicit values or ranges, or splitting ranges into multiple ranges with intermediate gaps.

Value Examples:

```
   015                          // illegal leading zero
   -123                         // illegal negative value
   0xabc                        // illegal hexadecimal value length
   0x80000000                   // legal hexadecimal value
   0x8080000000                 // illegal value, too large
```

Restriction Examples:

```
Unsigned32 (0 | 5..10)       // legal range spec
Unsigned32 (5..10 | 2..3)    // illegal ordering
Unsigned32 (4..8 | 5..10)    // illegal overlapping
```

3.7. Unsigned64

The Unsigned64 base type represents positive integer values between 0 and 2^64-1 (18446744073709551615).

Values of type Unsigned64 may be denoted as decimal or hexadecimal numbers. Decimal numbers other than zero MUST NOT have leading zero digits. Hexadecimal numbers are prefixed by `0x' and MUST have an even number of hexadecimal digits, where letters MAY be upper-case, but lower-case characters are RECOMMENDED.

When defining a type derived (directly or indirectly) from the Unsigned64 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters, with the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..', and an upper bound. Each value can be given in decimal or `0x'-prefixed hexadecimal notation. Hexadecimal numbers must have an even number of at least two digits. If multiple values or ranges are given, they all MUST be disjoint and MUST be in ascending order. If a value restriction is applied to an already restricted type, the new restriction MUST be equal or more limiting, that is raising the lower bounds, reducing the upper bounds, removing explicit values or ranges, or splitting ranges into multiple ranges with intermediate gaps.

Value Examples:

```
015                          // illegal leading zero
-123                         // illegal negative value
0xabc                        // illegal hexadecimal value length
0x8080000000                 // legal hexadecimal value
```

Restriction Examples:

```
Unsigned64 (1..10000000000) // legal range spec
Unsigned64 (5..10 | 2..3)   // illegal ordering
```

3.8. Float32

The Float32 base type represents floating point values of single precision as described by [IEEE754].

Values of type Float32 may be denoted as a decimal fraction with an optional exponent, as known from many programming languages. See the grammar rule `floatValue' of Appendix B for the detailed syntax. Special values are `snan' (signalling Not-a-Number), `qnan' (quiet Not-a-Number), `neginf' (negative infinity), and `posinf' (positive infinity). Note that -0.0 and +0.0 are different floating point values. 0.0 is equal to +0.0.

When defining a type derived (directly or indirectly) from the Float32 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters, with the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..', and an upper bound. If multiple values or ranges are given, they all MUST be disjoint and MUST be in ascending order. If a value restriction is applied to an already restricted type, the new restriction MUST be equal or more limiting, that is raising the lower bounds, reducing the upper bounds, removing explicit values or ranges, or splitting ranges into multiple ranges with intermediate gaps. The special values `snan', `qnan', `neginf', and `posinf' must be explicitly listed in restrictions if they shall be included, where `snan' and `qnan' cannot be used in ranges.

Note that encoding is not subject to this specification. It has to be described by protocols that transport objects of type Float32. Note also that most floating point encodings disallow the representation of many values that can be written as decimal fractions as used in SMIng for human readability. Therefore, explicit values in floating point type restrictions should be handled with care.

Value Examples:

```
00.1                        // illegal leading zero
3.1415                      // legal value
-2.5E+3                     // legal negative exponential value
```

Restriction Examples:

```
Float32 (-1.0..1.0)         // legal range spec
Float32 (1 | 3.3 | 5)       // legal, probably unrepresentable 3.3
Float32 (neginf..-0.0)      // legal range spec
Float32 (-10.0..10.0 | 0)   // illegal overlapping
```

3.9. Float64

The Float64 base type represents floating point values of double precision as described by [IEEE754].

Values of type Float64 may be denoted as a decimal fraction with an optional exponent, as known from many programming languages. See the grammar rule `floatValue' of Appendix B for the detailed syntax. Special values are `snan' (signalling Not-a-Number), `qnan' (quiet Not-a-Number), `neginf' (negative infinity), and `posinf' (positive infinity). Note that -0.0 and +0.0 are different floating point values. 0.0 is equal to +0.0.

When defining a type derived (directly or indirectly) from the Float64 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters, with the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..', and an upper bound. If multiple values or ranges are given, they all MUST be disjoint and MUST be in ascending order. If a value restriction is applied to an already restricted type, the new restriction MUST be equal or more limiting, that is raising the lower bounds, reducing the upper bounds, removing explicit values or ranges, or splitting ranges into multiple ranges with intermediate gaps. The special values `snan', `qnan', `neginf', and `posinf' must be explicitly listed in restrictions if they shall be included, where `snan' and `qnan' cannot be used in ranges.

Note that encoding is not subject to this specification. It has to be described by protocols that transport objects of type Float64. Note also that most floating point encodings disallow the representation of many values that can be written as decimal fractions as used in SMIng for human readability. Therefore, explicit values in floating point type restrictions should be handled with care.

Value Examples:

```
00.1                        // illegal leading zero
3.1415                      // legal value
-2.5E+3                     // legal negative exponential value
```

Restriction Examples:

```
Float64 (-1.0..1.0)         // legal range spec
Float64 (1 | 3.3 | 5)       // legal, probably unrepresentable 3.3
Float64 (neginf..-0.0)      // legal range spec
Float64 (-10.0..10.0 | 0)   // illegal overlapping
```

3.10. Float128

The Float128 base type represents floating point values of quadruple precision as described by [IEEE754].

Values of type Float128 may be denoted as a decimal fraction with an optional exponent, as known from many programming languages. See the grammar rule `floatValue' of Appendix B for the detailed syntax. Special values are `snan' (signalling Not-a-Number), `qnan' (quiet Not-a-Number), `neginf' (negative infinity), and `posinf' (positive infinity). Note that -0.0 and +0.0 are different floating point values. 0.0 is equal to +0.0.

When defining a type derived (directly or indirectly) from the Float128 base type, the set of possible values may be restricted by appending a list of ranges or explicit values, separated by pipe `|' characters, with the whole list enclosed in parenthesis. A range consists of a lower bound, two consecutive dots `..', and an upper bound. If multiple values or ranges are given, they all MUST be disjoint and MUST be in ascending order. If a value restriction is applied to an already restricted type, the new restriction MUST be equal or more limiting, that is raising the lower bounds, reducing the upper bounds, removing explicit values or ranges, or splitting ranges into multiple ranges with intermediate gaps. The special values `snan', `qnan', `neginf', and `posinf' must be explicitly listed in restrictions if they shall be included, where `snan' and `qnan' cannot be used in ranges.

Note that encoding is not subject to this specification. It has to be described by protocols that transport objects of type Float128. Note also that most floating point encodings disallow the representation of many values that can be written as decimal fractions as used in SMIng for human readability. Therefore, explicit values in floating point type restrictions should be handled with care.

Value Examples:

```
00.1                        // illegal leading zero
3.1415                      // legal value
-2.5E+3                     // legal negative exponential value
```

Restriction Examples:

```
Float128 (-1.0..1.0)         // legal range spec
Float128 (1 | 3.3 | 5)       // legal, probably unrepresentable 3.3
Float128 (neginf..-0.0)      // legal range spec
Float128 (-10.0..10.0 | 0)   // illegal overlapping
```

3.11. Enumeration

The Enumeration base type represents values from a set of integers in the range between -2^31 (-2147483648) and 2^31-1 (2147483647), where each value has an assigned name. The list of those named numbers has to be comma-separated, enclosed in parenthesis, and appended to the `Enumeration' keyword. Each named number is denoted by its lower-case identifier followed by the assigned integer value, denoted as a decimal or `0x'-prefixed hexadecimal number, enclosed in parenthesis. Hexadecimal numbers must have an even number of at least two digits. Every name and every number in an enumeration type MUST be unique. It is RECOMMENDED that values be positive, start at 1, and be numbered contiguously. All named numbers MUST be given in ascending order.

Values of enumeration types may be denoted as decimal or `0x'-prefixed hexadecimal numbers or preferably as their assigned names. Hexadecimal numbers must have an even number of at least two digits.

When types are derived (directly or indirectly) from an enumeration type, the set of named numbers may be equal or restricted by removing one or more named numbers, but no named numbers may be added or changed regarding its name, value, or both.

Type and Value Examples:

```
Enumeration (up(1), down(2), testing(3))
Enumeration (down(2), up(1)) // illegal order

0                            // legal (though not recommended) value
up                           // legal value given by name
2                            // legal value given by number
```

3.12. Bits

The Bits base type represents bit sets. That is, a Bits value is a set of flags identified by small integer numbers starting at 0. Each bit number has an assigned name. The list of those named numbers has to be comma-separated, enclosed in parenthesis, and appended to the `Bits' keyword. Each named number is denoted by its lower-case identifier followed by the assigned integer value, denoted as a decimal or `0x'-prefixed hexadecimal number, enclosed in parenthesis. Hexadecimal numbers must have an even number of at least two digits. Every name and every number in a bits type MUST be unique. It is RECOMMENDED that numbers start at 0 and be numbered contiguously. Negative numbers are forbidden. All named numbers MUST be given in ascending order.

Values of bits types may be denoted as a comma-separated list of decimal or `0x'-prefixed hexadecimal numbers or preferably their assigned names enclosed in parenthesis. Hexadecimal numbers must have an even number of at least two digits. There MUST NOT be any element (by name or number) listed more than once. Elements MUST be listed in ascending order.

When defining a type derived (directly or indirectly) from a bits type, the set of named numbers may be restricted by removing one or more named numbers, but no named numbers may be added or changed regarding its name, value, or both.

Type and Value Examples:

```
Bits (readable(0), writable(1), executable(2))
Bits (writable(1), readable(0) // illegal order

()                            // legal empty value
(readable, writable, 2)       // legal value
(0, readable, executable)     // illegal, readable(0) appears twice
(writable, 4)                 // illegal, element 4 out of range
```

3.13. Display Formats

Attribute and type definitions allow the specification of a format to be used when a value of that attribute or an attribute of that type is displayed. Format specifications are represented as textual data.

When the attribute or type has an underlying base type of Integer32, Integer64, Unsigned32, or Unsigned64, the format consists of an integer-format specification containing two parts. The first part is a single character suggesting a display format, either: `x' for hexadecimal, `d' for decimal, `o' for octal, or `b' for binary. For all types, when rendering the value, leading zeros are omitted, and for negative values, a minus sign is rendered immediately before the digits. The second part is always omitted for `x', `o', and `b', and need not be present for `d'. If present, the second part starts with a hyphen and is followed by a decimal number, which defines the implied decimal point when rendering the value. For example `d-2' suggests that a value of 1234 be rendered as `12.34'.

When the attribute or type has an underlying base type of OctetString, the format consists of one or more octet-format specifications. Each specification consists of five parts, with each part using and removing zero or more of the next octets from the

value and producing the next zero or more characters to be displayed. The octets within the value are processed in order of significance, most significant first.

The five parts of a octet-format specification are:

1. The (optional) repeat indicator. If present, this part is a `*', and indicates that the current octet of the value is to be used as the repeat count. The repeat count is an unsigned integer (which may be zero) specifying how many times the remainder of this octet-format specification should be successively applied. If the repeat indicator is not present, the repeat count is one.

2. The octet length: one or more decimal digits specifying the number of octets of the value to be used and formatted by this octet-specification. Note that the octet length can be zero. If less than this number of octets remain in the value, then the lesser number of octets are used.

3. The display format, either: `x' for hexadecimal, `d' for decimal, `o' for octal, `a' for ASCII, or `t' for UTF-8 [RFC3629]. If the octet length part is greater than one, and the display format part refers to a numeric format, then network byte-ordering (big-endian encoding) is used to interpret the octets in the value. The octets processed by the `t' display format do not necessarily form an integral number of UTF-8 characters. Trailing octets which do not form a valid UTF-8 encoded character are discarded.

4. The (optional) display separator character. If present, this part is a single character produced for display after each application of this octet-specification; however, this character is not produced for display if it would be immediately followed by the display of the repeat terminator character for this octet specification. This character can be any character other than a decimal digit and a `*'.

5. The (optional) repeat terminator character, which can be present only if the display separator character is present and this octet specification begins with a repeat indicator. If present, this part is a single character produced after all the zero or more repeated applications (as given by the repeat count) of this octet specification. This character can be any character other than a decimal digit and a `*'.

Output of a display separator character or a repeat terminator character is suppressed if it would occur as the last character of the display.

If the octets of the value are exhausted before all the octet format specifications have been used, then the excess specifications are ignored. If additional octets remain in the value after interpreting all the octet format specifications, then the last octet format specification is re-interpreted to process the additional octets, until no octets remain in the value.

Note that for some types, no format specifications are defined. For derived types and attributes that are based on such types, format specifications SHOULD be omitted. Implementations MUST ignore format specifications they cannot interpret. Also note that the SMIng grammar (Appendix B) does not specify the syntax of format specifications.

Display Format Examples:

Base Type	Format	Example Value	Rendered Value
OctetString	255a	"Hello World."	Hello World.
OctetString	1x:	"Hello!"	48:65:6c:6c:6f:21
OctetString	1d:1d:1d.1d,1a1d:1d	0x0d1e0f002d0400	13:30:15.0,-4:0
OctetString	1d.1d.1d.1d/2d	0x0a0000010400	10.0.0.1/1024
OctetString	*1x:/1x:	0x02aabbccddee	aa:bb/cc:dd:ee
Integer32	d-2	1234	12.34

4. The SMIng File Structure

The topmost container of SMIng information is a file. An SMIng file may contain zero, one or more modules. It is RECOMMENDED that modules be stored into separate files by their module names, where possible. However, for dedicated purposes, it may be reasonable to collect several modules in a single file.

The top level SMIng construct is the `module' statement (Section 5) that defines a single SMIng module. A module contains a sequence of sections in an obligatory order with different kinds of definitions. Whether these sections contain statements or remain empty mainly depends on the purpose of the module.

4.1. Comments

Comments can be included at any position in an SMIng file, except between the characters of a single token like those of a quoted string. However, it is RECOMMENDED that all substantive descriptions be placed within an appropriate description clause, so that the information is available to SMIng parsers.

Comments commence with a pair of adjacent slashes `//' and end at the end of the line.

4.2. Textual Data

Some statements, namely `organization', `contact', `description', `reference', `abnf', `format', and `units', get a textual argument. This text, as well as representations of OctetString values, have to be enclosed in double quotes. They may contain arbitrary characters with the following exceptional encoding rules:

A backslash character introduces a special character, which depends on the character that immediately follows the backslash:

```
\n      new line
\t      a tab character
\"      a double quote
\\      a single backslash
```

If the text contains a line break followed by whitespace which is used to indent the text according to the layout in the SMIng file, this prefixing whitespace is stripped from the text.

4.3. Statements and Arguments

SMIng has a very small set of basic grammar rules based on the concept of statements. Each statement starts with a lower-case keyword identifying the statement, followed by a number (possibly zero) of arguments. An argument may be quoted text, an identifier, a value of any base type, a list of identifiers enclosed in parenthesis `( )', or a statement block enclosed in curly braces `{ }'. Since statement blocks are valid arguments, it is possible to nest statement sequences. Each statement is terminated by a semicolon `;'.

The core set of statements may be extended using the SMIng `extension' statement. See Sections 6 and 11 for details.

At places where a statement is expected, but an unknown lower-case word is read, those statements MUST be skipped up to the proper semicolon, including nested statement blocks.

5. The module Statement

The `module' statement is used as a container of all definitions of a single SMIng module. It gets two arguments: an upper-case module name and a statement block that contains mandatory and optional statements and sections of statements in an obligatory order:

```
module <MODULE-NAME> {

    <optional import statements>
    <organization statement>
    <contact statement>
    <description statement>
    <optional reference statement>
    <at least one revision statement>

    <optional extension statements>

    <optional typedef statements>

    <optional identity statements>

    <optional class statements>

};
```

The optional `import' statements (Section 5.1) are followed by the mandatory `organization' (Section 5.2), `contact' (Section 5.3), and `description' (Section 5.4) statements and the optional `reference' statement (Section 5.5), which in turn are followed by at least one mandatory `revision' statement (Section 5.6). The part up to this point defines the module's meta information, i.e., information that describes the whole module but does not define any items used by applications in the first instance. This part of a module is followed by its main definitions, namely SMIng extensions (Section 6), derived types (Section 7), identities (Section 8), and classes (Section 9).

See the `moduleStatement' rule of the SMIng grammar (Appendix B) for the formal syntax of the `module' statement.

5.1. The module's import Statement

The optional module's `import' statement is used to import identifiers from external modules into the local module's namespace. It gets two arguments: the name of the external module and a comma-separated list of one or more identifiers to be imported enclosed in parenthesis.

Multiple `import' statements for the same module but with disjoint lists of identifiers are allowed, though NOT RECOMMENDED. The same identifier from the same module MUST NOT be imported multiple times. To import identifiers with the same name from different modules might be necessary and is allowed. To distinguish

them in the local module, they have to be referred by qualified names. Importing identifiers not used in the local module is NOT RECOMMENDED.

See the `importStatement' rule of the SMIng grammar (Appendix B) for the formal syntax of the `import' statement.

5.2. The module's organization Statement

The module's `organization' statement, which must be present, gets one argument which is used to specify a textual description of the organization(s) under whose auspices this module was developed.

5.3. The module's contact Statement

The module's `contact' statement, which must be present, gets one argument which is used to specify the name, postal address, telephone number, and electronic mail address of the person to whom technical queries concerning this module should be sent.

5.4. The module's description Statement

The module's `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of the contents of this module.

5.5. The module's reference Statement

The module's `reference' statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related management information, or some other document which provides additional information relevant to this module.

5.6. The module's revision Statement

The module's `revision' statement is repeatedly used to specify the editorial revisions of the module, including the initial revision. It gets one argument which is a statement block that holds detailed information in an obligatory order. A module MUST have at least one initial `revision' statement. For every editorial change, a new one MUST be added in front of the revisions sequence, so that all revisions are in reverse chronological order.

See the `revisionStatement' rule of the SMIng grammar (Appendix B) for the formal syntax of the `revision' statement.

5.6.1. The revision's date Statement

The revision's `date' statement, which must be present, gets one argument which is used to specify the date and time of the revision in the format `YYYY-MM-DD HH:MM' or `YYYY-MM-DD' which implies the time `00:00'. The time is always given in UTC.

See the `date' rule of the SMIng grammar (Appendix B) for the formal syntax of the revision's `date' statement.

5.6.2. The revision's description Statement

The revision's `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of the revision.

5.7. Usage Example

Consider how a skeletal module might be constructed:

```
module ACME-MIB {

  import NMRG-SMING (DisplayString);

  organization
             "IRTF Network Management Research Group (NMRG)";

  contact    "IRTF Network Management Research Group (NMRG)
              http://www.ibr.cs.tu-bs.de/projects/nmrg/

              Joe L. User

              ACME, Inc.
              42 Anywhere Drive
              Nowhere, CA 95134
              USA

              Phone: +1 800 555 0815
              EMail: joe@acme.example.com";

  description
             "The module for entities implementing the ACME protocol.

              Copyright (C) The Internet Society (2004).
              All Rights Reserved.
              This version of this MIB module is part of RFC 3780,
              see the RFC itself for legal notices.";
```

```
  revision {
    date            "2003-12-16";
    description
            "Initial revision, published as RFC 3780.";
  };

  // ... further definitions ...

}; // end of module ACME-MIB.
```

6. The extension Statement

The `extension' statement defines new statements to be used in the local module following this extension statement definition or in external modules that may import this extension statement definition. The `extension' statement gets two arguments: a lower-case extension statement identifier and a statement block that holds detailed extension information in an obligatory order.

Extension statement identifiers SHOULD NOT contain any upper-case characters.

Note that the SMIng extension feature does not allow the formal specification of the context, or argument syntax and semantics of an extension. Its only purpose is to declare the existence of an extension and to allow a unique reference to an extension. See Section 11 for detailed information on extensions and [RFC3781] for mappings of SMIng definitions to SNMP, which is formally defined as an extension.

See the `extensionStatement' rule of the SMIng grammar (Appendix B) for the formal syntax of the `extension' statement.

6.1. The extension's status Statement

The extension's `status' statement, which must be present, gets one argument which is used to specify whether this extension definition is current or historic. The value `current' means that the definition is current and valid. The value `obsolete' means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value `deprecated' also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/ existing implementations.

6.2. The extension's description Statement

The extension's `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of the extension statement.

It is RECOMMENDED that information on the extension's context, its semantics, and implementation conditions be included. See also Section 11.

6.3. The extension's reference Statement

The extension's `reference' statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related extension definitions, or some other document which provides additional information relevant to this extension.

6.4. The extension's abnf Statement

The extension's `abnf' statement, which need not be present, gets one argument which is used to specify a formal ABNF [RFC2234] grammar definition of the extension. This grammar can reference rule names from the core SMIng grammar (Appendix B).

Note that the `abnf' statement should contain only pure ABNF and no additional text, though comments prefixed by a semicolon are allowed but should probably be moved to the description statement. Note that double quotes within the ABNF grammar have to be represented as `\"' according to Section 4.2.

6.5. Usage Example

```
extension severity {
  status  current;
  description
         "The optional severity extension statement can only
          be applied to the statement block of an SMIng class'
          event definition. If it is present it denotes the
          severity level of the event in a range from 0
          (emergency) to 7 (debug).";
  abnf
         "severityStatement = severityKeyword sep number optsep \";\"
          severityKeyword   = \"severity\"";
};
```

7. The typedef Statement

The `typedef' statement defines new data types to be used in the local module or in external modules. It gets two arguments: an upper-case type identifier and a statement block that holds detailed type information in an obligatory order.

Type identifiers SHOULD NOT consist of all upper-case characters and SHOULD NOT contain hyphens.

See the `typedefStatement' rule of the SMIng grammar (Appendix B) for the formal syntax of the `typedef' statement.

7.1. The typedef's type Statement

The typedef's `type' statement, which must be present, gets one argument which is used to specify the type from which this type is derived. Optionally, type restrictions may be applied to the new type by appending subtyping information according to the rules of the base type. See Section 3 for SMIng base types and their type restrictions.

7.2. The typedef's default Statement

The typedef's `default' statement, which need not be present, gets one argument which is used to specify an acceptable default value for attributes of this type. A default value may be used when an attribute instance is created. That is, the value is a "hint" to implementors.

The value of the `default' statement must, of course, correspond to the (probably restricted) type specified in the typedef's `type' statement.

The default value of a type may be overwritten by a default value of an attribute of this type.

Note that for some types, default values make no sense.

7.3. The typedef's format Statement

The typedef's `format' statement, which need not be present, gets one argument which is used to give a hint as to how the value of an instance of an attribute of this type might be displayed. See Section 3.13 for a description of format specifications.

If no format is specified, it is inherited from the type given in the `type' statement. On the other hand, the format specification of a type may be semantically refined by a format specification of an attribute of this type.

7.4. The typedef's units Statement

The typedef's `units' statement, which need not be present, gets one argument which is used to specify a textual definition of the units associated with attributes of this type.

If no units are specified, they are inherited from the type given in the `type' statement. On the other hand, the units specification of a type may be semantically refined by a units specification of an attribute of this type.

The units specification has to be appropriate for values displayed according to the typedef's format specification, if present. For example, if the type defines frequency values of type Unsigned64 measured in thousands of Hertz, the format specification should be `d-3' and the units specification should be `Hertz' or `Hz'. If the format specification would be omitted, the units specification should be `Milli-Hertz' or `mHz'. Authors of SMIng modules should pay attention to keep format and units specifications in sync. Application implementors MUST NOT implement units specifications without implementing format specifications.

7.5. The typedef's status Statement

The typedef's `status' statement, which must be present, gets one argument which is used to specify whether this type definition is current or historic. The value `current' means that the definition is current and valid. The value `obsolete' means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value `deprecated' also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

Derived types SHOULD NOT be defined as `current' if their underlying type is `deprecated' or `obsolete'. Similarly, they SHOULD NOT be defined as `deprecated' if their underlying type is `obsolete'. Nevertheless, subsequent revisions of the underlying type cannot be avoided, but SHOULD be taken into account in subsequent revisions of the local module.

7.6. The typedef's description Statement

The typedef's `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of the newly defined type.

It is RECOMMENDED that all semantic definitions necessary for implementation, and to embody any information which would otherwise be communicated in any commentary annotations associated with this type definition be included.

7.7. The typedef's reference Statement

The typedef's `reference' statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related type definitions, or some other document which provides additional information relevant to this type definition.

7.8. Usage Examples

```
typedef RptrOperStatus {
  type            Enumeration (other(1), ok(2), rptrFailure(3),
                               groupFailure(4), portFailure(5),
                               generalFailure(6));
  default         other;       // undefined by default.
  status          deprecated;
  description
          "A type to indicate the operational state
           of a repeater.";
  reference
          "[IEEE 802.3 Mgt], 30.4.1.1.5, aRepeaterHealthState.";
};

typedef SnmpTransportDomain {
  type            Pointer (snmpTransportDomain);
  status          current;
  description
          "A pointer to an SNMP transport domain identity.";
};

typedef DateAndTime {
  type            OctetString (8 | 11);
  format          "2d-1d-1d,1d:1d:1d.1d,1a1d:1d";
  status          current;
  description
          "A date-time specification.
           ...
```

```
            Note that if only local time is known, then timezone
            information (fields 8-10) is not present.";
    reference
           "RFC 2579, SNMPv2-TC.DateAndTime.";
  };

  typedef Frequency {
    type            Unsigned64;
    format          "d-3"
    units           "Hertz";
    status          current;
    description
           "A wide-range frequency specification measured
            in thousands of Hertz.";
  };
```

8. The identity Statement

The `identity' statement is used to define a new abstract and untyped identity. Its only purpose is to denote its name, semantics, and existence. An identity can be defined either from scratch or derived from a parent identity. The `identity' statement gets the following two arguments: The first argument is a lower-case identity identifier. The second argument is a statement block that holds detailed identity information in an obligatory order.

See the `identityStatement' rule of the SMIng grammar (Appendix B) for the formal syntax of the `identity' statement.

8.1. The identity's parent Statement

The identity's `parent' statement must be present for a derived identity and must be absent for an identity defined from scratch. It gets one argument which is used to specify the parent identity from which this identity shall be derived.

8.2. The identity's status Statement

The identity's `status' statement, which must be present, gets one argument which is used to specify whether this identity definition is current or historic. The value `current' means that the definition is current and valid. The value `obsolete' means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value `deprecated' also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

Derived identities SHOULD NOT be defined as `current' if their parent identity is `deprecated' or `obsolete'. Similarly, they SHOULD NOT be defined as `deprecated' if their parent identity is `obsolete'. Nevertheless, subsequent revisions of the parent identity cannot be avoided, but SHOULD be taken into account in subsequent revisions of the local module.

8.3. The identity' description Statement

The identity's `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of the newly defined identity.

It is RECOMMENDED that all semantic definitions necessary for implementation, and to embody any information which would otherwise be communicated in any commentary annotations associated with this identity definition be included.

8.4. The identity's reference Statement

The identity's `reference' statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related identity definitions, or some other document which provides additional information relevant to this identity definition.

8.5. Usage Examples

```
identity null {
  status  current;
  description
          "An identity used to represent null pointer values.";
};

identity snmpTransportDomain {
  status  current;
  description
          "A generic SNMP transport domain identity.";
};

identity snmpUDPDomain {
  parent  snmpTransportDomain;
  status  current;
  description
          "The SNMP over UDP transport domain.";
};
```

9. The class Statement

The `class' statement is used to define a new class that represents a container of related attributes and events (Section 9.2, Section 9.4). A class can be defined either from scratch or derived from a parent class. A derived class inherits all attributes and events of the parent class and can be extended by additional attributes and events.

The `class' statement gets the following two arguments: The first argument is an upper-case class identifier. The second argument is a statement block that holds detailed class information in an obligatory order.

See the `classStatement' rule of the SMIng grammar (Appendix B) for the formal syntax of the `class' statement.

9.1. The class' extends Statement

The class' `extends' statement must be present for a class derived from a parent class and must be absent for a class defined from scratch. It gets one argument which is used to specify the parent class from which this class shall be derived.

9.2. The class' attribute Statement

The class' `attribute' statement, which can be present zero, one or multiple times, gets two arguments: the attribute name and a statement block that holds detailed attribute information in an obligatory order.

9.2.1. The attribute's type Statement

The attribute's `type' statement must be present. It gets at least one argument which is used to specify the type of the attribute: either a type name or a class name. In case of a type name, it may be restricted by a second argument according to the restriction rules described in Section 3.

9.2.2. The attribute's access Statement

The attribute's `access' statement must be present for attributes typed by a base type or derived type, and must be absent for attributes typed by a class. It gets one argument which is used to specify whether it makes sense to read and/or write an instance of the attribute, or to include its value in an event. This is the maximal level of access for the attribute. This maximal level of access is independent of any administrative authorization policy.

The value `readwrite' indicates that read and write access makes sense. The value `readonly' indicates that read access makes sense, but write access is never possible. The value `eventonly' indicates an object which is accessible only via an event.

These values are ordered, from least to greatest access level: `eventonly', `readonly', `readwrite'.

9.2.3. The attribute's default Statement

The attribute's `default' statement need not be present for attributes typed by a base type or derived type, and must be absent for attributes typed by a class. It gets one argument which is used to specify an acceptable default value for this attribute. A default value may be used when an attribute instance is created. That is, the value is a "hint" to implementors.

The value of the `default' statement must, of course, correspond to the (probably restricted) type specified in the attribute's `type' statement.

The attribute's default value overrides the default value of the underlying type definition if both are present.

9.2.4. The attribute's format Statement

The attribute's `format' statement need not be present for attributes typed by a base type or derived type, and must be absent for attributes typed by a class. It gets one argument which is used to give a hint as to how the value of an instance of this attribute might be displayed. See Section 3.13 for a description of format specifications.

The attribute's format specification overrides the format specification of the underlying type definition if both are present.

9.2.5. The attribute's units Statement

The attribute's `units' statement need not be present for attributes typed by a base type or derived type, and must be absent for attributes typed by a class. It gets one argument which is used to specify a textual definition of the units associated with this attribute.

The attribute's units specification overrides the units specification of the underlying type definition if both are present.

The units specification has to be appropriate for values displayed according to the attribute's format specification if present. For example, if the attribute represents a frequency value of type Unsigned64 measured in thousands of Hertz, the format specification should be `d-3' and the units specification should be `Hertz' or `Hz'. If the format specification would be omitted, the units specification should be `Milli-Hertz' or `mHz'. Authors of SMIng modules should pay attention to keep format and units specifications of type and attribute definitions in sync. Application implementors MUST NOT implement units specifications without implementing format specifications.

9.2.6. The attribute's status Statement

The attribute's `status' statement must be present. It gets one argument which is used to specify whether this attribute definition is current or historic. The value `current' means that the definition is current and valid. The value `obsolete' means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value `deprecated' also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/ existing implementations.

Attributes SHOULD NOT be defined as `current' if their type or their containing class is `deprecated' or `obsolete'. Similarly, they SHOULD NOT be defined as `deprecated' if their type or their containing class is `obsolete'. Nevertheless, subsequent revisions of used type definition cannot be avoided, but SHOULD be taken into account in subsequent revisions of the local module.

9.2.7. The attribute's description Statement

The attribute's `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of this attribute.

It is RECOMMENDED that all semantic definitions necessary for the implementation of this attribute be included.

9.2.8. The attribute's reference Statement

The attribute's `reference' statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related attribute definitions, or some other document which provides additional information relevant to this attribute definition.

9.3. The class' unique Statement

The class' `unique' statement, which need not be present, gets one argument that specifies a comma-separated list of attributes of this class, enclosed in parenthesis. If present, this list of attributes makes up a unique identification of all possible instances of this class. It can be used as a unique key in underlying protocols.

If the list is empty, the class should be regarded as a scalar class with only a single instance.

If the `unique' statement is not present, the class is not meant to be instantiated directly, but to be contained in other classes or the parent class of other refining classes.

If present, the attribute list MUST NOT contain any attribute more than once and the attributes should be ordered where appropriate so that the attributes that are most significant in most situations appear first.

9.4. The class' event Statement

The class' `event' statement is used to define an event related to an instance of this class that can occur asynchronously. It gets two arguments: a lower-case event identifier and a statement block that holds detailed information in an obligatory order.

See the `eventStatement' rule of the SMIng grammar (Appendix B) for the formal syntax of the `event' statement.

9.4.1. The event's status Statement

The event's `status' statement, which must be present, gets one argument which is used to specify whether this event definition is current or historic. The value `current' means that the definition is current and valid. The value `obsolete' means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value `deprecated' also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

9.4.2. The event's description Statement

The event's `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of this event.

It is RECOMMENDED that all semantic definitions necessary for the implementation of this event be included. In particular, which instance of the class is associated with an event of this type SHOULD be documented.

9.4.3. The event's reference Statement

The event's `reference' statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related event definitions, or some other document which provides additional information relevant to this event definition.

9.5. The class' status Statement

The class' `status' statement, which must be present, gets one argument which is used to specify whether this class definition is current or historic. The value `current' means that the definition is current and valid. The value `obsolete' means the definition is obsolete and should not be implemented and/or can be removed if previously implemented. While the value `deprecated' also indicates an obsolete definition, it permits new/continued implementation in order to foster interoperability with older/existing implementations.

Derived classes SHOULD NOT be defined as `current' if their parent class is `deprecated' or `obsolete'. Similarly, they SHOULD NOT be defined as `deprecated' if their parent class is `obsolete'. Nevertheless, subsequent revisions of the parent class cannot be avoided, but SHOULD be taken into account in subsequent revisions of the local module.

9.6. The class' description Statement

The class' `description' statement, which must be present, gets one argument which is used to specify a high-level textual description of the newly defined class.

It is RECOMMENDED that all semantic definitions necessary for implementation, and to embody any information which would otherwise be communicated in any commentary annotations associated with this class definition be included.

9.7. The class' reference Statement

The class' `reference' statement, which need not be present, gets one argument which is used to specify a textual cross-reference to some other document, either another module which defines related class definitions, or some other document which provides additional information relevant to this class definition.

9.8. Usage Example

Consider how an event might be described that signals a status change of an interface:

```
class Interface {
  // ...
  attribute speed {
    type        Gauge32;
    access      readonly;
    units       "bps";
    status      current;
    description
         "An estimate of the interface's current bandwidth
          in bits per second.";
  };
  // ...
  attribute adminStatus {
    type        AdminStatus;
    access      readwrite;
    status      current;
    description
         "The desired state of the interface.";
  };
  attribute operStatus {
    type        OperStatus;
    access      readonly;
    status      current;
    description
         "The current operational state of the interface.";
  };

  event linkDown {
    status      current;
    description
            "A linkDown event signifies that the ifOperStatus
             attribute for this interface instance is about to
             enter the down state from some other state (but not
             from the notPresent state).  This other state is
             indicated by the included value of ifOperStatus.";
```

```
    };

    status        current;
    description
              "A physical or logical network interface.";

  };
```

10. Extending a Module

As experience is gained with a module, it may be desirable to revise that module. However, changes are not allowed if they have any potential to cause interoperability problems between an implementation using an original specification and an implementation using an updated specification(s).

For any change, some statements near the top of the module MUST be updated to include information about the revision: specifically, a new `revision' statement (Section 5.6) must be included in front of the `revision' statements. Furthermore, any necessary changes MUST be applied to other statements, including the `organization' and `contact' statements (Section 5.2, Section 5.3).

Note that any definition contained in a module is available to be imported by any other module, and is referenced in an `import' statement via the module name. Thus, a module name MUST NOT be changed. Specifically, the module name (e.g., `ACME-MIB' in the example of Section 5.7) MUST NOT be changed when revising a module (except to correct typographical errors), and definitions MUST NOT be moved from one module to another.

Also note that obsolete definitions MUST NOT be removed from modules since their identifiers may still be referenced by other modules.

A definition may be revised in any of the following ways:

- In `typedef' statement blocks, a `type' statement containing an `Enumeration' or `Bits' type may have new named numbers added.

- In `typedef' statement blocks, the value of a `type' statement may be replaced by another type if the new type is derived (directly or indirectly) from the same base type, has the same set of values, and has identical semantics.

- In `attribute' statements where the `type' sub-statement specifies a class, the class may be replaced by another class if the new class is derived (directly or indirectly) from the base class and both classes have identical semantics.

- In `attribute' statements where the `type' sub-statement specifies a base type, a defined type, or an implicitly derived type (i.e., not a class), that type may be replaced by another type if the new type is derived (directly or indirectly) from the same base type, has the same set of values, and has identical semantics.

- In any statement block, a `status' statement value of `current' may be revised as `deprecated' or `obsolete'. Similarly, a `status' statement value of `deprecated' may be revised as `obsolete'. When making such a change, the `description' statement SHOULD be updated to explain the rationale.

- In `typedef' and `attribute' statement blocks, a `default' statement may be added or updated.

- In `typedef' and `attribute' statement blocks, a `units' statement may be added.

- A class may be augmented by adding new attributes.

- In any statement block, clarifications and additional information may be included in the `description' statement.

- In any statement block, a `reference' statement may be added or updated.

- Entirely new extensions, types, identities, and classes may be defined, using previously unassigned identifiers.

Otherwise, if the semantics of any previous definition are changed (i.e., if a non-editorial change is made to any definition other than those specifically allowed above), then this MUST be achieved by a new definition with a new identifier. In case of a class where the semantics of any attributes are changed, the new class can be defined by derivation from the old class and refining the changed attributes.

Note that changing the identifier associated with an existing definition is considered a semantic change, as these strings may be used in an `import' statement.

11. SMIng Language Extensibility

While the core SMIng language has a well defined set of statements (Section 5 through Section 9.4) that are used to specify those aspects of management information commonly regarded as necessary without management protocol specific information, there may be

further information people wish to express. Describing additional information informally in description statements has a disadvantage in that this information cannot be parsed by any program.

SMIng allows modules to include statements that are unknown to a parser but fulfil some core grammar rules (Section 4.3). Furthermore, additional statements may be defined by the `extension' statement (Section 6). Extensions can be used in the local module or in other modules that import the extension. This has some advantages:

- A parser can differentiate between statements known as extensions and unknown statements. This enables the parser to complain about unknown statements, e.g., due to typos.
- If an extension's definition contains a formal ABNF grammar definition and a parser is able to interpret this ABNF definition, this enables the parser to also complain about the wrong usage of an extension.
- Since there might be some common need for extensions, there is a relatively high probability of extension name collisions originated by different organizations, as long as there is no standardized extension for that purpose. The requirement to explicitly import extension statements allows those extensions to be distinguished.
- The supported extensions of an SMIng implementation, e.g., an SMIng module compiler, can be clearly expressed.

The only formal effect of an extension statement definition is to declare its existence and status, and optionally its ABNF grammar. All additional aspects SHOULD be described in the `description' statement:

- The detailed semantics of the new statement SHOULD be described.
- The contexts in which the new statement can be used SHOULD be described, e.g., a new statement may be designed to be used only in the statement block of a module, but not in other nested statement blocks. Others may be applicable in multiple contexts. In addition, the point in the sequence of an obligatory order of other statements, where the new statement may be inserted, might be prescribed.
- The circumstances that make the new statement mandatory or optional SHOULD be described.

- The syntax of the new statement SHOULD at least be described informally, if not supplied formally in an `abnf' statement.
- It might be reasonable to give some suggestions under which conditions the implementation of the new statement is adequate and how it could be integrated into existent implementations.

Some possible extension applications are:

- The formal mapping of SMIng definitions into the SNMP [RFC3781] framework is defined as an SMIng extension. Other mappings may follow in the future.
- Inlined annotations to definitions. For example, a vendor may wish to describe additional information to class and attribute definitions in private modules. An example are severity levels of events in the statement block of an `event' statement.
- Arbitrary annotations to external definitions. For example, a vendor may wish to describe additional information to definitions in a "standard" module. This allows a vendor to implement "standard" modules as well as additional private features, without redundant module definitions, but on top of "standard" module definitions.

12. Security Considerations

This document defines a language with which to write and read descriptions of management information. The language itself has no security impact on the Internet.

13. Acknowledgements

Since SMIng started as a close successor of SMIv2, some paragraphs and phrases are directly taken from the SMIv2 specifications [RFC2578], [RFC2579], [RFC2580] written by Jeff Case, Keith McCloghrie, David Perkins, Marshall T. Rose, Juergen Schoenwaelder, and Steven L. Waldbusser.

The authors would like to thank all participants of the 7th NMRG meeting held in Schloss Kleinheubach from 6-8 September 2000, which was a major step towards the current status of this memo, namely Heiko Dassow, David Durham, Keith McCloghrie, and Bert Wijnen.

Furthermore, several discussions within the SMING Working Group reflected experience with SMIv2 and influenced this specification at some points.

14. References

14.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

14.2. Informative References

[RFC3216] Elliott, C., Harrington, D., Jason, J., Schoenwaelder, J., Strauss, F. and W. Weiss, "SMIng Objectives", RFC 3216, December 2001.

[RFC3781] Strauss, F. and J. Schoenwaelder, "Next Generation Structure of Management Information (SMIng) Mappings to the Simple Network Management Protocol (SNMP)", RFC 3781, May 2004.

[RFC2578] McCloghrie, K., Perkins, D. and J. Schoenwaelder, "Structure of Management Information Version 2 (SMIv2)", STD 58, RFC 2578, April 1999.

[RFC2579] McCloghrie, K., Perkins, D. and J. Schoenwaelder, "Textual Conventions for SMIv2", STD 59, RFC 2579, April 1999.

[RFC2580] McCloghrie, K., Perkins, D. and J. Schoenwaelder, "Conformance Statements for SMIv2", STD 60, RFC 2580, April 1999.

[RFC3159] McCloghrie, K., Fine, M., Seligson, J., Chan, K., Hahn, S., Sahita, R., Smith, A. and F. Reichmeyer, "Structure of Policy Provisioning Information (SPPI)", RFC 3159, August 2001.

[RFC1155] Rose, M. and K. McCloghrie, "Structure and Identification of Management Information for TCP/IP-based Internets", STD 16, RFC 1155, May 1990.

[RFC1212] Rose, M. and K. McCloghrie, "Concise MIB Definitions", STD 16, RFC 1212, March 1991.

[RFC1215] Rose, M., "A Convention for Defining Traps for use with the SNMP", RFC 1215, March 1991.

[ASN1] International Organization for Standardization, "Specification of Abstract Syntax Notation One (ASN.1)", International Standard 8824, December 1987.

[RFC3411] Harrington, D., Presuhn, R. and B. Wijnen, "An Architecture for Describing Simple Network Management Protocol (SNMP) Management Frameworks", STD 62, RFC 3411, December 2002.

[IEEE754] Institute of Electrical and Electronics Engineers, "IEEE Standard for Binary Floating-Point Arithmetic", ANSI/IEEE Standard 754-1985, August 1985.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3084] Chan, K., Seligson, J., Durham, D., Gai, S., McCloghrie, K., Herzog, S., Reichmeyer, F., Yavatkar, R. and A. Smith, "COPS Usage for Policy Provisioning", RFC 3084, March 2001.

Appendix A. NMRG-SMING Module

Most SMIng modules are built on top of the definitions of some commonly used derived types. The definitions of these derived types are contained in the NMRG-SMING module which is defined below. Its derived types are generally applicable for modeling all areas of management information. Among these derived types are counter types, string types, and date and time related types.

This module is derived from RFC 2578 [RFC2578] and RFC 2579 [RFC2579].

```
module NMRG-SMING {

    organization    "IRTF Network Management Research Group (NMRG)";

    contact         "IRTF Network Management Research Group (NMRG)
                     http://www.ibr.cs.tu-bs.de/projects/nmrg/

                     Frank Strauss
                     TU Braunschweig
                     Muehlenpfordtstrasse 23
                     38106 Braunschweig
                     Germany
                     Phone: +49 531 391 3266
                     EMail: strauss@ibr.cs.tu-bs.de

                     Juergen Schoenwaelder
                     International University Bremen
                     P.O. Box 750 561
                     28725 Bremen
                     Germany
                     Phone: +49 421 200 3587
                     EMail: j.schoenwaelder@iu-bremen.de";

    description     "Core type definitions for SMIng. Several
                     type definitions are SMIng versions of
                     similar SMIv2 or SPPI definitions.

                     Copyright (C) The Internet Society (2004).
                     All Rights Reserved.
                     This version of this module is part of
                     RFC 3780, see the RFC itself for full
                     legal notices.";
```

```
    revision {
        date        "2003-12-16";
        description "Initial revision, published as RFC 3780.";
    };

    typedef Gauge32 {
        type        Unsigned32;
        description
           "The Gauge32 type represents a non-negative integer,
            which may increase or decrease, but shall never
            exceed a maximum value, nor fall below a minimum
            value.  The maximum value can not be greater than
            2^32-1 (4294967295 decimal), and the minimum value
            can not be smaller than 0.  The value of a Gauge32
            has its maximum value whenever the information
            being modeled is greater than or equal to its
            maximum value, and has its minimum value whenever
            the information being modeled is smaller than or
            equal to its minimum value.  If the information
            being modeled subsequently decreases below
            (increases above) the maximum (minimum) value, the
            Gauge32 also decreases (increases).";
        reference
           "RFC 2578, Sections 2. and 7.1.7.";
    };

    typedef Counter32 {
        type        Unsigned32;
        description
           "The Counter32 type represents a non-negative integer
            which monotonically increases until it reaches a
            maximum value of 2^32-1 (4294967295 decimal), when it
            wraps around and starts increasing again from zero.

            Counters have no defined `initial' value, and thus, a
            single value of a Counter has (in general) no information
            content.  Discontinuities in the monotonically increasing
            value normally occur at re-initialization of the
            management system, and at other times as specified in the
            description of an attribute using this type.  If such
            other times can occur, for example, the creation of a
            class instance that contains an attribute of type
            Counter32 at times other than re-initialization, then a
            corresponding attribute should be defined, with an
            appropriate type, to indicate the last discontinuity.
            Examples of appropriate types include: TimeStamp32,
            TimeStamp64, DateAndTime, TimeTicks32 or TimeTicks64
            (other types defined in this module).
```

```
            The value of the access statement for attributes with
            a type value of Counter32 should be either `readonly'
            or `eventonly'.

            A default statement should not be used for attributes
            with a type value of Counter32.";
        reference
           "RFC 2578, Sections 2. and 7.1.6.";
    };

    typedef Gauge64 {
        type        Unsigned64;
        description
           "The Gauge64 type represents a non-negative integer,
            which may increase or decrease, but shall never
            exceed a maximum value, nor fall below a minimum
            value.  The maximum value can not be greater than
            2^64-1 (18446744073709551615), and the minimum value
            can not be smaller than 0.  The value of a Gauge64
            has its maximum value whenever the information
            being modeled is greater than or equal to its
            maximum value, and has its minimum value whenever
            the information being modeled is smaller than or
            equal to its minimum value.  If the information
            being modeled subsequently decreases below
            (increases above) the maximum (minimum) value, the
            Gauge64 also decreases (increases).";
    };

    typedef Counter64 {
        type        Unsigned64;
        description
           "The Counter64 type represents a non-negative integer
            which monotonically increases until it reaches a
            maximum value of 2^64-1 (18446744073709551615), when
            it wraps around and starts increasing again from zero.

            Counters have no defined `initial' value, and thus, a
            single value of a Counter has (in general) no
            information content.  Discontinuities in the
            monotonically increasing value normally occur at
            re-initialization of the management system, and at
            other times as specified in the description of an
            attribute using this type.  If such other times can
            occur, for example, the creation of a class
            instance that contains an attribute of type Counter32
            at times other than re-initialization, then
            a corresponding attribute should be defined, with an
```

```
            appropriate type, to indicate the last discontinuity.
            Examples of appropriate types include: TimeStamp32,
            TimeStamp64, DateAndTime, TimeTicks32 or TimeTicks64
            (other types defined in this module).

            The value of the access statement for attributes with
            a type value of Counter64 should be either `readonly'
            or `eventonly'.

            A default statement should not be used for attributes
            with a type value of Counter64.";
        reference
           "RFC 2578, Sections 2. and 7.1.10.";
    };

    typedef Opaque {
        type        OctetString;
        status      obsolete;
        description
           "******* THIS TYPE DEFINITION IS OBSOLETE *******

            The Opaque type is provided solely for
            backward-compatibility, and shall not be used for
            newly-defined attributes and derived types.

            The Opaque type supports the capability to pass
            arbitrary ASN.1 syntax.  A value is encoded using
            the ASN.1 Basic Encoding Rules into a string of
            octets.  This, in turn, is encoded as an
            OctetString, in effect `double-wrapping' the
            original ASN.1 value.

            Note that a conforming implementation need only be
            able to accept and recognize opaquely-encoded data.
            It need not be able to unwrap the data and then
            interpret its contents.

            A requirement on `standard' modules is that no
            attribute may have a type value of Opaque and no
            type may be derived from the Opaque type.";
        reference
           "RFC 2578, Sections 2. and 7.1.9.";
    };

    typedef IpAddress {
        type        OctetString (4);
        status      deprecated;
        description
```

```
           "******* THIS TYPE DEFINITION IS DEPRECATED *******

            The IpAddress type represents a 32-bit Internet
            IPv4 address.  It is represented as an OctetString
            of length 4, in network byte-order.

            Note that the IpAddress type is present for
            historical reasons.";
        reference
           "RFC 2578, Sections 2. and 7.1.5.";
    };

    typedef TimeTicks32 {
        type        Unsigned32;
        description
           "The TimeTicks32 type represents a non-negative integer
            which represents the time, modulo 2^32 (4294967296
            decimal), in hundredths of a second between two epochs.
            When attributes are defined which use this type, the
            description of the attribute identifies both of the
            reference epochs.

            For example, the TimeStamp32 type (defined in this
            module) is based on the TimeTicks32 type.";
        reference
           "RFC 2578, Sections 2. and 7.1.8.";
    };

    typedef TimeTicks64 {
        type        Unsigned64;
        description
           "The TimeTicks64 type represents a non-negative integer
            which represents the time, modulo 2^64
            (18446744073709551616 decimal), in hundredths of a second
            between two epochs.  When attributes are defined which use
            this type, the description of the attribute identifies
            both of the reference epochs.

            For example, the TimeStamp64 type (defined in this
            module) is based on the TimeTicks64 type.";
    };

    typedef TimeStamp32 {
        type        TimeTicks32;
        description
           "The value of an associated TimeTicks32 attribute at
            which a specific occurrence happened.  The specific
            occurrence must be defined in the description of any
```

```
            attribute defined using this type.  When the specific
            occurrence occurred prior to the last time the
            associated TimeTicks32 attribute was zero, then the
            TimeStamp32 value is zero.  Note that this requires all
            TimeStamp32 values to be reset to zero when the value of
            the associated TimeTicks32 attribute reaches 497+ days
            and wraps around to zero.

            The associated TimeTicks32 attribute should be specified
            in the description of any attribute using this type.
            If no TimeTicks32 attribute has been specified, the
            default scalar attribute sysUpTime is used.";
        reference
           "RFC 2579, Section 2.";
    };

    typedef TimeStamp64 {
        type         TimeTicks64;
        description
           "The value of an associated TimeTicks64 attribute at which
            a specific occurrence happened.  The specific occurrence
            must be defined in the description of any attribute
            defined using this type.  When the specific occurrence
            occurred prior to the last time the associated TimeTicks64
            attribute was zero, then the TimeStamp64 value is zero.
            The associated TimeTicks64 attribute must be specified in
            the description of any attribute using this
            type. TimeTicks32 attributes must not be used as
            associated attributes.";
    };

    typedef TimeInterval32 {
        type         Integer32 (0..2147483647);
        description
           "A period of time, measured in units of 0.01 seconds.

            The TimeInterval32 type uses Integer32 rather than
            Unsigned32 for compatibility with RFC 2579.";
        reference
           "RFC 2579, Section 2.";
    };

    typedef TimeInterval64 {
        type         Integer64;
        description
           "A period of time, measured in units of 0.01 seconds.
            Note that negative values are allowed.";
    };
```

```
    typedef DateAndTime {
        type        OctetString (8 | 11);
        default     0x00000000000000000000;
        format      "2d-1d-1d,1d:1d:1d.1d,1a1d:1d";
        description
           "A date-time specification.

            field  octets  contents                  range
            -----  ------  --------                  -----
             1      1-2   year*                     0..65535
             2       3    month                     1..12 | 0
             3       4    day                       1..31 | 0
             4       5    hour                      0..23
             5       6    minutes                   0..59
             6       7    seconds                   0..60
                          (use 60 for leap-second)
             7       8    deci-seconds              0..9
             8       9    direction from UTC        '+' / '-'
             9      10    hours from UTC*           0..13
            10      11    minutes from UTC          0..59

            * Notes:
            - the value of year is in big-endian encoding
            - daylight saving time in New Zealand is +13

            For example, Tuesday May 26, 1992 at 1:30:15 PM EDT would
            be displayed as:

                         1992-5-26,13:30:15.0,-4:0

            Note that if only local time is known, then timezone
            information (fields 8-10) is not present.

            The two special values of 8 or 11 zero bytes denote an
            unknown date-time specification.";
        reference
           "RFC 2579, Section 2.";
    };

    typedef TruthValue {
        type        Enumeration (true(1), false(2));
        description
           "Represents a boolean value.";
        reference
           "RFC 2579, Section 2.";
    };

    typedef PhysAddress {
```

```
        type        OctetString;
        format      "1x:";
        description
           "Represents media- or physical-level addresses.";
        reference
           "RFC 2579, Section 2.";
    };

    typedef MacAddress {
        type        OctetString (6);
        format      "1x:";
        description
           "Represents an IEEE 802 MAC address represented in the
            `canonical' order defined by IEEE 802.1a, i.e., as if it
            were transmitted least significant bit first, even though
            802.5 (in contrast to other 802.x protocols) requires MAC
            addresses to be transmitted most significant bit first.";
        reference
           "RFC 2579, Section 2.";
    };

    // The DisplayString definition below does not impose a size
    // restriction and is thus not the same as the DisplayString
    // definition in RFC 2579. The DisplayString255 definition is
    // provided for mapping purposes.

    typedef DisplayString {
        type        OctetString;
        format      "1a";
        description
           "Represents textual information taken from the NVT ASCII
            character set, as defined in pages 4, 10-11 of RFC 854.

            To summarize RFC 854, the NVT ASCII repertoire specifies:

             - the use of character codes 0-127 (decimal)

             - the graphics characters (32-126) are interpreted as
               US ASCII

             - NUL, LF, CR, BEL, BS, HT, VT and FF have the special
               meanings specified in RFC 854

             - the other 25 codes have no standard interpretation

             - the sequence 'CR LF' means newline

             - the sequence 'CR NUL' means carriage-return
```

```
            - an 'LF' not preceded by a 'CR' means moving to the
              same column on the next line.

            - the sequence 'CR x' for any x other than LF or NUL is
              illegal.  (Note that this also means that a string may
              end with either 'CR LF' or 'CR NUL', but not with CR.)
        ";
    };

    typedef DisplayString255 {
        type        DisplayString (0..255);
        description
           "A DisplayString with a maximum length of 255 characters.
            Any attribute defined using this syntax may not exceed 255
            characters in length.

            The DisplayString255 type has the same semantics as the
            DisplayString textual convention defined in RFC 2579.";
        reference
           "RFC 2579, Section 2.";
    };

    // The Utf8String and Utf8String255 definitions below facilitate
    // internationalization. The definition is consistent with the
    // definition of SnmpAdminString in RFC 2571.

    typedef Utf8String {
        type        OctetString;
        format      "65535t";      // is there a better way ?
        description
           "A human readable string represented using the ISO/IEC IS
            10646-1 character set, encoded as an octet string using
            the UTF-8 transformation format described in RFC 3629.

            Since additional code points are added by amendments to
            the 10646 standard from time to time, implementations must
            be prepared to encounter any code point from 0x00000000 to
            0x7fffffff.  Byte sequences that do not correspond to the
            valid UTF-8 encoding of a code point or are outside this
            range are prohibited.

            The use of control codes should be avoided. When it is
            necessary to represent a newline, the control code
            sequence CR LF should be used.

            The use of leading or trailing white space should be
            avoided.
```

```
            For code points not directly supported by user interface
            hardware or software, an alternative means of entry and
            display, such as hexadecimal, may be provided.

            For information encoded in 7-bit US-ASCII, the UTF-8
            encoding is identical to the US-ASCII encoding.

            UTF-8 may require multiple bytes to represent a single
            character / code point; thus the length of a Utf8String in
            octets may be different from the number of characters
            encoded.  Similarly, size constraints refer to the number
            of encoded octets, not the number of characters
            represented by an encoding.";
    };

    typedef Utf8String255 {
        type        Utf8String (0..255);
        format      "255t";
        description
           "A Utf8String with a maximum length of 255 octets.  Note
            that the size of an Utf8String is measured in octets, not
            characters.";
    };

    identity null {
        description
           "An identity used to represent null pointer values.";
    };

};
```

Appendix B. SMIng ABNF Grammar

The SMIng grammar conforms to the Augmented Backus-Naur Form (ABNF) [RFC2234].

```
;;
;; sming.abnf -- SMIng grammar in ABNF notation (RFC 2234).
;;
;; @(#) $Id: sming.abnf,v 1.33 2003/10/23 19:31:55 strauss Exp $
;;
;; Copyright (C) The Internet Society (2004). All Rights Reserved.
;;

smingFile               = optsep *(moduleStatement optsep)

;;
;; Statement rules.
```

```
;;

moduleStatement         = moduleKeyword sep ucIdentifier optsep
                              "{" stmtsep
                              *(importStatement stmtsep)
                              organizationStatement stmtsep
                              contactStatement stmtsep
                              descriptionStatement stmtsep
                              *1(referenceStatement stmtsep)
                              1*(revisionStatement stmtsep)
                              *(extensionStatement stmtsep)
                              *(typedefStatement stmtsep)
                              *(identityStatement stmtsep)
                              *(classStatement stmtsep)
                          "}" optsep ";"

extensionStatement      = extensionKeyword sep lcIdentifier optsep
                              "{" stmtsep
                              statusStatement stmtsep
                              descriptionStatement stmtsep
                              *1(referenceStatement stmtsep)
                              *1(abnfStatement stmtsep)
                          "}" optsep ";"

typedefStatement        = typedefKeyword sep ucIdentifier optsep
                              "{" stmtsep
                              typedefTypeStatement stmtsep
                              *1(defaultStatement stmtsep)
                              *1(formatStatement stmtsep)
                              *1(unitsStatement stmtsep)
                              statusStatement stmtsep
                              descriptionStatement stmtsep
                              *1(referenceStatement stmtsep)
                          "}" optsep ";"

identityStatement       = identityStmtKeyword sep lcIdentifier optsep
                              "{" stmtsep
                              *1(parentStatement stmtsep)
                              statusStatement stmtsep
                              descriptionStatement stmtsep
                              *1(referenceStatement stmtsep)
                          "}" optsep ";"

classStatement          = classKeyword sep ucIdentifier optsep
                              "{" stmtsep
                              *1(extendsStatement stmtsep)
                              *(attributeStatement stmtsep)
                              *1(uniqueStatement stmtsep)
```

```
                            *(eventStatement stmtsep)
                            statusStatement stmtsep
                            descriptionStatement stmtsep
                            *1(referenceStatement stmtsep)
                          "}" optsep ";"

attributeStatement      = attributeKeyword sep
                            lcIdentifier optsep
                            "{" stmtsep
                            typeStatement stmtsep
                            *1(accessStatement stmtsep)
                            *1(defaultStatement stmtsep)
                            *1(formatStatement stmtsep)
                            *1(unitsStatement stmtsep)
                            statusStatement stmtsep
                            descriptionStatement stmtsep
                            *1(referenceStatement stmtsep)
                          "}" optsep ";"

uniqueStatement         = uniqueKeyword optsep
                            "(" optsep qlcIdentifierList
                            optsep ")" optsep ";"

eventStatement          = eventKeyword sep lcIdentifier
                            optsep "{" stmtsep
                            statusStatement stmtsep
                            descriptionStatement stmtsep
                            *1(referenceStatement stmtsep)
                          "}" optsep ";"

importStatement         = importKeyword sep ucIdentifier optsep
                            "(" optsep
                            identifierList optsep
                          ")" optsep ";"

revisionStatement       = revisionKeyword optsep "{" stmtsep
                            dateStatement stmtsep
                            descriptionStatement stmtsep
                          "}" optsep ";"

typedefTypeStatement    = typeKeyword sep refinedBaseType optsep ";"

typeStatement           = typeKeyword sep
                          (refinedBaseType / refinedType) optsep ";"

parentStatement         = parentKeyword sep qlcIdentifier optsep ";"

extendsStatement        = extendsKeyword sep qucIdentifier optsep ";"
```

```
dateStatement           = dateKeyword sep date optsep ";"

organizationStatement   = organizationKeyword sep text optsep ";"

contactStatement        = contactKeyword sep text optsep ";"

formatStatement         = formatKeyword sep format optsep ";"

unitsStatement          = unitsKeyword sep units optsep ";"

statusStatement         = statusKeyword sep status optsep ";"

accessStatement         = accessKeyword sep access optsep ";"

defaultStatement        = defaultKeyword sep anyValue optsep ";"

descriptionStatement    = descriptionKeyword sep text optsep ";"

referenceStatement      = referenceKeyword sep text optsep ";"

abnfStatement           = abnfKeyword sep text optsep ";"

;;
;;
;;

refinedBaseType         = ObjectIdentifierKeyword /
                          OctetStringKeyword *1(optsep numberSpec) /
                          PointerKeyword *1(optsep pointerSpec) /
                          Integer32Keyword *1(optsep numberSpec) /
                          Unsigned32Keyword *1(optsep numberSpec) /
                          Integer64Keyword *1(optsep numberSpec) /
                          Unsigned64Keyword *1(optsep numberSpec) /
                          Float32Keyword *1(optsep floatSpec) /
                          Float64Keyword *1(optsep floatSpec) /
                          Float128Keyword *1(optsep floatSpec) /
                          EnumerationKeyword
                                      optsep namedSignedNumberSpec /
                          BitsKeyword optsep namedNumberSpec

refinedType             = qucIdentifier *1(optsep anySpec)

anySpec                 = pointerSpec / numberSpec / floatSpec

pointerSpec             = "(" optsep qlcIdentifier optsep ")"
```

```
numberSpec              = "(" optsep numberElement
                              *furtherNumberElement
                              optsep ")"

furtherNumberElement    = optsep "|" optsep numberElement

numberElement           = signedNumber *1numberUpperLimit

numberUpperLimit        = optsep ".." optsep signedNumber

floatSpec               = "(" optsep floatElement
                              *furtherFloatElement
                              optsep ")"

furtherFloatElement     = optsep "|" optsep floatElement

floatElement            = floatValue *1floatUpperLimit

floatUpperLimit         = optsep ".." optsep floatValue

namedNumberSpec         = "(" optsep namedNumberList optsep ")"

namedNumberList         = namedNumberItem
                              *(optsep "," optsep namedNumberItem)

namedNumberItem         = lcIdentifier optsep "(" optsep number
                              optsep ")"

namedSignedNumberSpec   = "(" optsep namedSignedNumberList optsep ")"

namedSignedNumberList   = namedSignedNumberItem
                              *(optsep "," optsep
                                           namedSignedNumberItem)

namedSignedNumberItem   = lcIdentifier optsep "(" optsep signedNumber
                              optsep ")"

identifierList          = identifier
                              *(optsep "," optsep identifier)

qIdentifierList         = qIdentifier
                              *(optsep "," optsep qIdentifier)

qlcIdentifierList       = qlcIdentifier
                              *(optsep "," optsep qlcIdentifier)

bitsValue               = "(" optsep bitsList optsep ")"
```

```
bitsList                = *1(lcIdentifier
                             *(optsep "," optsep lcIdentifier))

;;
;; Other basic rules.
;;
identifier              = ucIdentifier / lcIdentifier

qIdentifier             = qucIdentifier / qlcIdentifier

ucIdentifier            = ucAlpha *63(ALPHA / DIGIT / "-")

qucIdentifier           = *1(ucIdentifier "::") ucIdentifier

lcIdentifier            = lcAlpha *63(ALPHA / DIGIT / "-")

qlcIdentifier           = *1(ucIdentifier "::") lcIdentifier

attrIdentifier          = lcIdentifier *("." lcIdentifier)

qattrIdentifier         = *1(ucIdentifier ".") attrIdentifier

cattrIdentifier         = ucIdentifier "."
                             lcIdentifier *("." lcIdentifier)

qcattrIdentifier        = qucIdentifier "."
                             lcIdentifier *("." lcIdentifier)

text                    = textSegment *(optsep textSegment)

textSegment             = DQUOTE *textAtom DQUOTE
                          ; See Section 4.2.

textAtom                = textVChar / HTAB / SP / lineBreak

date                    = DQUOTE 4DIGIT "-" 2DIGIT "-" 2DIGIT
                             *1(" " 2DIGIT ":" 2DIGIT)
                             DQUOTE
                          ; always in UTC

format                  = textSegment

units                   = textSegment

anyValue                = bitsValue /
                          signedNumber /
                          hexadecimalNumber /
```

```
                          floatValue /
                          text /
                          objectIdentifier
                          ; Note: `objectIdentifier' includes the
                          ; syntax of enumeration labels and
                          ; identities.
                          ; They are not named literally to
                          ; avoid reduce/reduce conflicts when
                          ; building LR parsers based on this
                          ; grammar.

status                  = currentKeyword /
                          deprecatedKeyword /
                          obsoleteKeyword

access                  = eventonlyKeyword /
                          readonlyKeyword /
                          readwriteKeyword

objectIdentifier        = (qlcIdentifier / subid "." subid)
                              *127("." subid)

subid                   = decimalNumber

number                  = hexadecimalNumber / decimalNumber

negativeNumber          = "-" decimalNumber

signedNumber            = number / negativeNumber

decimalNumber           = "0" / (nonZeroDigit *DIGIT)

zeroDecimalNumber       = 1*DIGIT

hexadecimalNumber       = %x30 %x78 ; "0x" with x only lower-case
                          1*(HEXDIG HEXDIG)

floatValue              = neginfKeyword /
                          posinfKeyword /
                          snanKeyword /
                          qnanKeyword /
                          signedNumber "." zeroDecimalNumber
                              *1("E" ("+"/"-") zeroDecimalNumber)

;;
;; Rules to skip unknown statements
;; with arbitrary arguments and blocks.
;;
```

```
unknownStatement        = unknownKeyword optsep *unknownArgument
                              optsep ";"

unknownArgument         = ("(" optsep unknownList optsep ")") /
                          ("{" optsep *unknownStatement optsep "}") /
                          qucIdentifier /
                          anyValue /
                          anySpec

unknownList             = namedNumberList /
                          qIdentifierList

unknownKeyword          = lcIdentifier

;;
;; Keyword rules.
;;
;; Typically, keywords are represented by tokens returned from the
;; lexical analyzer.  Note, that the lexer has to be stateful to
;; distinguish keywords from identifiers depending on the context
;; position in the input stream.
;;

moduleKeyword       =  %x6D %x6F %x64 %x75 %x6C %x65
importKeyword       =  %x69 %x6D %x70 %x6F %x72 %x74
revisionKeyword     =  %x72 %x65 %x76 %x69 %x73 %x69 %x6F %x6E
dateKeyword         =  %x64 %x61 %x74 %x65
organizationKeyword =  %x6F %x72 %x67 %x61 %x6E %x69 %x7A %x61 %x74
                       %x69 %x6F %x6E
contactKeyword      =  %x63 %x6F %x6E %x74 %x61 %x63 %x74
descriptionKeyword  =  %x64 %x65 %x73 %x63 %x72 %x69 %x70 %x74 %x69
                       %x6F %x6E
referenceKeyword    =  %x72 %x65 %x66 %x65 %x72 %x65 %x6E %x63 %x65
extensionKeyword    =  %x65 %x78 %x74 %x65 %x6E %x73 %x69 %x6F %x6E
typedefKeyword      =  %x74 %x79 %x70 %x65 %x64 %x65 %x66
typeKeyword         =  %x74 %x79 %x70 %x65
parentKeyword       =  %x70 %x61 %x72 %x65 %x6E %x74
identityStmtKeyword =  %x69 %x64 %x65 %x6E %x74 %x69 %x74 %x79
classKeyword        =  %x63 %x6C %x61 %x73 %x73
extendsKeyword      =  %x65 %x78 %x74 %x65 %x6E %x64 %x73
attributeKeyword    =  %x61 %x74 %x74 %x72 %x69 %x62 %x75 %x74 %x65
uniqueKeyword       =  %x75 %x6E %x69 %x71 %x75 %x65
eventKeyword        =  %x65 %x76 %x65 %x6E %x74
formatKeyword       =  %x66 %x6F %x72 %x6D %x61 %x74
unitsKeyword        =  %x75 %x6E %x69 %x74 %x73
statusKeyword       =  %x73 %x74 %x61 %x74 %x75 %x73
accessKeyword       =  %x61 %x63 %x63 %x65 %x73 %x73
defaultKeyword      =  %x64 %x65 %x66 %x61 %x75 %x6C %x74
```

```
abnfKeyword          =  %x61 %x62 %x6E %x66

;; Base type keywords.

OctetStringKeyword   =  %x4F %x63 %x74 %x65 %x74 %x53 %x74 %x72 %x69
                        %x6E %x67
PointerKeyword       =  %x50 %x6F %x69 %x6E %x74 %x65 %x72
ObjectIdentifierKeyword  =  %x4F %x62 %x6A %x65 %x63 %x74 %x49 %x64
                        %x65 %x6E %x74 %x69 %x66 %x69 %x65 %x72
Integer32Keyword     =  %x49 %x6E %x74 %x65 %x67 %x65 %x72 %x33 %x32
Unsigned32Keyword    =  %x55 %x6E %x73 %x69 %x67 %x6E %x65 %x64 %x33
                        %x32
Integer64Keyword     =  %x49 %x6E %x74 %x65 %x67 %x65 %x72 %x36 %x34
Unsigned64Keyword    =  %x55 %x6E %x73 %x69 %x67 %x6E %x65 %x64 %x36
                        %x34
Float32Keyword       =  %x46 %x6C %x6F %x61 %x74 %x33 %x32
Float64Keyword       =  %x46 %x6C %x6F %x61 %x74 %x36 %x34
Float128Keyword      =  %x46 %x6C %x6F %x61 %x74 %x31 %x32 %x38
BitsKeyword          =  %x42 %x69 %x74 %x73
EnumerationKeyword   =  %x45 %x6E %x75 %x6D %x65 %x72 %x61 %x74 %x69
                        %x6F %x6E

;; Status keywords.

currentKeyword       =  %x63 %x75 %x72 %x72 %x65 %x6E %x74
deprecatedKeyword    =  %x64 %x65 %x70 %x72 %x65 %x63 %x61 %x74 %x65
                        %x64
obsoleteKeyword      =  %x6F %x62 %x73 %x6F %x6C %x65 %x74 %x65

;; Access keywords.

eventonlyKeyword     =  %x65 %x76 %x65 %x6E %x74 %x6F %x6E %x6C %x79
readonlyKeyword      =  %x72 %x65 %x61 %x64 %x6F %x6E %x6C %x79
readwriteKeyword     =  %x72 %x65 %x61 %x64 %x77 %x72 %x69 %x74 %x65

;; Special floating point values' keywords.

neginfKeyword        =  %x6E %x65 %x67 %x69 %x6E %x66
posinfKeyword        =  %x70 %x6F %x73 %x69 %x6E %x66
snanKeyword          =  %x73 %x6E %x61 %x6E
qnanKeyword          =  %x71 %x6E %x61 %x6E

;;
;; Some low level rules.
;; These tokens are typically skipped by the lexical analyzer.
;;
```

```
sep                     = 1*(comment / lineBreak / WSP)
                          ; unconditional separator

optsep                  = *(comment / lineBreak / WSP)

stmtsep                 = *(comment /
                            lineBreak /
                            WSP /
                            unknownStatement)

comment                 = "//" *(WSP / VCHAR) lineBreak

lineBreak               = CRLF / LF

;;
;; Encoding specific rules.
;;

textVChar               = %x21 / %x23-7E
                          ; any VCHAR except DQUOTE

ucAlpha                 = %x41-5A

lcAlpha                 = %x61-7A

nonZeroDigit            = %x31-39

;;
;; RFC 2234 core rules.
;;

ALPHA          =  %x41-5A / %x61-7A
                       ; A-Z / a-z

CR             =  %x0D
                       ; carriage return

CRLF           =  CR LF
                       ; Internet standard newline

DIGIT          =  %x30-39
                       ; 0-9

DQUOTE         =  %x22
                       ; " (Double Quote)

HEXDIG         =  DIGIT /
                  %x61 / %x62 / %x63 / %x64 / %x65 / %x66
```

```
                        ; only lower-case a..f

HTAB            =  %x09
                        ; horizontal tab

LF              =  %x0A
                        ; linefeed

SP              =  %x20
                        ; space

VCHAR           =  %x21-7E
                        ; visible (printing) characters

WSP             =  SP / HTAB
                        ; white space

;; End of ABNF
```

Authors' Addresses

Frank Strauss
TU Braunschweig
Muehlenpfordtstrasse 23
38106 Braunschweig
Germany

Phone: +49 531 391 3266
EMail: strauss@ibr.cs.tu-bs.de
URI: http://www.ibr.cs.tu-bs.de/

Juergen Schoenwaelder
International University Bremen
P.O. Box 750 561
28725 Bremen
Germany

Phone: +49 421 200 3587
EMail: j.schoenwaelder@iu-bremen.de
URI: http://www.eecs.iu-bremen.de/

Full Copyright Statement

Intellectual Property

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.